OTERRO

DATABASE ENGINE

Version 11

Help Manual

R:BASE Technologies, Inc.

Oterro 11 for Windows

Help Manual

by R:BASE Technologies, Inc.

Welcome to Oterro 11 for Windows!

A high-end performance solution for database application developers, Oterro is a relational ODBC database engine ideal for use in a file erver environment, with both 64-bit and 32-bit drivers included. Oterro is the solution for programmers who are exceeding the limits of heir current engine. It's the sophisticated high-end database engine that you won't outgrow. This engine has been optimized for serious custom application development. Oterro adheres to Codd's theory of true relational database management. Oterro is able to operate in a omplex multi-user environment at high capacity and high speed. In short, Oterro 11 is the ODBC driver for the R:BASE database.

Table of Contents

Copyright © 1982-2025 R:BASE Technologies, Inc.

Copyright © 1982-2025 R:BASE Technologies, Inc.

Copyright © 1982-2025 R:BASE Technologies, Inc.

Copyright © 1982-2025 R:BASE Technologies, Inc.

Part I

1 Introduction

1.1 Introducing Oterro 11

A high-end performance solution for database application developers, Oterro is a full relational ODBC database engine ideal for use in a file server environment, with both 64-bit and 32-bit drivers included. Oterro is the solution for programmers who are exceeding the limits of their current engine. It's the sophisticated high-end database engine that you won't outgrow. This engine has been optimized for serious custom application development. Oterro adheres to Codd's theory of true relational database management. Oterro is able to operate in a complex multi-user environment at high capacity and high speed. In short, Oterro 11 is the ODBC driver for the R:BASE 11 database.

The Oterro Engine provides an interface between an application you create and a database. This version of the Oterro Engine is intended for application developers who want to use other development tools for creating powerful database management systems (DBMS) applications.

The Oterro Engine conforms to ANSI 1989 Level 2 SQL with 1992 extensions. The Oterro Engine is written to the Microsoft Open Database Connectivity (ODBC) specification (Level 3), and therefore operates within the ODBC framework.

1.2 Copyrights

Information in this document, including URL and other Internet web site references, is subject to change without notice. The example companies, individuals, products, organizations and events depicted herein are completely fictitious. Any similarity to a company, individual, product, organization or event is completely unintentional. R:BASE Technologies, Inc. shall not be liable for errors contained herein or for incidental consequential damages in connection with the furnishing, performance, or use of this material. This document contains proprietary information, which is protected by copyright. Complying with all applicable copyright laws is the responsibility of the user. Without limiting the rights under copyright, no part of this document may be reproduced, stored in or introduced into a retrieval system, or transmitted in any form or by any means (electronic, mechanical, photocopying, recording, or otherwise), or for any purpose, without the express written consent of R:BASE Technologies, Inc. We reserve the right to make changes from time to time in the contents hereof without obligation to notify any person of such revision or changes. We also reserve the right to change the specification without notice and may therefore not coincide with the contents of this document. The manufacturer assumes no responsibilities with regard to the performance or use of third party products.

Products that are referred to in this document may be either trademarks and/or registered trademarks of the respective owners. The publisher and the author make no claim to these trademarks.

The software described in this document is furnished under a license agreement. The software may be used or copied only in accordance with the terms of that agreement. Any unauthorized use or duplication of the software is forbidden.

R:BASE Technologies, Inc. may have patents, patent applications, trademarks, copyrights, or other intellectual property rights covering subject matter in this document. Except as expressly provided in any written license agreement from R:BASE Technologies, Inc., the furnishing of this document does not give you any license to these patents, trademarks, copyrights, or other intellectual property.

Trademarks

R:BASE®, Oterro®, RBAdmin®, R:Scope®, R:Mail®, R:Charts®, R:Spell Checker®, R:Docs®, R:BASE Editor®, R:BASE Plugin Power Pack®, R:Style®, RBZip®, R:Mail Editor®, R:BASE Dependency Viewer®, R:Archive®, R:Chat®, R:PDF Form Filler®, R:FTPClient®, R:SFTPClient®, R:PDFWorks®, R:Magellan®, R:WEB Reports®, R:WEB Gateway®, R:PDFMerge®, R:PDFSearch®, R:Documenter®, RBInstaller®, RBUpdater®, R:AmazonS3®, R:GAP®, R:Mail Viewer®, R:Capture®, R:Synchronizer®, R:Biometric®, R:CAD Viewer®, R:DXF®, R:Twain2PDF®, R:Scheduler®, R:Scribbler®, R:SmartSig®, R:OutLink®, R:HASH®, R:JobTrack®, R:TimeTrack®, R:Manufacturing®, R:QBDataDirect®, R:QBSynchronizer®, and R:QBDBExtractor®, and Pocket R:BASE® are trademarks or registered trademarks of R:BASE Technologies, Inc. All Rights Reserved. All other brand, product names, company names and logos are trademarks or registered trademarks of their respective companies.

Windows, Windows 11-10, Windows Server 2025-2016, Azure Maps, Word, Excel, Access, SQL Server, and Outlook are registered trademarks of Microsoft Corporation. OpenOffice is a registered trademark of the Apache Software Foundation.

Printed: July 2025 in Murrysville, PA

First Edition

1.3 R:BASE for Windows

R:BASE is an Industrial-Strength, Multi-User Relational Database. But R:BASE is not just a Database Management System; it is a total GUI development environment for all Windows desktop and network applications. R:BASE for Windows is the ideal Database Management Suite for creating and maintaining your mission critical data with a true graphical user interface. Since its introduction in 1981 as the first PC-based database management system based on Dr. Codd's relational model, R:BASE has led as the first 32-bit DBMS in its class, providing programming-free application development, automatic multi-user capabilities, 4GL (a full-featured programming language in the R:BASE base product) and embedded ANSI SQL. And now with R:BASE for Windows, we have added a whole new look and feel to enhance the applications you develop in R:BASE. You can rapidly produce the type of results that previously would have required various third party development tools. Simply using native controls, you can now design cool applications at a fraction of the cost and development time when compared to other database and development tools available.

1.4 Oterro License Summary

Oterro is licensed in a variety of ways allowing for easy customization. You should refer to the license that shipped with your product for exact details on your licensing. In general, there are two types or categories of licensing. The first type allows for a certain number of connections to a database (this number includes all connections, whether made by Oterro or by other R:BASE products). If this count is reached, new connections to the database cannot be made by the limited version of Oterro. Your other unlimited products will not be affected. These numbered versions of Oterro are available in 1, 5, 10, and 25 user counts. These licenses are not cumulative. For example, purchasing a 1 user and a 5 user does not allow 6 users. The second type allows for unlimited connections and comes in three flavors: Site Unlimited (not distributable), Single Application Unlimited (distributable with a single application) and Multi Application Unlimited (which can be distributed with any number of applications).

1.5 Complimentary Support

30 DAY LIMITED COMPLIMENTARY TECHNICAL SUPPORT

A. LICENSEE RESPONSIBILITIES.

1. To help us expedite the process and provide high quality assistance, the licensee must provide proof of purchase. Proof of purchase is defined as the following: registration number, purchase date, version and build number, and company or individual to which product is registered.

2. To have operating system, workstations, and local network installed and functional. R:BASE Technologies will NOT be responsible for resolving issues not pertaining to the software product.

3. Our support staff deals with advanced issues, therefore the person contacting R:BASE Technologies for assistance should be the system administrator or have other R:BASE/SQL experience and be able to understand and implement the advice given.

4. To have the database, application, and command files being reviewed, safely backed-up before attempting assistance. R:BASE Technologies will NOT be held responsible for lost data or corruption as a result of advice given.

B. R:BASE TECHNOLOGIES, INC. RESPONSIBILITIES.

1. To provide quality assistance in a timely manner to aid in the installation of the product and elementary conversion of database, application, and command files within 30 days of the date of purchase.

2. To provide a reasonable solution for any solvable issue. Not all issues may be solved, and therefore we will acknowledge the existence of known issues, or bugs, which we are presently aware of, that have no reasonable work-around.

R:BASE Technologies reserves the right to limit the amount of support time allotted to a maximum of 2 HOURS during the 30-Day Complimentary Technical Support period. We also reserve the right to limit the quantity of calls from a particular licensee to 30 MINUTES in a single day. Issues are dealt with on a case-by-case basis, and are handled at the discretion of the support agent assigned to the case. Complimentary Support is limited to INSTALLATION and ELEMENTARY CONVERSION related issues ONLY. Our support hours are Monday through Friday, from 10:00 AM to 6:00 PM (EST).

For application, design, or advanced conversion assistance, R:BASE Technologies offers Technical Support Plans of various types to meet your needs. Please visit the Support page at https://www.rbase.com/support for details and pricing.

Part II

2 Programming

The Oterro Engine allows developers programming in ODBC compliant products to manipulate databases and their associated tables and views with full support for security, data integrity, and relational flexibility. For database applications using a high-level programming language, the Oterro Engine effectively meets the demand for speed and flexibility.

The Oterro Engine provides functions for loading and manipulating databases in both single- and multi-user environments. These functions provide Core, Level 1, and Level 2 ODBC conformance for the developer.

2.1 System Requirements

The following system specifications are recommended for the optimal use of R:BASE and R:BASE-related software.

Workstation Hardware

- 2-Core 2GHz+ CPU
- 2 GB of available RAM (4 GB recommended)
- 2 GB of available hard disk space
- 1024x768 or higher resolution video adapter and display
- Standard mouse or compatible pointing device
- Standard keyboard

Server Hardware

- 2-Core 2GHz+ CPU
- 6 GB of available RAM (8 GB recommended)

Operating System

- Microsoft Windows 11 (Professional)
- Microsoft Windows 10 (Professional)
- Microsoft Windows Server 2025
- Microsoft Windows Server 2022
- Microsoft Windows Server 2019
- Microsoft Windows Server 2016

Network

- Ethernet infrastructure (Gigabyte recommended)
- Internet connection recommended, but not required, for license activation, software updates, and support
- Anti-virus programs should exclude the R:BASE program, and any add-on product, executable and database files

2.2 Windows Programming

The following discussion assumes a working knowledge of Windows software development.

The Oterro Engine is supplied in a Windows Dynamic Link Library (DLL) format. The DLL can be used by Windows developers who use any development environment supporting access to DLLs.

2.3 OTERRO11.CFG File

Make sure an OTERRO11.CFG file is available with the appropriate settings when using the Oterro Engine. Use the OTERRO11.CFG file to set required database environment settings, such as MULTI-USER. An OTERRO11.CFG file is included with the Oterro installation and should be located in either of the below folder locations, based upon how the product installer was executed at the User Info screen (Install App for: Anyone/Only me).

```
C:\Users\Public\RBTI
C:\Users\<UserName>\RBTI
```

If no OTERRO11.CFG file is present, the Oterro Engine creates a new one. If you are working in the Visual Basic development environment, the OTERRO11.CFG is created where you started Visual Basic. If you are running a compiled application, the OTERRO11.CFG is created in the directory with the compiled application.

The OTERRO11.CFG file sets up the default database environment. Settings can be changed by calling the SQLSetConnectOption function or by editing the OTERRO11.CFG file.

2.4 Open Database Connectivity

Open Database Connectivity (ODBC), which is included in the Oterro Engine, is Microsoft's initiative for a standard SQL interface to database products. The Microsoft ODBC specification includes three levels of conformance: Core, Level 1, and Level 2 and the Oterro Engine supports all of these. Because Microsoft's ODBC specification is the basis for the Oterro Engine, the Oterro Engine can be used with the Microsoft ODBC Driver Manager and the ODBC SDK. The ODBC interface allows Windows applications to access database engines, such as the Oterro Engine, using SQL as a standard language. Also, ODBC allows the developer to write one application that has the potential to access data from databases created with different ODBC-compliant products. For example, you could write an application that would select data from an Oterro database as well as SQL Server or Oracle databases, separately or all at the same time, depending on which engines (drivers) were installed.

The ODBC driver manager (32-bit or 64-bit) must be configured for use with the Oterro Engine (the ODBC server). The function calls in the Oterro Engine are exactly the same as those in the ODBC Driver Manager, so no additional code needs to be written.

Part III

3 How to Use the Oterro Engine

The following provides guidelines for developing applications using the Oterro Engine.

Since there are numerous versions of Visual Basic, the following discussions are brief; such as only the data control property settings that are required to make a connection to an Oterro database are covered here.

3.1 Data Access Objects (DAO)

You can access and manipulate data in an Oterro database table by using the Microsoft data control (DAO) with a remote ODBC database. The following describes the required property settings when using data controls with the Oterro Engine.

DAO Control Properties

Set data control properties as follows:

Property	Setting
Connect	Specifies the Oterro database name and must at least contain ODBC. You must type in ODBC; and the connection string. ODBC is not available from the drop down list. The Oterro database must be registered in the ODBC administrator. Enter the entire connection string in the following format: **ODBC;DSN=C:\RBTI\OTERRO11\samples\bluzvan\bluzvan;UID=none;PWD=none;** Elements that are not included in the connection string will be prompted for.
DatabaseName	Leave the database name blank.
RecordSource	A table name.

The above data control settings retrieve all rows from a table. To select specific rows, you must set the Options properties setting to 64 and enter a SELECT command including WHERE clause as the RecordSource property. The result set that is generated cannot be updated.

3.2 Configuring the Database Environment

According to the ODBC specifications, all ODBC-compliant drivers (such as the Oterro Engine), by default, connect databases with transaction processing and AUTOCOMMIT on.

Although transaction processing works well when it is on, problems can occur, such as needing to recover databases that have had failed transactions. Because problems can occur, it might be prudent to turn transaction processing off when developing an application. You can turn transaction processing off by calling the function SQLSetConnectOption with the appropriate arguments, or by setting TRANSACT off in the OTERRO11.CFG file. The OTERRO11.CFG file is installed with the product in the C:\RBTI\Oterro11 directory or is created when the database is connected if a CFG file is not found.

If the Oterro Engine SQLSetConnectOption is set on or AUTORECOVER is on in the OTERRO11.CFG file, databases with failed transactions are recovered automatically when you connect to the database. The default setting for AUTORECOVER is ON.

Database Modes

Setting	OTERRO11.CFG	Options
ACCESS MODE	N/A	READ ONLY or READ WRITE
AUTO COMMIT	AUTOCOMM	ON or OFF
AUTO CONVERT	AUTOCONV	ON or OFF
AUTO RECOVER	AUTORECO	ON or OFF
AUTO ROWVER	AUTOROWV	ON or OFF
AUTO SYNC	AUTOSYNC	ON or OFF
AUTO UPGRADE	AUTOUPGR	ON or OFF

FASTLOCKS	FASTLOCK	ON or OFF
MAX # of TRANSACTIONS	MAXTRANS	1 to 255
MULTI-USER MODE	MULTI	ON or OFF
STATICDB	STATICDB	ON or OFF
TRANSACTION MODE	TRANSACT	ON or OFF

The tables below list the database commands available through SQLExecDirect as well as available SET commands that can also be sent through SQLExecDirect or that can be set in the OTERRO11.CFG file. Settings can be retrieved with the SQLGetConnectOption function after allocating a connection handle.

3.3 Available Commands

ALTER TABLE	DROP
ATTACH*	GRANT
AUTOCHK*	INSERT
AUTONUM*	PACK*
COMMENT ON	RELOAD*
CREATE INDEX	RENAME*
CREATE SCHEMA	REVOKE
CREATE TABLE	RULES*
CREATE VIEW	SELECT
DELETE	SET* (see below)
DELETE DUPLICATES*	UNLOAD*
DETACH*	UPDATE

***** Denotes Oterro database-specific commands that are non-SQL, but are helpful in Oterro databases.

Available SET Commands

Special Characters

SET Command	Default
BLANK	(space)
DELIMIT	, (comma)
MANY	%
NULL	-0-
QUOTES	' (single quote)
SINGLE	_
IDQUOTES	` (reverse quote)

Operating Conditions

SET Command	Default
AUTOCOMMIT	ON
MANOPT	OFF
AUTOCONVERT	ON
MIRROR	OFF
AUTORECOVER	ON
NAME	USER**************
AUTOROWVER	ON
NOTE_PAD	10
AUTOSYNC	ON
NULL	-0-
AUTOUPGRADE	ON
QUALCOLS	10
CASE	OFF
RULES	ON
CLEAR	ON
SCRATCH	ON

CURRENCY	$ PREFIX 2 B
SORT	OFF
DATE CENTURY	19
TIME FORMAT	HH:MM:SS
DATE FORMAT	YYYY-MM-DD
TIME SEQUENCE	HHMMSS
DATE SEQUENCE	YYYYMMDD
TOLERANCE	0
DATE YEAR	0
USER	PUBLIC
FASTFK	OFF
WAIT	4
INTERVAL	5

3.4 Handling Data of Variable Lengths

All functions that point to data of a variable length have an associated length argument containing column names and resulting parameter values for textual data types. These length arguments are useful for programming languages that do not use null-terminated strings and for passing binary data.

The length parameter has the following semantics when used on input (the application is passing data to the Oterro Engine), and output (the application is receiving data from the Oterro Engine).

Length Parameter for Input:

length >= 0
This indicates the actual length.

Length Parameter for Output:

An application always allocates memory for output buffers and must tell the Oterro Engine the length of the output buffer; conversely, the Oterro Engine must tell the application how much data was actually placed in the buffer, and have a way to pass indicator data—such as truncation, or a NULL (database) value for a particular column in a particular row—to the application:

Length=SQL_NULL_DATA (-1)
This tells the application that null data was passed.

Length=SQL_SUCCESS_WITH_INFO (1)
This tells the application that the output buffer was smaller than the target data, resulting in an error code.

3.5 Terminating Transactions and Disconnecting

When database access is complete, the statement, connection, and environment handles must be released using SQLFreeStmt, SQLDisconnect, SQLFreeConnect, and SQLFreeEnv. SQLDisconnect frees and drops any allocated statement handles as if SQLFreeStmt with SQL_DROP was called.

Before releasing the connection handle, the database must be disconnected. When pending transactions exist on the connection, the disconnect is rejected; therefore, the application must ensure that all transactions are committed or rolled back before disconnecting. Use SQLTransact to commit or rollback the current transaction. If this cannot be done, a disconnect can be attempted; however, look for the error SQLSTATE 25000 to determine if a commit or rollback must be performed. For an example of terminating and disconnecting, see the code examples for SQLDisconnect.

NOTE: When using Visual Basic DAO data controls, the database is disconnected only on exit of the application. When using Visual Basic RemoteDataControls, code can be written in the application to disconnect the database. Be sure to exit the application using the SQLFreeStmt, SQLDisconnect, SQLFreeConnect, or SQLFreeEnv commands as otherwise, temporary files with the .$$$ extension will not be deleted.

3.6 Error Checking

Error information is available from the environment, connection, and statement handles. In general, when a function deals with a handle other than allocating or freeing it, error information is stored in the handle used.

When an Oterro Engine function fails, it returns SQL_ERROR, or if the error is non-fatal, it returns SQL_SUCCESS_WITH_INFO. To get additional information about these errors (if available), call SQLError with the appropriate handles as arguments. The information provided consists of the error-code number for whatever data source was used, the SQLSTATE and the database error-message text. (SQLSTATE is a five character string with a null termination character.) For more information, see SQLError.

Error codes

Return Value	**Value**	**Description**
SQL_SUCCESS	0	The function completed successfully; no additional information is available.
SQL_SUCCESS_WITH_INFO	1	The function completed successfully; warning or additional information is available by calling SQLError.This return value usually indicates value truncation.
SQL_NO_DATA_FOUND	100	A return code from SQLFetch or SQLError indicates that all rows from a result set have been fetched.
SQL_ERROR	-1	The function failed—check SQLError for more specific failure information.
SQL_INVALID_HANDLE	-2	The function failed due to a null or invalid connection or statement handle, indicating a programming error. No additional information is available.

Use SQLError to perform error checking where appropriate. For an example of establishing connections, see the code examples for SQLAllocConnect.

When SQL_INVALID_HANDLE or SQL_NO_DATA_FOUND is returned, no further information is available from the Oterro Engine.

After a connection to a database is made, the main section of the application is run. Running the application involves calling several functions to execute SQL statements and retrieve results. One of the primary goals of an ODBC/Oterro Engine application is to handle recoverable errors, such as truncation and syntax errors, and to report fatal or severe errors. Handling recoverable errors and reporting fatal errors is accomplished by making calls to SQLError whenever an error is returned.

The statement handle is the structure associated with all SQL statements and other data access and manipulation functions. A statement handle is not a thread of statement execution; it represents a single SQL statement. A statement handle can be used repeatedly by using SQLFreeStmt. Before reusing the statement handle, you must close any cursor that might have been opened on it by a statement returning a result set. Closing a cursor is done by calling SQLFreeStmt with the option SQL_CLOSE. SQLFreeStmt can also free bound columns, free parameters on SQL statements, and totally free the statement handle.

After executing an SQL statement, use the SQLGetData function to retrieve the results from the table queried. Presentation of these results is under the control of the developer. Several other functions, such as SQLNumResultCols and SQLColAttributes, are useful for determining display information, such as column number and size.

When data is retrieved from a result set, SQLFetch is called first. When SQLFetch successfully returns, it may be followed by a call to SQLGetData to get the actual data for a column.

For an example of executing SQL statements and retrieving results, see the code examples for SQLExecDirect or SQLTables.

NOTE: When using DAO or RDC, these functions are automatically handled by the data control and bound objects. However, using RDC, you can access the statement, connection, and environment handles generated by the data control: RDCname.environment.henv, RDCname.connection.hdbc, RDCname.resultset.hstmt.

3.7 Oterro Debug Setting

For Oterro debugging, you can add the OTDEBUG setting in the Oterro configuration file, which creates a log file to help understand a possible issue.

The following examples provide the supported use of the OTDEBUG setting in the configuration file:

a) The OTDEBUG setting does not appear in the CFG file, which means that no debug log file is created.

b) OTDEBUG OFF (result is the same as a)

c) OTDEBUG ON (Debug log file is created within C:\oterro.log)

d) OTDEBUG ON D:\OT_TEST\MyTest.log (Debug log file MyTest.log is created within D:\OT_TEST)

Important: The debug setting and logging adds overhead to the Oterro engine and performance will decrease. After logging has been captured for a desired event where an issue occurs, the debug setting should be set to OFF in the configuration file.

3.8 Retrieving Status and Error Information

After a connection to a database has been made, error handling is critical. The most efficient way to check errors is to create a function to dump all errors for the environment, connection, and statement handles. For the Visual Basic examples included in this product, error checking is done automatically by defining a function that always calls SQLError when an error occurs. The error-checking function defined in the example is *ErrorCheck*.

As soon as you retrieve an error from the Oterro Engine, the error message is displayed even when it is truncated, and it cannot be displayed again. To ensure accommodation of the entire error message, specify a buffer of approximately 512 characters.

When truncation occurs in functions other than SQLGetData, the length of the output data is greater than or equal to the length of the given buffer. The actual length of the textual output-data is the length of the buffer minus one byte for the null terminator character. The required length of the buffer is the length of the output data (supplied by the Engine) plus one byte for the null termination character.

ODBC allows multiple error codes on one handle. To ensure cross-platform compatibility, continue reading errors until SQLError returns SQL_NO_DATA_FOUND. In the future, the Oterro Engine might return more than one error code; therefore, your code is prepared for upward compatibility.

For code examples of retrieving status and error information, see SQLError.

NOTE: Not applicable when using DAO as you don't have access to the handles. However, using RDC, you can access the statement, connection, and environment handles generated by the data control: RDCname.environment.henv, RDCname.connection.hdbc, RDCname.resultset.hstmt.

3.9 General Programming Tips

Designing your Application—Programming languages, such as Visual Basic, provide simple ways to use Windows controls. Design and program the user interface before trying to make the Oterro Engine function calls. After you have written and tested the user interface, start plugging in the function calls to the database.

Reusing Code—So that you don't duplicate your efforts, you can use one set of code for both buttons and menu commands. For example, Windows applications commonly use toolbars with tools that perform the same actions as menu commands. For a File Open button, which is the same as the File: Open menu command, you don't have to duplicate the code to connect to the database. Write the code to connect the File: Open menu command in a procedure called FILEOPEN. Then, for the button, simply call the existing FILEOPEN function.

Debugging—Several methods for debugging your application can be found in program languages and software development kits. Regardless of the method used, the Oterro Engine DLL cannot be directly debugged. However, the ODBC SDK provides tools that capture the actual commands being returned from the Oterro Engine. For more information about this method, refer to the respective manual(s).

Recovering Databases—Running applications with transaction processing set to on has the potential of leaving the database in an unusable state when transactions have been corrupted. When transactions are corrupted, connections to the database will fail. In order to restore the database to a working state, you need to be sure the AUTORECO is set to on in the SQLSetConnectOption or in the OTERRO11.CFG file.

Part IV

4 Oterro Engine Functions

This chapter includes the description, syntax, arguments, return values, related functions, errors, and code examples in Visual Basic for the Oterro Engine functions.

The table below groups the functions by categories of functionality: Driver Manager Specific Functions, Establishing and Freeing Connections, Executing SQL Statements, Retrieving Data and Values, Controlling Transactions, Handling Data and Values, Accessing the Data Dictionary, and Visual Basic Non-Supported Functions.

SQL Functions grouped by category

Category	**Function**
Driver Manager Specific Functions	SQLDrivers SQLDataSources
Establishing and Freeing Connections *(Although these functions are not supported in direct calls to the ODBC API through Visual Basic, they are used in the underlying code for many of the Data Controls and Remote Data Objects.)*	SQLAllocConnect SQLAllocEnv SQLAllocHandle SQLBrowseConnect SQLConnect SQLDisconnect SQLDriverConnect SQLFreeConnect SQLFreeEnv SQLGetConnectAttr SQLGetConnectOption SQLGetDiagRec SQLGetStmtAttr SQLSetConnectAttr SQLSetConnectOption SQLSetEnvAttr SQLSetStmtAttr
Executing SQL Statements	SQLAllocStmt SQLBulkOperations SQLCloseCursor SQLExecDirect SQLExecute SQLExtendedFetch SQLFetch SQLFetchScroll SQLFreeHandle SQLFreeStmt SQLGetCursorName SQLNativeSql SQLPrepare SQLSetCursorName
Retrieving Data and Values	SQLColAttributes SQLDescribeCol SQLGetData SQLNumResultCols SQLRowCount
Controlling Transactions	SQLCancel SQLEndTran SQLTransact
Handling Data and Values	SQLError SQLGetFunctions SQLGetInfo SQLGetStmtOption SQLSetScrollOptions SQLSetStmtOption
Accessing the Data Dictionary	SQLColumnPrivileges SQLColumns

	SQLForeignKeys SQLGetTypeInfo SQLMoreResults SQLPrimaryKeys SQLProcedureColumns SQLProcedures SQLSetPos SQLSpecialColumns SQLStatistics SQLTablePrivileges SQLTables
Functions not supported in Visual Basic *(Although these functions are not supported in direct calls to the ODBC API through Visual Basic, they are used in the underlying code for many of the Data Controls, and RemoteDataObjects.)*	SQLBindCol SQLBindParameter SQLDescribeParam SQLNumParams SQLParamOptions SQLPutData

4.1 SQLAllocConnect

SQLAllocConnect allocates the connection handle required for connecting to any database with the Oterro Engine and associates it with the environment handle specified by henv.

Syntax

```
RETCODE = SQLAllocConnect (henv, hdbc)
```

Arguments

Type	Argument	Use	Description
Long	henv	Input	The environment handle
Long	hdbc	Output	A pointer to storage for the connection handle

Return Values

```
SQL_SUCCESS, SQL_ERROR, or SQL_INVALID_HANDLE
```

Comments

When SQLAllocConnect returns SQL_ERROR, then the value to which hdbc points is set to zero. You can call the function SQLError for more information concerning the error. When you call SQLError for more information, SQL_NULL_HSTMT and SQL_NULL_HDBC must be passed for the hstmt and hdbc arguments in the SQLError function.

A valid environment handle must be established before allocating a connection. See SQLAllocEnv.

One connection handle is required for each connection to any database. Multiple connections to a single database are available when the database is on a network server and the Oterro database restrictions on multi-user access are satisfied (see the Multi-User Mode entry in Chapter 5), such as enabling the multi-user mode using the SQLSetConnectOption. Each user connecting to the same database must have the same database environment mode. This is achieved by setting MULTI, STATICDB, and TRANSACT the same for each user. Use the SQLSetConnectOption function or the OTERRO11.CFG file to set the mode.

CAUTION: If you specify a pointer to a connection handle that has already been allocated, SQLAllocConnect overwrites the pointer regardless of its contents.

Related Functions

Function	Description
SQLConnect	Opens a connection to a database.
SQLBrowseConnect	Retrieves values required to connect to a data source.
SQLDisconnect	Closes the connection to a database.
SQLDriverConnect	Prompts for information to open a connection to a database.
SQLFreeConnect	Frees the connection handle.
SQLGetConnectOption	Queries the status of a connection option.
SQLSetConnectOption	Sets a database connection option.

Errors

SQLSTATE	Description
S1000	An error has occurred that has no defined SQLSTATE—see the error message text.
S1001	A memory allocation failure.
S1009	An invalid argument value—a null pointer was passed.

Visual Basic Example

SQLAPI.BAS:
Declare Function SQLAllocConnect Lib "ODBC32.DLL" (ByVal henv&, hdbc&) As Integer

CODE:
```
Global szConnectOut As String * 512
Global cbConnectOut As Integer
Global dbstr As String
Global dbdir As String

Private Sub fdrvconn_Click()
     Dim i As Integer
     retcode = SQLAllocEnv(henv&)
     retcode = SQLAllocConnect(henv&, hdbc&)
          errorcheck retcode
     retcode = SQLDriverConnect(hdbc&, hwnd&, dbstr, SQL_NTS, szConnectOut,
     255, cbConnectOut, SQL_DRIVER_COMPLETE)
     If retcode <> 0 Then
          errorcheck retcode
          GoTo lend
     End If
     'get the database path
     retcode = SQLGetInfo(hdbc&, SQL_DATABASE_NAME, szConnectOut, 512,
     cbConnectOut)
          errorcheck retcode
     dbstr = Chop(szConnectOut)
     i = InStr(dbstr, "\") dbdir = Left$(dbstr, i)
     Do While i <> 0
          i = InStr(i + 1, dbstr, "\")
          If i <> 0 Then
               dbdir = Left$(dbstr, i)
          End If
     Loop retcode = SQLAllocStmt(hdbc&, hstmt&)
          errorcheck retcode
lend:
```

```
End Sub
```

4.2 SQLAllocEnv

SQLAllocEnv allocates an environment handle used for maintaining information on the current connections to the Oterro Engine.

Syntax

```
RETCODE = SQLAllocEnv (henv)
```

Arguments

Type	Argument	Use	Description
Long	henv	Output	A pointer to storage for the environment handle.

Return Values

```
SQL_SUCCESS, or SQL_ERROR
```

Comments

If SQLAllocEnv returns SQL_ERROR, the function sets henv to zero.

Only one environment handle should be allocated and used at a time.

The environment handle must be created before allocating a connection handle.

Caution: If you specify a pointer to an environment handle that has already been allocated, SQLAllocEnv overwrites the pointer regardless of its previous contents.

Related Functions

Function	Description
SQLFreeEnv	Frees the environment handle.

Errors

Because there is no allocated handle to pass information, no SQLSTATE can be returned.

Visual Basic Example

SQLAPI.BAS:
Declare Function SQLAllocEnv Lib "ODBC32.DLL" (phenv&) As Integer

CODE:

```
Global szConnectOut As String * 512
Global cbConnectOut As Integer
Global dbstr As String
Global dbdir As String


Private Sub fdrvconn_Click()
     Dim i As Integer
     retcode = SQLAllocEnv(henv&)
     retcode = SQLAllocConnect(henv&, hdbc&)
          errorcheck retcode
     retcode = SQLDriverConnect(hdbc&, hwnd&, dbstr, SQL_NTS, szConnectOut,
255, cbConnectOut, SQL_DRIVER_COMPLETE)
     If retcode <> 0 Then
          errorcheck retcode
          GoTo lend
     End If
     'get the database path
     retcode = SQLGetInfo(hdbc&, SQL_DATABASE_NAME, szConnectOut, 512,
cbConnectOut)
          errorcheck retcode
     dbstr = Chop(szConnectOut)
     i = InStr(dbstr, "\")
     dbdir = Left$(dbstr, i)
     Do While i <> 0
          i = InStr(i + 1, dbstr, "\")
          If i <> 0 Then
               dbdir = Left$(dbstr, i)
          End If
     Loop
     retcode = SQLAllocStmt(hdbc&, hstmt&)
          errorcheck retcode
lend:
End Sub
```

4.3 SQLAllocHandle

SQLAllocHandle allocates an environment, connection, statement, or descriptor handle.

This function, added in Oterro 3.0, is a generic function for allocating handles that replaces the ODBC 2.0 functions SQLAllocConnect, SQLAllocEnv, and SQLAllocStmt. To allow applications calling SQLAllocHandle to work with ODBC 2.x drivers, a call to SQLAllocHandle is mapped in the Driver Manager to SQLAllocConnect, SQLAllocEnv, or SQLAllocStmt, as appropriate.

Syntax

```
RETCODE = SQLAllocHandle( HandleType, InputHandle, OutputHandlePtr)
```

Arguments

Type	Argument	Use	Description
Integer	HandleType	Input	The type of handle to be allocated by SQLAllocHandle. Must be one of the following values: SQL_HANDLE_ENV SQL_HANDLE_DBC SQL_HANDLE_DESC SQL_HANDLE_STMT
Long	InputHandle	Input	The input handle in whose context the new handle is to be allocated. If HandleType is SQL_HANDLE_ENV, this is SQL_NULL_HANDLE. If HandleType is SQL_HANDLE_DBC, this must be an environment handle, and if it is SQL_HANDLE_STMT or SQL_HANDLE_DESC, it must be a connection handle.

Long	OutputHandlePtr	Output	Pointer to a buffer in which to return the handle to the newly allocated data structure.

Return Values

`SQL_SUCCESS, SQL_SUCCESS_WITH_INFO, SQL_INVALID_HANDLE,` or `SQL_ERROR`

When allocating a handle other than an environment handle, if SQLAllocHandle returns SQL_ERROR, it sets OutputHandlePtr to SQL_NULL_HDBC, SQL_NULL_HSTMT, or SQL_NULL_HDESC, depending on the value of HandleType, unless the output argument is a null pointer. The application can then obtain additional information from the diagnostic data structure associated with the handle in the InputHandle argument.

Environment Handle Allocation Errors

Environment allocation occurs both within the Driver Manager and within each driver. The error returned by SQLAllocHandle with a HandleType of SQL_HANDLE_ENV depends on the level in which the error occurred.

If the Driver Manager cannot allocate memory for *OutputHandlePtr when SQLAllocHandle with a HandleType of SQL_HANDLE_ENV is called, or the application provides a null pointer for OutputHandlePtr, SQLAllocHandle returns SQL_ERROR. The Driver Manager sets *OutputHandlePtr to SQL_NULL_HENV (unless the application provided a null pointer, which returns SQL_ERROR). There is no handle with which to associate additional diagnostic information. (If the driver has additional diagnostic information, it will put the information on a skeletal handle that it allocates; the Driver Manager will read the information from the diagnostic structure associated with this handle.)

The Driver Manager does not call the driver-level environment handle allocation function until the application calls SQLConnect, SQLBrowseConnect, or SQLDriverConnect. If an error occurs in the driver-level SQLAllocHandle function, then the Driver Manager–level SQLConnect, SQLBrowseConnect, or SQLDriverConnect function returns SQL_ERROR. The diagnostic data structure contains SQLSTATE IM004 (Driver's SQLAllocHandle failed), followed by a driver-specific SQLSTATE value from the driver. For example, SQLSTATE HY001 (Memory allocation error) indicates that the Driver Manager's call to the driver-level SQLAllocHandle returned SQL_ERROR. The error is returned on a connection handle.

For more information about the flow of function calls between the Driver Manager and a driver, see the SQLConnect Function.

Errors

The following table lists the SQLSTATE values typically returned by SQLAllocHandle and explains each one in the context of this function; the notation "(DM)" precedes the descriptions of SQLSTATEs returned by the Driver Manager. The return code associated with each SQLSTATE value is SQL_ERROR, unless noted otherwise.

SQLSTATE	Error	Description
01000	General warning	Driver-specific informational message. (Function returns SQL_SUCCESS_WITH_INFO.)
08003	Connection does not exist	(DM) The HandleType argument was SQL_HANDLE_STMT or SQL_HANDLE_DESC, but the connection specified by the InputHandle argument was not open. The connection process must be completed successfully (and the connection must be open) for the driver to allocate a statement or descriptor handle.
HY000	General error	An error occurred for which there was no specific SQLSTATE and for which no implementation-specific SQLSTATE was defined. The error message returned by SQLGetDiagRec in the *MessageText buffer describes the error and its cause.

HY001	Memory allocation error	(DM) The Driver Manager was unable to allocate memory for the specified handle. The driver was unable to allocate memory for the specified handle.
HY009	Invalid use of null pointer	(DM) The OutputHandlePtr argument was a null pointer.
HY010	Function sequence error	(DM) The HandleType argument was SQL_HANDLE_DBC, and SQLSetEnvAttr has not been called to set the SQL_ODBC_VERSION environment attribute.
HY013	Memory management error	The HandleType argument was SQL_HANDLE_DBC, SQL_HANDLE_STMT, or SQL_HANDLE_DESC; and the function call could not be processed because the underlying memory objects could not be accessed, possibly because of low memory conditions.
HY014	Limit on the number of handles exceeded	The driver-defined limit for the number of handles that can be allocated for the type of handle indicated by the HandleType argument has been reached.
HY092	Invalid attribute/option identifier	(DM) The HandleType argument was not: SQL_HANDLE_ENV, SQL_HANDLE_DBC, SQL_HANDLE_STMT, or SQL_HANDLE_DESC.
HYC00	Optional feature not implemented	The HandleType argument was SQL_HANDLE_DESC and the driver was an ODBC 2.x driver.
HYT01	Connection timeout expired	The connection timeout period expired before the data source responded to the request. The connection timeout period is set through SQLSetConnectAttr, SQL_ATTR_CONNECTION_TIMEOUT.
IM001	Driver does not support this function	(DM) The HandleType argument was SQL_HANDLE_STMT, and the driver was not a valid ODBC driver.(DM) The HandleType argument was SQL_HANDLE_DESC, and the driver does not support allocating a descriptor handle.

Comments

SQLAllocHandle is used to allocate handles for environments, connections, statements, and descriptors, as described in the following sections. For general information about handles, see Handles.

More than one environment, connection, or statement handle can be allocated by an application at a time if multiple allocations are supported by the driver. In ODBC, no limit is defined on the number of environment, connection, statement, or descriptor handles that can be allocated at any one time. Drivers may impose a limit on the number of a certain type of handle that can be allocated at a time; for more information, see the driver documentation.

If the application calls SQLAllocHandle with *OutputHandlePtr set to an environment, connection, statement, or descriptor handle that already exists, the driver overwrites the information associated with the handle, unless the application is using connection pooling (see "Allocating an Environment Attribute for Connection Pooling" later in this section). The Driver Manager does not check to see whether the handle entered in *OutputHandlePtr is already being used, nor does it check the previous contents of a handle before overwriting them.

Note: It is incorrect ODBC application programming to call SQLAllocHandle two times with the same application variable defined for *OutputHandlePtr without calling SQLFreeHandle to free the handle before reallocating it. Overwriting ODBC handles in such a manner could lead to inconsistent behavior or errors on the part of ODBC drivers.

On operating systems that support multiple threads, applications can use the same environment, connection, statement, or descriptor handle on different threads. Drivers must therefore support safe, multithread access to this information; one way to achieve this, for example, is by using a critical section or a semaphore. For more information about threading, see Multithreading.

Allocating an Environment Handle

An environment handle provides access to global information such as valid connection handles and active connection handles. For general information about environment handles, see Environment Handles.

To request an environment handle, an application calls SQLAllocHandle with a HandleType of SQL_HANDLE_ENV and an InputHandle of SQL_NULL_HANDLE. The driver allocates memory for the environment information and passes the value of the associated handle back in the *OutputHandlePtr argument. The application passes the *OutputHandle value in all subsequent calls that require an environment handle argument.

Under a Driver Manager's environment handle, if there already exists a driver's environment handle, then SQLAllocHandle with a HandleType of SQL_HANDLE_ENV is not called in that driver when a connection is made, only SQLAllocHandle with a HandleType of SQL_HANDLE_DBC. If a driver's environment handle does not exist under the Driver Manager's environment handle, both SQLAllocHandle with a HandleType of SQL_HANDLE_ENV and SQLAllocHandle with a HandleType of SQL_HANDLE_DBC are called in the driver when the first connection handle of the environment is connected to the driver.

When the Driver Manager processes the SQLAllocHandle function with a HandleType of SQL_HANDLE_ENV, it checks the Trace keyword in the [ODBC] section of the system information. If it is set to 1, the Driver Manager enables tracing for the current application on a computer that is running Microsoft® Windows® 95/98, Microsoft Windows NT® Server/Windows 2000 Server, or Microsoft Windows NT Workstation/Windows 2000 Professional. If the trace flag is set, tracing starts when the first environment handle is allocated and ends when the last environment handle is freed.

After allocating an environment handle, an application must call SQLSetEnvAttr on the environment handle to set the SQL_ATTR_ODBC_VERSION environment attribute. If this attribute is not set before SQLAllocHandle is called to allocate a connection handle on the environment, the call to allocate the connection will return SQLSTATE HY010 (Function sequence error).

Allocating Shared Environments for Connection Pooling

Environments can be shared among multiple components on a single process. A shared environment can be used by more than one component at the same time. When a component uses a shared environment, it can use pooled connections, which allow it to allocate and use an existing connection without re-creating that connection.

Before allocating a shared environment that can be used for connection pooling, an application must call SQLSetEnvAttr to set the SQL_ATTR_CONNECTION_POOLING environment attribute to SQL_CP_ONE_PER_DRIVER or SQL_CP_ONE_PER_HENV. SQLSetEnvAttr in this case is called with EnvironmentHandle set to null, which makes the attribute a process-level attribute.

After connection pooling has been enabled, an application calls SQLAllocHandle with the HandleType argument set to SQL_HANDLE_ENV. The environment allocated by this call will be an implicit shared environment because connection pooling has been enabled. (For more information about connection pooling, see SQLConnect Function.)

When a shared environment is allocated, the environment that will be used is not determined until SQLAllocHandle with a HandleType of SQL_HANDLE_DBC is called. At that point, the Driver Manager tries to find an existing environment that matches the environment attributes requested by the application. If no such environment exists, one is created as a shared environment. The Driver Manager maintains a reference count for each shared environment; the count is set to 1 when the environment is first created. If a matching environment is found, the handle of that environment is returned to the application and the reference count is incremented. An environment handle allocated in this manner can be used in any ODBC function that accepts an environment handle as an input argument.

Allocating a Connection Handle

A connection handle provides access to information such as the valid statement and descriptor handles on the connection and whether a transaction is currently open. For general information about connection handles, see Connection Handles.

To request a connection handle, an application calls SQLAllocHandle with a HandleType of SQL_HANDLE_DBC. The InputHandle argument is set to the environment handle that was returned by the call to SQLAllocHandle that allocated that handle. The driver allocates memory for the connection information and passes the value of the associated handle back in *OutputHandlePtr. The application passes the *OutputHandlePtr value in all subsequent calls that require a connection handle.

The Driver Manager processes the SQLAllocHandle function and calls the driver's SQLAllocHandle function when the application calls SQLConnect, SQLBrowseConnect, or SQLDriverConnect. (For more information, see SQLConnect Function.)

If the SQL_ATTR_ODBC_VERSION environment attribute is not set before SQLAllocHandle is called to allocate a connection handle on the environment, the call to allocate the connection will return SQLSTATE HY010 (Function sequence error).

When an application calls SQLAllocHandle with the InputHandle argument set to SQL_HANDLE_DBC and also set to a shared environment handle, the Driver Manager tries to find an existing shared environment that matches the environment attributes set by the application. If no such environment exists, one is created, with a reference count (maintained by the Driver Manager) of 1. If a matching shared environment is found, that handle is returned to the application and its reference count is incremented.

The actual connection that will be used is not determined by the Driver Manager until SQLConnect or SQLDriverConnect is called. The Driver Manager uses the connection options in the call to SQLConnect (or the connection keywords in the call to SQLDriverConnect) and the connection attributes set after connection allocation to determine which connection in the pool should be used. For more information, see SQLConnect Function.

Allocating a Statement Handle

A statement handle provides access to statement information, such as error messages, the cursor name, and status information for SQL statement processing. For general information about statement handles, see Statement Handles.

To request a statement handle, an application connects to a data source and then calls SQLAllocHandle before it submits SQL statements. In this call, HandleType should be set to SQL_HANDLE_STMT and InputHandle should be set to the connection handle that was returned by the call to SQLAllocHandle that allocated that handle. The driver allocates memory for the statement information, associates the statement handle with the specified connection, and passes the value of the associated handle back in *OutputHandlePtr. The application passes the *OutputHandlePtr value in all subsequent calls that require a statement handle.

When the statement handle is allocated, the driver automatically allocates a set of four descriptors and assigns the handles for these descriptors to the SQL_ATTR_APP_ROW_DESC, SQL_ATTR_APP_PARAM_DESC, SQL_ATTR_IMP_ROW_DESC, and SQL_ATTR_IMP_PARAM_DESC statement attributes. These are referred to as implicitly allocated descriptors. To allocate an application descriptor explicitly, see the following section, "Allocating a Descriptor Handle."

Allocating a Descriptor Handle

When an application calls SQLAllocHandle with a HandleType of SQL_HANDLE_DESC, the driver allocates an application descriptor. These are referred to as explicitly allocated descriptors. The application directs a driver to use an explicitly allocated application descriptor instead of an automatically allocated one for a given statement handle by calling the SQLSetStmtAttr function with the SQL_ATTR_APP_ROW_DESC or SQL_ATTR_APP_PARAM_DESC attribute. An implementation descriptor cannot be allocated explicitly, nor can an implementation descriptor be specified in an SQLSetStmtAttr function call.

Explicitly allocated descriptors are associated with a connection handle instead of a statement handle (as automatically allocated descriptors are). Descriptors remain allocated only when an application is actually connected to the database. Because explicitly allocated descriptors are associated with a connection handle, an application can associate an explicitly allocated descriptor with more than one statement within a connection. An implicitly allocated application descriptor, on the other hand, cannot be associated with more than one statement handle. (It cannot be associated with any statement handle other than the one that it was allocated for.) Explicitly allocated descriptor handles can be freed explicitly either by the application or by calling SQLFreeHandle with a HandleType of SQL_HANDLE_DESC, or implicitly when the connection is closed.

When the explicitly allocated descriptor is freed, the implicitly allocated descriptor is again associated with the statement. (The SQL_ATTR_APP_ROW_DESC or SQL_ATTR_APP_PARAM_DESC attribute for that statement is again set to the implicitly allocated descriptor handle.) This is true for all statements that were associated with the explicitly allocated descriptor on the connection.

For more information about descriptors, see Descriptors.

Related Functions

Function	Description
SQLExecDirect	Executing an SQL statement
SQLExecute	Executing a prepared SQL statement
SQLFreeHandle	Freeing an environment, connection, statement, or descriptor handle
SQLPrepare	Preparing a statement for execution
SQLSetConnectAttr	Setting a connection attribute
SQLSetEnvAttr	Setting an environment attribute
SQLSetStmtAttr	Setting a statement attribute

Code Example

```
// SQLBrowseConnect_Function.cpp
// compile with: odbc32.lib
#include <windows.h>
#include <sqltypes.h>
#include <sqlext.h>

#define BRWS_LEN 100
SQLHENV henv;
SQLHDBC hdbc;
SQLHSTMT hstmt;
SQLRETURN retcode;
SQLCHAR szConnStrIn[BRWS_LEN], szConnStrOut[BRWS_LEN];
SQLSMALLINT cbConnStrOut;

void GetUserInput(SQLCHAR * szConnStrOut, SQLCHAR * szConnStrIn) {}

int main() {
   // Allocate the environment handle.
   retcode = SQLAllocHandle(SQL_HANDLE_ENV, SQL_NULL_HANDLE, &henv);
   if (retcode == SQL_SUCCESS || retcode == SQL_SUCCESS_WITH_INFO) {

      // Set the version environment attribute.
      retcode = SQLSetEnvAttr(henv, SQL_ATTR_ODBC_VERSION, (SQLPOINTER*)
SQL_OV_ODBC3, 0);
      if (retcode == SQL_SUCCESS || retcode == SQL_SUCCESS_WITH_INFO) {

         // Allocate the connection handle.
         retcode = SQLAllocHandle(SQL_HANDLE_DBC, henv, &hdbc);
         if (retcode == SQL_SUCCESS || retcode == SQL_SUCCESS_WITH_INFO) {
            // Call SQLBrowseConnect until it returns a value other than
SQL_NEED_DATA
            // (pass data source name the first time).  If SQL_NEED_DATA is
returned, call GetUserInput
            // (not shown) to build a dialog from the values in
szConnStrOut.  The user-supplied values
            // are returned in szConnStrIn, which is passed in the next
call to SQLBrowseConnect.
```

```
            strcpy_s((char*)szConnStrIn, _countof(szConnStrIn),
"DSN=Sales");
            do {
               retcode = SQLBrowseConnect(hdbc, szConnStrIn, SQL_NTS,
                  szConnStrOut, BRWS_LEN, &cbConnStrOut);
               if (retcode == SQL_NEED_DATA)
                  GetUserInput(szConnStrOut, szConnStrIn);
            } while (retcode == SQL_NEED_DATA);

            if (retcode == SQL_SUCCESS || retcode == SQL_SUCCESS_WITH_INFO)
{

               // Allocate the statement handle.
               retcode = SQLAllocHandle(SQL_HANDLE_STMT, hdbc, &hstmt);

               if (retcode == SQL_SUCCESS || retcode ==
SQL_SUCCESS_WITH_INFO)
                  // Process data after successful connection
                  SQLFreeHandle(SQL_HANDLE_STMT, hstmt);
               SQLDisconnect(hdbc);
            }
         }
         SQLFreeHandle(SQL_HANDLE_DBC, hdbc);
      }
   }
   SQLFreeHandle(SQL_HANDLE_ENV, henv);
}
```

4.4 SQLAllocStmt

SQLAllocStmt allocates a new statement handle and associates it with the connection handle specified by *hdbc*.

Syntax

RETCODE = SQLAllocStmt (hdbc, hstmt)

Arguments

Type	Argument	Use	Description
Long	hdbc	Input	The database connection handle.
Long	hstmt	Output	A pointer to storage for the statement handle.

Return Values

SQL_SUCCESS, SQL_INVALID_HANDLE, or SQL_ERROR

Comments

The hstmt is the pointer to a statement handle, which is an input parameter for all functions that process SQL commands. All information relating to descriptors, result values, and status information is associated with the statement handle by the Oterro Engine.

A valid connection handle must be established before allocating a statement handle. See SQLAllocConnect.

If SQL_ERROR is returned, the hstmt argument is set to zero. SQLError can be called with this hstmt (or SQL_NULL_HSTMT) and the hdbc passed to SQLAllocStmt for more information.

Caution: If you specify a pointer to a statement handle that has already been allocated, SQLAllocStmt overwrites the pointer regardless of its contents.

Related Functions

Function	Description
SQLExecDirect	Executes an SQL statement.
SQLExecute	Executes a prepared SQL statement.
SQLFreeStmt	Frees a statement handle.
SQLPrepare	Prepares an SQL statement for execution.

Errors

SQLSTATE	Description
08003	No database has been connected.
S1000	An error has occurred that has no defined SQLSTATE—see the error message text.
S1001	A memory allocation failure.
S1009	An invalid argument value—a null pointer was passed.

Visual Basic Example

SQLAPI.BAS:
Declare Function SQLAllocStmt Lib "ODBC32.DLL" (ByVal hdbc&, phstmt&) As Integer

CODE:

```
Global szConnectOut As String * 512
Global cbConnectOut As Integer
Global dbstr As String
Global dbdir As String


Private Sub fdrvconn_Click()
     Dim i As Integer
     retcode = SQLAllocEnv(henv&)
     retcode = SQLAllocConnect(henv&, hdbc&)
          errorcheck retcode
     retcode = SQLDriverConnect(hdbc&, hwnd&, dbstr, SQL_NTS, szConnectOut,
255, cbConnectOut, SQL_DRIVER_COMPLETE)
     If retcode <> 0 Then
          errorcheck retcode
          GoTo lend
     End If
     'get the database path
     retcode = SQLGetInfo(hdbc&, SQL_DATABASE_NAME, szConnectOut, 512,
cbConnectOut)
          errorcheck retcode
     dbstr = Chop(szConnectOut)
     i = InStr(dbstr, "\")
     dbdir = Left$(dbstr, i)
     Do While i <> 0
          i = InStr(i + 1, dbstr, "\")
          If i <> 0 Then
               dbdir = Left$(dbstr, i)
          End If
     Loop
     retcode = SQLAllocStmt(hdbc&, hstmt&)
          errorcheck retcode
lend:
End Sub
```

4.5 SQLBindCol

Note: This function cannot be called from Visual Basic because it uses a pointer to a data structure as an input argument but does not use that pointer immediately. Since Visual Basic moves data around, the pointers would become invalid. Use SQLGetData instead. The SQLBindCol function is included here with the syntax for using the C or C++ programming language.

SQLBindCol defines storage and conversions for a column in a result set.

Syntax

`RETCODE PASCAL` SQLBindCol (hstmt, icol, fCType, rgbValue, cbValueMax, pcbValue)

Arguments

Type	Argument	Use	Description
HSTMT	hstmt	Input	The statement handle.
UWORD	icol	Input	The column number in the result data, starting at 1.
SWORD	fCType	Input	The C data type the value in the buffer is converted to.
PTR	rgbValue	Input	A pointer to storage for the data.
SDWORD	cbValueMax	Input	The maximum length of the rgbValue buffer.
SDWORD FAR*	pcbValue	Output	The number of bytes placed in the rgbValue buffer.

Return Values

`SQL_SUCCESS, SQL_SUCCESS_WITH_INFO, SQL_ERROR, or SQL_INVALID_HANDLE`

Comments

The columns in the result set are numbered sequentially, starting at one. The number of columns in a result set is determined by calling SQLNumResultCols after executing the SQL statement that generates the result set. (The result set is generated by SQLExecDirect or SQLExecute.) SQLBindCol should (in general) be called before SQLFetch; however, you can call SQLBindCol after fetching one or more rows.

The argument fCType can be set to SQL_C_DEFAULT or any other legal C data type for the data-type conversion. When SQL_C_DEFAULT is specified, the data is placed in the rgbValue buffer using the appropriate C data type. When SQL_C_CHAR is specified for a non-character column type, data converts to a character string. For more information about conversion of data types, see "Data Types and Retrieval for C" in the Appendix.

The argument rgbValue is a pointer to storage of the converted data. The user's application is responsible for allocating enough storage for the converted data as specified by fCType. For variable length data, the application must allocate the maximum length of a column to ensure that all data is retrieved. When the application does not allocate the maximum length, the data could be truncated.

When this function is called, cbValueMax indicates the maximum number of bytes to be stored in rgbValue. After each SQLFetch, pcbValue indicates the actual number of bytes transferred to rgbValue, or SQL_NULL_DATA (-1) when data value is NULL. When the returned pcbValue is greater than or equal to cbValue and SQL_SUCCESS_WITH_INFO is the return code, the data is truncated. To retrieve all the desired data, you must enlarge your output buffer and re-execute the select statement, or use SQLGetData.

4.6 SQLBindParameter

Note: This function cannot be called from Visual Basic because it uses a pointer to a data structure as an input argument but does not use that pointer immediately. Since Visual Basic moves data around, the pointers would become invalid. It is included here with the syntax for using the C and C++ programming language.

SQLBindParameter defines storage for a parameter marker in an SQL statement.

Syntax

```
RETCODE PASCAL SQLBindParameter (hStmt, ipar, fParamType, fCType, fSqlType, cbColDef,
ibScale, rgbValue, cbValueMax, pcbValue)
```

Arguments

Type	Argument	Use	Description
HSTMT	hStmt	Input	Statement handle.
UWORD	ipar	Input	The parameter to set, starting at 1.
SWORD	fParamType	Input	The type of the parameter: SQL_PARAM_INPUT SQL_PARAM_OUTPUT SQL_PARAM_INPUT_OUTPUT
SWORD	fCType	Input	Convert to this C data type in the value buffer. See Appendix.
SWORD	fSqlType	Input	The ODBC SQL data type.
UDWORD	cbColDef	Input	The length/precision of the column.
SWORD	ibScale	Input	The scale of the column.
PTR	rgbValue	Input/ Output	A pointer to storage for the data.
SDWORD	cbValueMax	Input	Maximum length of the rgbValue buffer.
SDWORD FAR*	pcbValue	Input/ Output	The number of bytes placed in the rbgValue buffer.

Return Values

```
SQL_SUCCESS, SQL_SUCCESS_WITH_INFO, SQL_ERROR, or SQL_INVALID_HANDLE.
```

Comments

Bindings from SQLBindParameter remain in effect until the application calls SQLBindParameter again or until the application call SQLFreeStmt with the SQL_DROP or SQL_RESET_PARAMS option.

If pcbValue is a null pointer, the driver presumes that all input parameter values are not NULL and that character and binary data are null-terminated. If fParamType is SQL_PARAM_OUTPUT and rgbValue and pcbValue are both null pointers, the driver discards the output value.

4.7 SQLBrowseConnect

QLBrowseConnect is called repeatedly to find and enumerate the attributes and values needed to connect to a data source.

Syntax

```
RETCODE = SQLBrowseConnect (hdbc, szConnStrIn, cbConnStrIn, szConnStrOut,
cbConnStrOutMax, pcbConnStrOut)
```

Arguments

Type	Argument	Use	Description
Long	hdbc	Input	The database connection handle.
String	szConnStrIn	Input	The connection string.
Integer	cbConnStrIn	Input	The length of the connection string.
String	szConnStrOut	Output	A pointer to the filled connection string.
Integer	cbConnStrOutMax	Input	The maximum length of the output connection string.
Integer	pcbConnStrOut	Output	The length of the returned connection string.

Return Values

```
SQL_SUCCESS, SQL_SUCCESS_WITH_INFO, SQL_NEED_DATA, SQL_ERROR, or
SQL_INVALID_HANDLE.
```

Comments

When SQLBrowseConnect is called for the first time on a connection handle (hdbc), the connection string must contain the DSN keyword or the Driver keyword.

On each call to SQLBrowseConnect, the application itemizes the connection attribute values in the connection string. The driver returns successive levels of attributes and attribute values in the filled connection string; it returns SQL_NEED_DATA as long as there are connection attributes that have not yet been specified in the connection string.

When all levels of connection and their related attributes have been specified, the driver returns SQL_SUCCESS, the connection to the data source is complete, and a complete connection string is returned to the application. The connection string is suitable to use along with SQLDriverConnect with the SQL_DRIVER_NO_PROMPT option to establish another connection.

Related Functions

Function	Description
SQLAllocConnect	Allocates a connection handle.
SQLConnect	Opens a connection to a database.
SQLDataSources	Returns the data source names.
SQLDisconnect	Closes the connection to a database.
SQLDriverConnect	Prompts for information to open a connection to a database.
SQLDrivers	Returns driver descriptions and attributes.
SQLFreeConnect	Frees the connection handle.

Errors

SQLSTATE	Description
01000	Driver-specific informational message. (The function returns SQL_SUCCESS_WITH_INFO.)

01004	The data was truncated.
01S00	An invalid connection attribute was specified. (SQL_SUCCESS_WITH_INFO was returned.)
08001	Unable to connect to data source.
08002	The connection is already in use.
08004	The data source rejected the connection.
08S01	The data source connection failed before the function completed processing.
28000	No access is available for this user.
IM001	The driver associated with the hstmt does not support the function.
IM002	The data source was not found and a default driver was not specified.
IM003	The specified driver could not be loaded.
IM004	The driver's SQLAllocEnv failed.
IM005	The driver's SQLAllocConnect failed.
IM006	The driver's SQLSetConnectOption failed.
IM009	The driver was unable to load the specified translation DLL.
IM010	The data source name was too long.
IM011	The driver name was too long.
IM012	DRIVER keyword syntax error.
S1000	An error has occurred that has no defined SQLSTATE—see the error message text.
S1001	A memory allocation failure.
S1090	An invalid string or buffer length.
S1T00	The timeout period expired before the connection was completed.

Visual Basic Example

SQLAPI.BAS:
Declare Function SQLBrowseConnect Lib "ODBC32.DLL" (ByVal hdbc&, ByVal szConnStrIn$, ByVal cbConnStrIn%, ByVal szConnStrOut$, ByVal cbConnStrOutMax%, pcbConnStrOut%) As Integer

CODE:

```
Global msg As String
Global szConnectOut As String * 512
Global cbConnectOut As Integer
Global dbstr As String


Private Sub mc2brocon_Click()
     retcode = SQLAllocEnv(hEnv&)
     retcode = SQLAllocConnect(hEnv&, hdbc&)
          xconn = 1
          errorcheck retcode
     dbstr = "DSN=C:\RBTI\OTERRO11\vbtypes;" & vbNullChar
     msg = "initial values"
     dbstr = InputBox(msg, "SQLBrowseConnect", Chop(dbstr)) & vbNullChar
     retcode = SQLBrowseConnect(hdbc&, dbstr, SQL_NTS, szConnectOut, 512,
cbConnectOut)
     If retcode = SQL_NEED_DATA Then
          Do While SQLBrowseConnect(hdbc&, dbstr, SQL_NTS, szConnectOut,
512, cbConnectOut) = SQL_NEED_DATA
               msg = szConnectOut
               dbstr = InputBox(msg, "SQLBrowseConnect", Chop(dbstr)) &
vbNullChar
          Loop
     End If
     retcode = SQLAllocStmt(hdbc&, hStmt&)
```

```
            xconn = 0
            errorcheck retcode
End Sub
```

4.8 SQLBulkOperations

SQLBulkOperations performs bulk insertions and bulk bookmark operations, including update, delete, and fetch by bookmark.

Syntax

RETCODE = **SQLBulkOperations**(*StatementHandle*, *Operation*)

Arguments

Type	Argument	Use	Description
Long	StatementHandle	Input	Statement handle.
Integer	Operation	Input	Operation to perform: SQL_ADD SQL_UPDATE_BY_BOOKMARK SQL_DELETE_BY_BOOKMARK SQL_FETCH_BY_BOOKMARK

Return Values

SQL_SUCCESS, SQL_SUCCESS_WITH_INFO, SQL_NEED_DATA, SQL_STILL_EXECUTING, SQL_ERROR, or SQL_INVALID_HANDLE.

Errors

SQLSTATE	Error	Description
01000	General warning	Driver-specific informational message. (Function returns SQL_SUCCESS_WITH_INFO.)
01004	String data right truncation	The Operation argument was SQL_FETCH_BY_BOOKMARK, and string or binary data returned for a column or columns with a data type of SQL_C_CHAR or SQL_C_BINARY resulted in the truncation of nonblank character or non-NULL binary data.
01S01	Error in row	The Operation argument was SQL_ADD, and an error occurred in one or more rows while performing the operation but at least one row was successfully added. (Function returns SQL_SUCCESS_WITH_INFO.) (This error is raised only when an application is working with an ODBC 2.x driver.)
01S07	Fractional truncation	The Operation argument was SQL_FETCH_BY_BOOKMARK, the data type of the application buffer was not SQL_C_CHAR or SQL_C_BINARY, and the data returned to application buffers for one or more columns was truncated. (For numeric C data types, the fractional part of the number was truncated. For time, timestamp, and interval C data types that contain a time component, the fractional portion of the time was truncated.) (Function returns SQL_SUCCESS_WITH_INFO.)
07006	Restricted data type attribute violation	The Operation argument was SQL_FETCH_BY_BOOKMARK, and the data value of a column in the result set could not be converted to the data type specified by the TargetType argument in the call to SQLBindCol.

		The Operation argument was SQL_UPDATE_BY_BOOKMARK or SQL_ADD, and the data value in the application buffers could not be converted to the data type of a column in the result set.
07009	Invalid descriptor index	The argument Operation was SQL_ADD, and a column was bound with a column number greater than the number of columns in the result set.
21S02	Degree of derived table does not match column list	The argument Operation was SQL_UPDATE_BY_BOOKMARK; and no columns were updatable because all columns were either unbound or read-only, or the value in the bound length/indicator buffer was SQL_COLUMN_IGNORE.
22001	String data right truncation	The assignment of a character or binary value to a column in the result set resulted in the truncation of nonblank (for characters) or non-null (for binary) characters or bytes.
22003	Numeric value out of range	The Operation argument was SQL_ADD or SQL_UPDATE_BY_BOOKMARK, and the assignment of a numeric value to a column in the result set caused the whole (as opposed to fractional) part of the number to be truncated. The argument Operation was SQL_FETCH_BY_BOOKMARK, and returning the numeric value for one or more bound columns would have caused a loss of significant digits.
22007	Invalid datetime format	The Operation argument was SQL_ADD or SQL_UPDATE_BY_BOOKMARK, and the assignment of a date or timestamp value to a column in the result set caused the year, month, or day field to be out of range. The argument Operation was SQL_FETCH_BY_BOOKMARK, and returning the date or timestamp value for one or more bound columns would have caused the year, month, or day field to be out of range.
22008	Date/time field overflow	The Operation argument was SQL_ADD or SQL_UPDATE_BY_BOOKMARK, and the performance of datetime arithmetic on data being sent to a column in the result set resulted in a datetime field (the year, month, day, hour, minute, or second field) of the result falling outside the permissible range of values for the field or being invalid based on the Gregorian calendar's natural rules for datetimes. The Operation argument was SQL_FETCH_BY_BOOKMARK, and the performance of datetime arithmetic on data being retrieved from the result set resulted in a datetime field (the year, month, day, hour, minute, or second field) of the result falling outside the permissible range of values for the field or being invalid based on the Gregorian calendar's natural rules for datetimes.
22015	Interval field overflow	The Operation argument was SQL_ADD or SQL_UPDATE_BY_BOOKMARK, and the assignment of an exact numeric or interval C type to an interval SQL data type caused a loss of significant digits. The Operation argument was SQL_ADD or SQL_UPDATE_BY_BOOKMARK; when assigning to an interval SQL type, there was no representation of the value of the C type in the interval SQL type. The Operation argument was SQL_FETCH_BY_BOOKMARK, and assigning from an exact numeric or interval SQL type to an interval C type caused a loss of significant digits in the leading field. The Operation argument was SQL_FETCH_BY_BOOKMARK; when assigning to an interval C type, there was no

		representation of the value of the SQL type in the interval C type.
22018	Invalid character value for cast specification	The Operation argument was SQL_FETCH_BY_BOOKMARK; the C type was an exact or approximate numeric, a datetime, or an interval data type; the SQL type of the column was a character data type; and the value in the column was not a valid literal of the bound C type. The argument Operation was SQL_ADD or SQL_UPDATE_BY_BOOKMARK; the SQL type was an exact or approximate numeric, a datetime, or an interval data type; the C type was SQL_C_CHAR; and the value in the column was not a valid literal of the bound SQL type.
23000	Integrity constraint violation	The Operation argument was SQL_ADD, SQL_DELETE_BY_BOOKMARK, or SQL_UPDATE_BY_BOOKMARK, and an integrity constraint was violated. The Operation argument was SQL_ADD, and a column that was not bound is defined as NOT NULL and has no default. The Operation argument was SQL_ADD, the length specified in the bound StrLen_or_IndPtr buffer was SQL_COLUMN_IGNORE, and the column did not have a default value.
24000	Invalid cursor state	The StatementHandle was in an executed state, but no result set was associated with the StatementHandle.
40001	Serialization failure	The transaction was rolled back because of a resource deadlock with another transaction.
40003	Statement completion unknown	The associated connection failed during the execution of this function, and the state of the transaction cannot be determined.
42000	Syntax error or access violation	The driver was unable to lock the row as needed to perform the operation requested in the Operation argument.
44000	WITH CHECK OPTION violation	The Operation argument was SQL_ADD or SQL_UPDATE_BY_BOOKMARK, and the insert or update was performed on a viewed table (or a table derived from the viewed table) that was created by specifying WITH CHECK OPTION, in such a way that one or more rows affected by the insert or update will no longer be present in the viewed table.
HY000	General error	An error occurred for which there was no specific SQLSTATE and for which no implementation-specific SQLSTATE was defined. The error message returned by SQLGetDiagRec in the *MessageText buffer describes the error and its cause.
HY001	Memory allocation error	The driver was unable to allocate memory required to support execution or completion of the function.
HY008	Operation canceled	Asynchronous processing was enabled for the StatementHandle. The function was called, and before it completed execution, SQLCancel was called on the StatementHandle. Then the function was called again on the StatementHandle. The function was called, and before it completed execution, SQLCancel was called on the StatementHandle from a different thread in a multithread application.
HY010	Function sequence error	(DM) The specified StatementHandle was not in an executed state. The function was called without first calling SQLExecDirect, SQLExecute, or a catalog function. (DM) An asynchronously executing function (not this one) was called for the StatementHandle and was still executing when this function was called. (DM) SQLExecute, SQLExecDirect, or SQLSetPos was called for the StatementHandle and returned SQL_NEED_DATA. This function was called before data was sent for all data-at-execution parameters or columns.

		(DM) The driver was an ODBC 2.x driver, and SQLBulkOperations was called for a StatementHandle before SQLFetchScroll or SQLFetch was called. (DM) SQLBulkOperations was called after SQLExtendedFetch was called on the StatementHandle.
HY011	Attribute cannot be set now	(DM) The driver was an ODBC 2.x driver, and the SQL_ATTR_ROW_STATUS_PTR statement attribute was set between calls to SQLFetch or SQLFetchScroll and SQLBulkOperations.
HY013	Memory management error	The function call could not be processed because the underlying memory objects could not be accessed, possibly because of low memory conditions.
HY090	Invalid string or buffer length	The Operation argument was SQL_ADD or SQL_UPDATE_BY_BOOKMARK; a data value was not a null pointer; the C data type was SQL_C_BINARY or SQL_C_CHAR; and the column length value was less than 0, but not equal to SQL_DATA_AT_EXEC, SQL_COLUMN_IGNORE, SQL_NTS, or SQL_NULL_DATA, or less than or equal to SQL_LEN_DATA_AT_EXEC_OFFSET. The value in a length/indicator buffer was SQL_DATA_AT_EXEC; the SQL type was either SQL_LONGVARCHAR, SQL_LONGVARBINARY, or a long data source–specific data type; and the SQL_NEED_LONG_DATA_LEN information type in SQLGetInfo was "Y". The Operation argument was SQL_ADD, the SQL_ATTR_USE_BOOKMARK statement attribute was set to SQL_UB_VARIABLE, and column 0 was bound to a buffer whose length was not equal to the maximum length for the bookmark for this result set. (This length is available in the SQL_DESC_OCTET_LENGTH field of the IRD and can be obtained by calling SQLDescribeCol, SQLColAttribute, or SQLGetDescField.)
HY092	Invalid attribute identifier	(DM) The value specified for the Operation argument was invalid. The Operation argument was SQL_ADD, SQL_UPDATE_BY_BOOKMARK, or SQL_DELETE_BY_BOOKMARK, and the SQL_ATTR_CONCURRENCY statement attribute was set to SQL_CONCUR_READ_ONLY. The Operation argument was SQL_DELETE_BY_BOOKMARK, SQL_FETCH_BY_BOOKMARK, or SQL_UPDATE_BY_BOOKMARK, and the bookmark column was not bound or the SQL_ATTR_USE_BOOKMARKS statement attribute was set to SQL_UB_OFF.
HYC00	Optional feature not implemented	The driver or data source does not support the operation requested in the Operation argument.
HYT00	Timeout expired	The query timeout period expired before the data source returned the result set. The timeout period is set through SQLSetStmtAttr with an Attribute argument of SQL_ATTR_QUERY_TIMEOUT.

Comments

An application uses SQLBulkOperations to perform the following operations on the base table or view that corresponds to the current query:

- Add new rows.
- Update a set of rows where each row is identified by a bookmark.

- Delete a set of rows where each row is identified by a bookmark.
- Fetch a set of rows where each row is identified by a bookmark.

After a call to SQLBulkOperations, the block cursor position is undefined. The application has to call SQLFetchScroll to set the cursor position. An application should call SQLFetchScroll only with a FetchOrientation argument of SQL_FETCH_FIRST, SQL_FETCH_LAST, SQL_FETCH_ABSOLUTE, or SQL_FETCH_BOOKMARK. The cursor position is undefined if the application calls SQLFetch or SQLFetchScroll with a FetchOrientation argument of SQL_FETCH_PRIOR, SQL_FETCH_NEXT, or SQL_FETCH_RELATIVE.

A column can be ignored in bulk operations performed by a call to SQLBulkOperations by setting the column length/indicator buffer specified in the call to SQLBindCol, to SQL_COLUMN_IGNORE.

It is not necessary for the application to set the SQL_ATTR_ROW_OPERATION_PTR statement attribute when it calls SQLBulkOperations because rows cannot be ignored when performing bulk operations with this function.

The buffer pointed to by the SQL_ATTR_ROWS_FETCHED_PTR statement attribute contains the number of rows affected by a call to SQLBulkOperations.

When the Operation argument is SQL_ADD or SQL_UPDATE_BY_BOOKMARK and the select-list of the query specification associated with the cursor contains more than one reference to the same column, it is driver-defined whether an error is generated or the driver ignores the duplicated references and performs the requested operations.

Related Functions

Function	Description
SQLBindCol	Binding a buffer to a column in a result set
SQLCancel	Canceling statement processing
SQLFetchScroll	Fetching a block of data or scrolling through a result set
SQLSetPos	Positioning the cursor, refreshing data in the rowset, or updating or deleting data in the rowset
SQLSetStmtAttr	Setting a statement attribute

Code Example

The following example fetches 10 rows of data at a time from the Customers table. It then prompts the user for an action to take. To reduce network traffic, the example buffer updates, deletes, and inserts locally in the bound arrays, but at offsets past the rowset data. When the user chooses to send updates, deletes, and inserts to the data source, the code sets the binding offset appropriately and calls SQLBulkOperations. For simplicity, the user cannot buffer more than 10 updates, deletes, or inserts.

```
// SQLBulkOperations_Function.cpp
// compile with: ODBC32.lib
#include <windows.h>
#include <sqlext.h>
#include "stdio.h"

#define UPDATE_ROW 100
#define DELETE_ROW 101
#define ADD_ROW 102
#define SEND_TO_DATA_SOURCE 103
#define UPDATE_OFFSET 10
#define INSERT_OFFSET 20
#define DELETE_OFFSET 30

// Define structure for customer data (assume 10 byte maximum bookmark
size).
```

```
typedef struct tagCustStruct {
   SQLCHAR Bookmark[10];
   SQLINTEGER BookmarkLen;
   SQLUINTEGER CustomerID;
   SQLINTEGER CustIDInd;
   SQLCHAR CompanyName[51];
   SQLINTEGER NameLenOrInd;
   SQLCHAR Address[51];
   SQLINTEGER AddressLenOrInd;
   SQLCHAR Phone[11];
   SQLINTEGER PhoneLenOrInd;
} CustStruct;

// Allocate 40 of these structures. Elements 0-9 are for the current
rowset,
// elements 10-19 are for the buffered updates, elements 20-29 are for
// the buffered inserts, and elements 30-39 are for the buffered deletes.
CustStruct CustArray[40];
SQLUSMALLINT RowStatusArray[10], Action, RowNum, NumUpdates = 0, NumInserts
= 0,
NumDeletes = 0;
SQLINTEGER BindOffset = 0;
SQLRETURN retcode;
SQLHENV henv = NULL;
SQLHDBC hdbc = NULL;
SQLPOINTER rgbValue;
SQLHSTMT hstmt = NULL;

int main() {
   retcode = SQLAllocHandle(SQL_HANDLE_ENV, SQL_NULL_HANDLE, &henv);
   retcode = SQLSetEnvAttr(henv, SQL_ATTR_ODBC_VERSION, (SQLPOINTER*)
SQL_OV_ODBC3, 0);

   retcode = SQLAllocHandle(SQL_HANDLE_DBC, henv, &hdbc);
   retcode = SQLSetConnectAttr(hdbc, SQL_LOGIN_TIMEOUT, (SQLPOINTER)
(rgbValue), 0);

   retcode = SQLConnect(hdbc, (SQLCHAR*) "Northwind", SQL_NTS, (SQLCHAR*)
NULL, 0, NULL, 0);
   retcode = SQLAllocHandle(SQL_HANDLE_STMT, hdbc, &hstmt);

   // Set the following statement attributes:
   // SQL_ATTR_CURSOR_TYPE:           Keyset-driven
   // SQL_ATTR_ROW_BIND_TYPE:         Row-wise
   // SQL_ATTR_ROW_ARRAY_SIZE:        10
   // SQL_ATTR_USE_BOOKMARKS:         Use variable-length bookmarks
   // SQL_ATTR_ROW_STATUS_PTR:        Points to RowStatusArray
   // SQL_ATTR_ROW_BIND_OFFSET_PTR:   Points to BindOffset
   retcode = SQLSetStmtAttr(hstmt, SQL_ATTR_CURSOR_TYPE, (SQLPOINTER)
SQL_CURSOR_KEYSET_DRIVEN, 0);
   retcode = SQLSetStmtAttr(hstmt, SQL_ATTR_ROW_BIND_TYPE, (SQLPOINTER)
sizeof(CustStruct), 0);
   retcode = SQLSetStmtAttr(hstmt, SQL_ATTR_ROW_ARRAY_SIZE, (SQLPOINTER)10,
0);
```

```
   retcode = SQLSetStmtAttr(hstmt, SQL_ATTR_USE_BOOKMARKS, (SQLPOINTER)
SQL_UB_VARIABLE, 0);
   retcode = SQLSetStmtAttr(hstmt, SQL_ATTR_ROW_STATUS_PTR, RowStatusArray,
0);
   retcode = SQLSetStmtAttr(hstmt, SQL_ATTR_ROW_BIND_OFFSET_PTR,
&BindOffset, 0);

   // Bind arrays to the bookmark, CustomerID, CompanyName, Address, and
Phone columns.
   retcode = SQLBindCol(hstmt, 0, SQL_C_VARBOOKMARK, CustArray[0].Bookmark,
sizeof(CustArray[0].Bookmark), &CustArray[0].BookmarkLen);
   retcode = SQLBindCol(hstmt, 1, SQL_C_ULONG, &CustArray[0].CustomerID, 0,
&CustArray[0].CustIDInd);
   retcode = SQLBindCol(hstmt, 2, SQL_C_CHAR, CustArray[0].CompanyName,
sizeof(CustArray[0].CompanyName), &CustArray[0].NameLenOrInd);
   retcode = SQLBindCol(hstmt, 3, SQL_C_CHAR, CustArray[0].Address,
sizeof(CustArray[0].Address), &CustArray[0].AddressLenOrInd);
   retcode = SQLBindCol(hstmt, 4, SQL_C_CHAR, CustArray[0].Phone,
sizeof(CustArray[0].Phone), &CustArray[0].PhoneLenOrInd);

   // Execute a statement to retrieve rows from the Customers table.
   retcode = SQLExecDirect(hstmt, (SQLCHAR*)"SELECT CustomerID,
CompanyName, Address, Phone FROM Customers", SQL_NTS);

   // Fetch and display the first 10 rows.
   retcode = SQLFetchScroll(hstmt, SQL_FETCH_NEXT, 0);
   // DisplayCustData(CustArray, 10);

   // Call GetAction to get an action and a row number from the user.
   // while (GetAction(&Action, &RowNum)) {
   Action = SQL_FETCH_NEXT;
   RowNum = 2;
   switch (Action) {
      case SQL_FETCH_NEXT:
      case SQL_FETCH_PRIOR:
      case SQL_FETCH_FIRST:
      case SQL_FETCH_LAST:
      case SQL_FETCH_ABSOLUTE:
      case SQL_FETCH_RELATIVE:
         // Fetch and display the requested data.
         SQLFetchScroll(hstmt, Action, RowNum);
         // DisplayCustData(CustArray, 10);
         break;

      case UPDATE_ROW:
         // Check if we have reached the maximum number of buffered
updates.
         if (NumUpdates < 10) {
            // Get the new customer data and place it in the next available
element of
            // the buffered updates section of CustArray, copy the bookmark
of the row
            // being updated to the same element, and increment the update
counter.
```

```
            // Checking to see we have not already buffered an update for
this
            // row not shown.
            // GetNewCustData(CustArray, UPDATE_OFFSET + NumUpdates);
            memcpy(CustArray[UPDATE_OFFSET + NumUpdates].Bookmark,
               CustArray[RowNum - 1].Bookmark,
               CustArray[RowNum - 1].BookmarkLen);
            CustArray[UPDATE_OFFSET + NumUpdates].BookmarkLen =
               CustArray[RowNum - 1].BookmarkLen;
            NumUpdates++;
         } else {
            printf("Buffers full. Send buffered changes to the data
source.");
         }
         break;
      case DELETE_ROW:
         // Check if we have reached the maximum number of buffered
deletes.
         if (NumDeletes < 10) {
            // Copy the bookmark of the row being deleted to the next
available element
            // of the buffered deletes section of CustArray and increment
the delete
            // counter. Checking to see we have not already buffered an
update for
            // this row not shown.
            memcpy(CustArray[DELETE_OFFSET + NumDeletes].Bookmark,
               CustArray[RowNum - 1].Bookmark,
               CustArray[RowNum - 1].BookmarkLen);

            CustArray[DELETE_OFFSET + NumDeletes].BookmarkLen =
               CustArray[RowNum - 1].BookmarkLen;

            NumDeletes++;
         } else
            printf("Buffers full. Send buffered changes to the data
source.");
         break;

      case ADD_ROW:
         // reached maximum number of buffered inserts?
         if (NumInserts < 10) {
            // Get the new customer data and place it in the next available
element of
            // the buffered inserts section of CustArray and increment
insert counter.
            // GetNewCustData(CustArray, INSERT_OFFSET + NumInserts);
            NumInserts++;
         } else
            printf("Buffers full. Send buffered changes to the data
source.");
         break;

      case SEND_TO_DATA_SOURCE:
         // If there are any buffered updates, inserts, or deletes, set the
array size
         // to that number, set the binding offset to use the data in the
buffered
         // update, insert, or delete part of CustArray, and call
SQLBulkOperations to
         // do the updates, inserts, or deletes. Because we will never have
more than
         // 10 updates, inserts, or deletes, we can use the same row status
array.
         if (NumUpdates) {
            SQLSetStmtAttr(hstmt, SQL_ATTR_ROW_ARRAY_SIZE, (SQLPOINTER)
NumUpdates, 0);
            BindOffset = UPDATE_OFFSET * sizeof(CustStruct);
            SQLBulkOperations(hstmt, SQL_UPDATE_BY_BOOKMARK);
            NumUpdates = 0;
         }

         if (NumInserts) {
            SQLSetStmtAttr(hstmt, SQL_ATTR_ROW_ARRAY_SIZE, (SQLPOINTER)
NumInserts, 0);
            BindOffset = INSERT_OFFSET * sizeof(CustStruct);
            SQLBulkOperations(hstmt, SQL_ADD);
            NumInserts = 0;
         }

         if (NumDeletes) {
            SQLSetStmtAttr(hstmt, SQL_ATTR_ROW_ARRAY_SIZE, (SQLPOINTER)
NumDeletes, 0);
            BindOffset = DELETE_OFFSET * sizeof(CustStruct);
            SQLBulkOperations(hstmt, SQL_DELETE_BY_BOOKMARK);
            NumDeletes = 0;
         }

         // If there were any updates, inserts, or deletes, reset the
binding offset
         // and array size to their original values.
         if (NumUpdates || NumInserts || NumDeletes) {
            SQLSetStmtAttr(hstmt, SQL_ATTR_ROW_ARRAY_SIZE, (SQLPOINTER)10,
0);
            BindOffset = 0;
         }
         break;
   }
   // }

   // Close the cursor.
   SQLFreeStmt(hstmt, SQL_CLOSE);
}
```

4.9 SQLCancel

SQLCancel closes the cursor associated with the hstmt and discards all pending results.

Syntax

RETCODE = SQLCancel (hstmt)

Arguments

Type	Argument	Use	Description
Long	hstmt	Input	The statement handle.

Return Values

```
SQL_SUCCESS, SQL_ERROR, or SQL_INVALID_HANDLE
```

Comments

SQLCancel has the same effect as calling SQLFreeStmt with SQL_CLOSE as the option.

Related Functions

Function	Description
SQLExecDirect	Executes an SQL statement.
SQLExecute	Executes a prepared SQL statement.
SQLExtendedFetch	Fetches one row of a result set; allows scrolling.
SQLFreeStmt	Ends processing on the statement.
SQLSetStmtOption	Sets options for a statement handle.
SQLSpecialColumns	Returns information about a set of columns.
SQLStatistics	Returns statistics for tables and indexes.

Errors

SQLSTATE	Description
S1000	An error has occurred that has no defined SQLSTATE—see the error message text.
S1001	A memory allocation failure.

Visual Basic Example

SQLAPI.BAS:
Declare Function SQLCancel Lib "ODBC32.DLL" (ByVal hstmt&) As Integer

CODE:
```
Private Sub ms2native_Click()
     Dim sqlstring as String
     Dim cblong1 As Long
     Dim nativesql As String * 512
     sqlstring = "SELECT * FROM numbers" & vbNullChar
     retcode = SQLNativeSql(hdbc&, sqlstring, SQL_NTS, nativesql, 512,
cblong1)
          errorcheck retcode
     bufstring = InputBox(sqlstring, "SQLNativeSql", nativesql)
```

```
    retcode = SQLCancel(hStmt&)
End Sub
```

4.10 SQLCloseCursor

SQLCloseCursor closes a cursor that has been opened on a statement and discards pending results.

Syntax

RETCODE = SQLCloseCursor(StatementHandle)

Arguments

Type	Argument	Use	Description
Long	StatementHandle	Input	Statement handle.

Return Values

SQL_SUCCESS, SQL_SUCCESS_WITH_INFO, SQL_ERROR, or SQL_INVALID_HANDLE

Errors

When SQLCloseCursor returns SQL_ERROR or SQL_SUCCESS_WITH_INFO, an associated SQLSTATE value may be obtained by calling SQLGetDiagRec with a HandleType of SQL_HANDLE_STMT and a Handle of StatementHandle. The following table lists the SQLSTATE values commonly returned by SQLCloseCursor and explains each one in the context of this function; the notation "(DM)" precedes the descriptions of SQLSTATEs returned by the Driver Manager. The return code associated with each SQLSTATE value is SQL_ERROR, unless noted otherwise.

SQLSTATE	Error	Description
01000	General warning	Driver-specific informational message. (Function returns SQL_SUCCESS_WITH_INFO.)
24000	Invalid cursor state	No cursor was open on the StatementHandle. (This is returned only by an ODBC 3.x driver.)
HY000	General error	An error occurred for which there was no specific SQLSTATE and for which no implementation-specific SQLSTATE was defined. The error message returned by SQLGetDiagRec in the *MessageText buffer describes the error and its cause.
HY001	Memory allocation error	The driver was unable to allocate memory required to support execution or completion of the function.
HY010	Function sequence error	(DM) An asynchronously executing function was called for the StatementHandle and was still executing when this function was called. (DM) SQLExecute, SQLExecDirect, SQLBulkOperations, or SQLSetPos was called for the StatementHandle and returned SQL_NEED_DATA. This function was called before data was sent for all data-at-execution parameters or columns.
HY013	Memory management error	The function call could not be processed because the underlying memory objects could not be accessed, possibly because of low memory conditions.
HYT01	Connection timeout expired	The connection timeout period expired before the data source responded to the request. The connection timeout period is set through SQLSetConnectAttr, SQL_ATTR_CONNECTION_TIMEOUT.
IM001	Driver does not support this function	(DM) The driver associated with the StatementHandle does not support the function.

Comments

SQLCloseCursor returns SQLSTATE 24000 (Invalid cursor state) if no cursor is open. Calling SQLCloseCursor is equivalent to calling SQLFreeStmt with the SQL_CLOSE option, with the exception that SQLFreeStmt with SQL_CLOSE has no effect on the application if no cursor is open on the statement, while SQLCloseCursor returns SQLSTATE 24000 (Invalid cursor state).

Note: If an ODBC 3.x application working with an ODBC 2.x driver calls SQLCloseCursor when no cursor is open, SQLSTATE 24000 (Invalid cursor state) is not returned, because the Driver Manager maps SQLCloseCursor to SQLFreeStmt with SQL_CLOSE.

Related Functions

Function	Description
SQLCancel	Canceling statement processing
SQLFreeHandle	Freeing a handle
SQLMoreResults	Processing multiple result sets

Code Example

```
// SQLBrowseConnect_Function.cpp
// compile with: odbc32.lib
#include <windows.h>
#include <sqltypes.h>
#include <sqlext.h>

#define BRWS_LEN 100
SQLHENV henv;
SQLHDBC hdbc;
SQLHSTMT hstmt;
SQLRETURN retcode;
SQLCHAR szConnStrIn[BRWS_LEN], szConnStrOut[BRWS_LEN];
SQLSMALLINT cbConnStrOut;

void GetUserInput(SQLCHAR * szConnStrOut, SQLCHAR * szConnStrIn) {}

int main() {
   // Allocate the environment handle.
   retcode = SQLAllocHandle(SQL_HANDLE_ENV, SQL_NULL_HANDLE, &henv);
   if (retcode == SQL_SUCCESS || retcode == SQL_SUCCESS_WITH_INFO) {

      // Set the version environment attribute.
      retcode = SQLSetEnvAttr(henv, SQL_ATTR_ODBC_VERSION, (SQLPOINTER*)
SQL_OV_ODBC3, 0);
      if (retcode == SQL_SUCCESS || retcode == SQL_SUCCESS_WITH_INFO) {

         // Allocate the connection handle.
         retcode = SQLAllocHandle(SQL_HANDLE_DBC, henv, &hdbc);
         if (retcode == SQL_SUCCESS || retcode == SQL_SUCCESS_WITH_INFO) {
            // Call SQLBrowseConnect until it returns a value other than
SQL_NEED_DATA
            // (pass data source name the first time).  If SQL_NEED_DATA is
returned, call GetUserInput
```

```
            // (not shown) to build a dialog from the values in
szConnStrOut.  The user-supplied values
            // are returned in szConnStrIn, which is passed in the next
call to SQLBrowseConnect.

            strcpy_s((char*)szConnStrIn, _countof(szConnStrIn),
"DSN=Sales");
            do {
               retcode = SQLBrowseConnect(hdbc, szConnStrIn, SQL_NTS,
                  szConnStrOut, BRWS_LEN, &cbConnStrOut);
               if (retcode == SQL_NEED_DATA)
                  GetUserInput(szConnStrOut, szConnStrIn);
            } while (retcode == SQL_NEED_DATA);

            if (retcode == SQL_SUCCESS || retcode == SQL_SUCCESS_WITH_INFO)
{

               // Allocate the statement handle.
               retcode = SQLAllocHandle(SQL_HANDLE_STMT, hdbc, &hstmt);

               if (retcode == SQL_SUCCESS || retcode ==
SQL_SUCCESS_WITH_INFO)
                  // Process data after successful connection
                  SQLFreeHandle(SQL_HANDLE_STMT, hstmt);
               SQLDisconnect(hdbc);
            }
         }
         SQLFreeHandle(SQL_HANDLE_DBC, hdbc);
      }
   }
   SQLFreeHandle(SQL_HANDLE_ENV, henv);
}
```

4.11 SQLColAttributes

SQLColAttributes returns several different types of information about a result set. The information is either returned as a string or as a 32-bit integer.

Syntax

RETCODE = SQLColAttributes(hstmt, icol, fDescType, rgbDesc, cbDescMax, pcbDesc, pfDesc)

Arguments

Type	Argument	Use	Description
Long	hstmt	Input	The statement handle.
Integer	icol	Input	The column number in the result data, starting at 1.
Integer	fDescType	Input	The type of information desired.
String	rgbDesc	Output	Where to put the string output data.
Integer	cbDescMax	Input	The maximum number of bytes to store in the rgbDesc buffer.
Integer	pcbDesc	Output	The number of bytes placed in the rgbDesc buffer.
Long	pfDesc	Output	A pointer to a 32-bit integer result.

Return Values

```
SQL_SUCCESS, SQL_SUCCESS_WITH_INFO, SQL_ERROR, or SQL_INVALID_HANDLE
```

Information Types

fDescType Constant	Description
SQL_COLUMN_AUTO_INCREMENT	1 if icol is an auto-increment column, 0 otherwise. By ODBC definition, the Oterro database autonumber columns are not auto-increment columns; therefore, this has no significance on Oterro database autonumber columns.
SQL_COLUMN_CASE_SENSITIVE	1 if sorts on icol are case sensitive, 0 otherwise.
SQL_COLUMN_COUNT	The number of columns in this result set; when calling with this parameter, icol can be zero.
SQL_COLUMN_DISPLAY_SIZE	The number of spaces which are required to display this column on the screen.
SQL_COLUMN_LABEL	The label for the column specified by icol.
SQL_COLUMN_LENGTH	The storage size of the column specified by icol. The space this column requires when fetching with bound columns or calling SQLGetData with SQL_C_DEFAULT as the data type.
SQL_COLUMN_MONEY	1 if icol is the currency data type, 0 otherwise.
SQL_COLUMN_NAME	The name of the column specified by icol.
SQL_COLUMN_NULLABLE	1 if a NULL value is legal in this column, 0 if NULLs are not allowed.
SQL_COLUMN_OWNER_NAME	The owner of the table containing the column specified by icol.
SQL_COLUMN_PRECISION	The numeric precision of the column icol; Oterro database NUMERIC data type only.
SQL_COLUMN_QUALIFIER_NAME	The qualifier of the table containing the column specified by icol.
SQL_COLUMN_SCALE	The numeric scale of icol; Oterro database NUMERIC data type only.
SQL_COLUMN_SEARCHABLE	This set defines which operators can be used to search on the column specified by icol. The operators are: SQL_UNSEARCHABLE (0) SQL_LIKE_ONLY (1) SQL_ALL_EXCEPT_LIKE (2) SQL_SEARCHABLE (3) Oterro uses operators SQL_ALL_EXCEPT_LIKE (2) and SQL_SEARCHABLE (3).
SQL_COLUMN_TABLE_NAME	The name of the table containing the column specified by icol.
SQL_COLUMN_TYPE	The SQL data type of the column specified by icol.
SQL_COLUMN_TYPE_NAME	The Oterro name for the data type of the column specified by icol.
SQL_COLUMN_UNSIGNED	1 if icol is unsigned, 0 otherwise. Oterro Engine returns 0, it does not allow unsigned.
SQL_COLUMN_UPDATABLE	Depending on the read/write permissions of icol, either: SQL_ATTR_READONLY (0) SQL_ATTR_WRITE (1) SQL_ATTR_READWRITE_UNKNOWN (2)

Related Functions

Function	Description
SQLDescribeCol	Describes a column in a result set.
SQLFetch	Fetches one row of a result set.
SQLGetData	Gets result data for a column in a result set.
SQLGetTypeInfo	Returns information about the data types in the database.
SQLNumResultCols	Returns the number of columns in a result set.

Errors

SQLSTATE	Description

01004	The data was truncated (SQL_SUCCESS_WITH_INFO was returned).
24000	An invalid cursor state: A cursor is currently open on the statement handle.
S1000	An error has occurred that has no defined SQLSTATE—see the error message text.
S1001	A memory allocation failure.
S1002	An invalid column number.
S1090	An invalid string or buffer length.
S1091	The descriptor type was out of range—fDescType was not a valid information type.

Visual Basic Example

SQLAPI.BAS:
Declare Function SQLColAttributes Lib "ODBC32.DLL" (ByVal hstmt&, ByVal icol%, ByVal fDescType%, ByVal rgbDesc$, ByVal cbDescMax%, pcbDesc%, pfDesc&) As Integer

CODE:

```
Global xarray1(50) As Variant
Global sqlstring As String


Private Sub ms2descr_Click()
     Dim i As Integer
     Dim n As Integer
     Dim szcol11 As String * 18
     Dim cbcol11 As Integer
     Dim cbcol12 As Integer
     Dim cblong1 As Long
     Dim cbcol13 As Integer
     Dim cbcol14 As Integer
     sqlstring = "select * from texts" & vbNullChar
     retcode = SQLExecDirect(hstmt&, sqlstring, SQL_NTS)
          errorcheck retcode
     retcode = SQLNumResultCols(hStmt&, colnum)
          errorcheck retcode
     i = 1
     Do While i <= colnum
          retcode = SQLDescribeCol(hStmt&, i, szcol11, 18, cbcol11,
cbcol12, cblong1, cbcol13, cbcol14)
               errorcheck retcode
          xarray1(i) = Chop(szcol11)
          retcode = SQLColAttributes(hStmt&, i, SQL_COLUMN_DISPLAY_SIZE,
szcol11, 18, cbcol11, cblong1)
               errorcheck retcode
          xarray1(i) = xarray1(i) & "; " & cblong1
          retcode = SQLColAttributes(hStmt&, i, SQL_COLUMN_SEARCHABLE,
szcol11, 18, cbcol11, cblong1)
               errorcheck retcode
          xarray1(i) = xarray1(i) & "; " & cblong1
          i = i + 1
     Loop
     view2.List1.Clear
     i = 1
     n = 1
     Do While n <> 0
          view2.List1.AddItem xarray1(i)
          i = i + 1
          n = Len(xarray1(i))
     Loop
     retcode = SQLFreeStmt(hStmt&, SQL_CLOSE)
End Sub
```

4.12 SQLColumnPrivileges

SQLColumn Privileges returns a list of columns and associated privileges for the specified table.

Syntax

RETCODE = SQLColumnPrivileges(hstmt, szTableQualifier, cbTableQualifier, szTableOwner, cbTableOwner, szTableName, cbTableName, szColumnName, cbColumnName)

Arguments

Type	Argument	Use	Description
Long	hstmt	Input	The statement handle.
String	szTableQualifier	Input	The table qualifier.
Integer	cbTableQualifier	Input	The length of the table qualifier.
String	szTableOwner	Input	The name of the table owner.
Integer	cbTableOwner	Input	The length of the table owner name.
String	szTableName	Input	The table name.
Integer	cbTableName	Input	The length of the table name.
String	szColumnName	Input	The string search pattern for column names.
Integer	cbColumnName	Input	The length of the column name.

Return Values

SQL_SUCCESS, SQL_SUCCESS_WITH_INFO, SQL_STILL_EXECUTING, SQL_ERROR, or SQL_INVALID_HANDLE.

Result Set

Column Name	Data Type	Comments
TABLE_QUALIFIER	TEXT 18	The table qualifier. Always NULL for the Oterro Engine.
TABLE_OWNER	TEXT 18	The table owner. Always NULL for the Oterro Engine.
TABLE_NAME	TEXT 18	The table name.
COLUMN_NAME	TEXT 18	The column name.
GRANTOR	TEXT 18	Name of the user who granted the privilege. Always NULL for the Oterro Engine.
GRANTEE	TEXT 18	Name of the user granted the privilege.
PRIVILEGE	TEXT 128	Identifies the column privilege: SELECT, UPDATE, INSERT, DELETE, ALTER, REFERENCE.
IS_GRANTABLE	TEXT 3	Indicates whether the grantee is permitted to grant privileges to other users. Always NULL for the Oterro Engine.

Related Functions

Function	Description
SQLColumns	Returns the columns in a table(s).
SQLExtendedFetch	Fetches one row of a result set; allows scrolling.
SQLFetch	Fetches one row of a result set.
SQLTablePrivileges	Returns the privileges assigned to the table.

SQLTables	Returns the tables in a database.

Errors

SQLSTATE	Description
01000	A driver-specific informational message. (Function returns SQL_SUCCESS_WITH_INFO)
08S01	The data source connection failed before the function completed processing.
24000	An invalid cursor state. A cursor is currently open on the statement handle.
IM001	The driver associated with the hstmt does not support the function.
S1000	An error has occurred that has no defined SQLSTATE—see the error message text.
S1001	A memory allocation failure.
S1008	An operation was canceled.
S1010	The statement was not prepared.
S1090	An invalid string or buffer length.
S1C00	The driver or data source does not support the specified type.
S1T00	The timeout period expired before the data source returned the result.

Visual Basic Example

SQLAPI.BAS:
Declare Function SQLColumnPrivileges Lib "ODBC32.DLL" (ByVal hstmt&, ByVal szTableQualifier$, ByVal cbTableQualifier%, ByVal szTableOwner$, ByVal cbTableOwner%, ByVal szTableName$, ByVal cbTableName%, ByVal szColumnName$, ByVal cbColumnName%) As Integer

CODE:
```
Global colnum As Integer
Global colresults As String * 5000
Global cbcolresults As Long


Private Sub mdb1cpriv_Click()
     retcode = SQLColumnPrivileges(hStmt&, "", 0, "", 0, "bigtext", 7, "",
0)
          errorcheck retcode
     loadtest
End Sub


Sub loadtest()
     Dim i As Integer
     Dim cblong1 As Long
     Dim cbint1 As Integer
     i = 0
     retcode = SQLNumResultCols(hStmt&, colnum)
          errorcheck retcode
     i = 1
     Do While SQLFetch(hStmt&) = SQL_SUCCESS
          Do While i <= colnum
               retcode = SQLGetData(hStmt&, i, SQL_C_CHAR, colresults,
5000, cbcolresults)
               view1.text1.Text = view1.text1.Text & vbCrLf & "Col" & i &
": " & Chop(colresults)
               i = i + 1
```

```
            Loop
            i = 1
        Loop
        retcode = SQLFreeStmt(hStmt&, SQL_CLOSE)
    End Sub
```

4.13 SQLColumns

SQLColumns returns the list of columns for a given table.

Syntax

RETCODE = SQLColumns(hstmt, szTableQualifier, cbTableQualifier, szTableOwner, cbTableOwner, szTableName, cbTableName, szColumnName, cbColumnName)

Arguments

Type	Argument	Use	Description
Long	hstmt	Input	The statement handle.
String	szTableQualifier	Input	The buffer containing the table qualifier.
Long	cbTableQualifier	Input	The length of the table qualifier.
String	szTableOwner	Input	The buffer containing the table-owner name.
Long	cbTableOwner	Input	The length of the table-owner name.
String	szTableName	Input	The buffer containing the table name.
Long	cbTableName	Input	The length of the table name.
String	szColumnName	Input	The buffer containing the column name.
Long	cbColumnName	Input	The length of the column name.

Return Values

```
SQL_SUCCESS, SQL_ERROR, or SQL_INVALID_HANDLE
```

Result Set

Column Name	Data Type	Comments
TABLE_QUALIFIER	TEXT 18	The table qualifier. Always NULL for the Oterro Engine.
TABLE_OWNER	TEXT 18	The table owner name. Always NULL for the Oterro Engine.
TABLE_NAME	TEXT 18	The table name.
COLUMN_NAME	TEXT 18	The column name.
DATA_TYPE	INTEGER	Either the ODBC or Oterro database data type. See SQLGetTypeInfo for a list of valid data types.
TYPE_NAME	TEXT 8	The textual name of the data type, for example, INTEGER.
PRECISION	INTEGER	The precision of the data type for the COLUMN_NAME.
LENGTH	INTEGER	The length of the data type for COLUMN_NAME.
SCALE	INTEGER	The scale of the data type for COLUMN_NAME.
RADIX	INTEGER	The base of the number system used in the database. 10 for all numeric data types (INTEGER, DOUBLE etc.). NULL for all non-numeric data types (TEXT, DATE, etc.).
NULLABLE	INTEGER	Indicates if NULL values are allowed in the column.
REMARKS	NOTE	A description of the column, if one has been defined.

Copyright © 1982-2025 R:BASE Technologies, Inc.

Comments

The result set is retrieved by using SQLFetch.

Always use NULL for cbTableQualifier and use NULL for cbTableOwner when accessing the Oterro Engine.

Related Functions

Function	Description
SQLColumnPrivileges	Returns the privileges assigned to a column.
SQLSpecialColumns	Returns information about a set of columns.
SQLTablePrivileges	Returns the privileges assigned to the table.
SQLTables	Returns the tables in a database.

Errors

SQLSTATE	Description
01000	A driver-specific informational message. (The function returns SQL_SUCCESS_WITH_INFO.)
08S01	The data source connection failed before the function completed processing.
24000	An invalid cursor state: A cursor is currently open on the statement handle.
IM001	The driver associated with the hstmt does not support the function.
S1000	An error has occurred that has no defined SQLSTATE—see the error message text.
S1001	A memory allocation failure.
S1090	An invalid string or buffer length.
S1C00	The driver or data source does not support the specified type.

Visual Basic Example

SQLAPI.BAS:
Declare Function SQLColumns Lib "ODBC32.DLL" (ByVal hstmt&, ByVal szTableQualifier$, ByVal cbTableQualifier%, ByVal szTableOwner$, ByVal cbTableOwner%, ByVal szTableName$, ByVal cbTableName%, ByVal szColumnName$, ByVal cbColumnName%) As Integer

```
CODE:
Global colnum As Integer
Global szTableName As String * 20
Global cbTableName As Integer
Global szFirst As String * 1500
Global cbFirst As Long


cbTableName = Len(dbstr1.list1.List(list1.ListIndex)) - 2
szTableName = Right(dbstr1.list1.List(list1.ListIndex), cbTableName) &
vbNullChar
retcode = SQLColumns(hstmt&, "", 0, "", 0, szTableName, cbTableName, "", 0)
     errorcheck retcode
loadgrid


Sub loadgrid()
     Dim i As Integer
     Dim n As Integer
     n = 1
     i = 1
     retcode = SQLNumResultCols(hstmt&, colnum)
          errorcheck retcode
     Do While SQLFetch(hstmt&) = SQL_SUCCESS
          dbstr1.Grid1.Row = n
          Do While i <= colnum
               retcode = SQLGetData(hstmt&, i, SQL_C_CHAR, szFirst, 255,
cbFirst)
               If dbstr1.Grid1.ColWidth(i) < Abs(cbFirst) * 120 Then
                    dbstr1.Grid1.ColWidth(i) = Abs(cbFirst) * 120
               End If
               dbstr1.Grid1.Col = i
               dbstr1.Grid1.Text = Chop(szFirst)
               i = i + 1
          Loop
          n = n + 1
          i = 1
     Loop
     retcode = SQLFreeStmt(hstmt&, SQL_CLOSE)
End Sub
```

4.14 SQLConnect

SQLConnect establishes a connection to a database. This connection handle is used by the program to store all the information related to the connection, such as general status information, transaction state, and error information.

Syntax

RETCODE = SQLConnect(hdbc, szDSN, cbDSN, szUID, cbUID, szAuthStr, cbAuthStr)

Arguments

Type	Argument	Use	Description
Long	hdbc	Input	The database connection handle.
String	szDSN	Input	The buffer containing the database name.
Integer	cbDSN	Input	The length of the database name.
String	szUID	Input	The buffer containing the user identifier.
Integer	cbUID	Input	The length of the user identifier.
String	szAuthStr	Input	The buffer containing the password.
Integer	cbAuthStr	Input	The length of the password.

Return Values

```
SQL_SUCCESS, SQL_ERROR, or SQL_INVALID_HANDLE
```

Comments

Database names must contain explicit paths.

When the user does not have access to any of the tables in the database, the value SQL_ERROR is returned.

By default, transaction processing is set ON, and AUTOCOMMIT is set ON for this function. You can override these defaults by calling the function SQLSetConnectOption with the appropriate arguments, or by setting AUTOCOMMIT OFF or TRANSACT OFF in the OTERRO11.CFG file.

When a user identifier and password are not defined, you must send a NULL, or the string "NONE" or "PUBLIC" for both.

When SQLConnect is called without releasing the connection handle, the connection options that were available in the previous connection are available in the new one.

When a database is connected using SQLConnect, the ODBC Driver Manager is not used. To operate through the ODBC Driver Manager, use SQLDriverConnect.

Since the SQLAPI.BAS is defined to connect through the ODBC Driver Manager, this function will not work. To use this function, the SQLAPI.BAS file must be modified to substitute "OTERRO11.DLL" for "ODBC32.DLL" in the Declare Function statements.

Related Functions

Function	Description
SQLAllocConnect	Allocates a connection handle.
SQLBrowseConnect	Retrieves values required to connect to a data source.
SQLDataSources	Returns the data source names.
SQLDisconnect	Closes the connection to a database.
SQLDriverConnect	Prompts for information to open a connection to a database.
SQLDrivers	Returns the driver descriptions.
SQLFreeConnect	Frees the connection handle.
SQLGetConnectOption	Queries the status of a connection option.
SQLSetConnectOption	Sets a database connection option.

Errors

SQLSTATE	Description
08001	Unable to connect to the data source.
08002	The connection is already in use.
08004	The data source rejected the connection.
28000	No access is available for this user.
S1000	An error has occurred that has no defined SQLSTATE—see the error message text.
S1001	A memory allocation failure.
S1090	An invalid string or buffer length.

Visual Basic Example

SQLAPI.BAS:
Declare Function SQLConnect Lib "ODBC32.DLL" (ByVal hdbc&, ByVal szDSN$, ByVal cbDSN%, ByVal szUID$, ByVal cbUID%, ByVal szAuthStr$, ByVal cbAuthStr%) As Integer

CODE:
```
Global Const xdisco = "No Database Connected"
Global dbstr As String

dbstr = "c:\database\mydata"
dbconn dbstr

Sub dbconn(dbstr As String) 'uses dbstr for database name
```

```
        retcode = SQLAllocConnect(henv&, hdbc&)
        retcode = SQLConnect(hdbc&, dbstr, SQL_NTS, " ", 0, " ", 0)
             errorcheck retcode
        If retcode = 0 Then
        Else
             dbstr = xdisco
        End If
        retcode = SQLAllocStmt(hdbc&, hstmt&)
             errorcheck retcode
   End Sub
```

4.15 SQLDataSources

SQLDataSources returns a list of data source names from the ODBC Driver Manager.

Syntax

RETCODE = SQLDataSources(hEnv, fDirection, pucDSN, sDSNMaxLen, psDSNLen, pucDesc, sDescMaxLen, psDescLen)

Arguments

Type	Argument	Use	Description
Long	hEnv	Input	The environment handle.
Integer	fDirection	Input	Determines if the Driver Manager fetches the next data source name in the list (SQL_FETCH_NEXT) or the first (SQL_FETCH_FIRST).
String	pucDSN	Output	Pointer to storage for the data source name.
Integer	sDSNMaxLen	Input	Maximum length of the pucDSN buffer.
Integer	psDSNLen	Output	Total number of bytes available to return in pucDSN.
String	pucDesc	Output	Pointer to storage for the description of the driver associated with the data source.
Integer	sDescMaxLen	Input	Maximum length of the pucDesc buffer.
Integer	psDescLen	Output	Total number of bytes available to return in pucDesc.

Return Values

SQL_SUCCESS, SQL_SUCCESS_WITH_INFO, SQL_NO_DATA_FOUND, SQL_ERROR, or SQL_INVALID_HANDLE

Comments

Because SQLDataSources is implemented by the ODBC Driver Manager it is supported for all drivers. SQLDataSources allows you to query the list of data source names without displaying a secondary window as with SQLDriverConnect.

Related Functions

Function	Description
SQLBrowseConnect	Retrieves values required to connect to a data source.
SQLConnect	Opens a connection to a database.
SQLDriverConnect	Prompts for information to open a connection to a database.

SQLDrivers	Returns driver descriptions and attributes.

Errors

SQLSTATE	Description
01000	A Driver Manager-specific informational message (Function returns SQL_SUCCESS_WITH_INFO).
01004	The data was truncated (SQL_SUCCESS_WITH_INFO was returned).
S1000	An error has occurred that has no defined SQLSTATE—see the error message text.
S1001	A memory allocation failure.
S1090	An invalid string or buffer length.
S1103	The direction option was out of range.

Visual Basic Example

SQLAPI.BAS:

Declare Function SQLDataSources Lib "ODBC32.DLL" (ByVal henv&, ByVal fDirection%, ByVal pucDSN$, ByVal sDSNMaxLen%, psDSNLen%, ByVal pucDesc$, ByVal sDescMaxLen%, psDescLen%) As Integer

CODE:

```
Global szOut1 As String * 512
Global cbOut1 As Integer
Global szOut2 As String * 512
Global cbOut2 As Integer
Global xarray1(50) As Variant

Private Sub mdr1data_Click()
     retcode = SQLDataSources(hEnv&, SQL_FETCH_FIRST, szOut1, 512, cbOut1,
szOut2, 512, cbOut2)
          errorcheck retcode
     xarray1(1) = Chop(szOut1)
     i = 2
     Do While retcode = 0
          retcode = SQLDataSources(hEnv&, SQL_FETCH_NEXT, szOut1, 512,
cbOut1, szOut2, 512, cbOut2)
          If retcode <> 100 Then
               errorcheck retcode
               xarray1(i) = Chop(szOut1)
               If i < 50 Then
                    i = i + 1
               End If
           End If
     Loop
     view2.List1.Clear
     i = 1
     n = 1
     Do While n <> 0
          view2.List1.AddItem xarray1(i)
          i = i + 1
          n = Len(xarray1(i))
     Loop
     retcode = SQLFreeStmt(hStmt&, SQL_CLOSE)
End Sub
```

4.16 SQLDescribeCol

SQLDescribeCol returns the column name, type, length, and scale for one column in the result set.

Syntax

`RETCODE = SQLDescribeCol(hstmt, icol, szColName, cbColNameMax, pcbColName, pfSqlType, pcbColDef, pibScale, pfNullable)`

Arguments

Type	Argument	Use	Description
Long	hstmt	Input	The statement handle.
Integer	icol	Input	The column number in the result data, starting at 1.
String	szColName	Output	The buffer containing the column name.
Integer	cbColNameMax	Input	The maximum length of the column-name buffer.
Integer	pcbColName	Output	The number of bytes placed in the column-name buffer.
Integer	pfSqlType	Output	The SQL data type of the column.
Long	pcbColDef	Output	The length or precision for this column's data type.
Integer	pibScale	Output	The scale for this column's data type, if applicable.
Integer	pfNullable	Output	Returns one of the following: SQL_PERMIT_NULLS when the column allows null values, SQL_NO_NULLS when NULLS are not allowed, or SQL_NULLABLE_UNKNOWN when unknown.

Return Values

`SQL_SUCCESS, SQL_ERROR, SQL_SUCCESS_WITH_INFO,` or `SQL_INVALID_HANDLE`

Comments

The SQL data type of the column is pfSqlType and is one of the defined SQL data types. The length of the column definition or column precision is described by pcbColDef. Length information differs depending on the class of the data type, as follows:

pfSqlType	pcbColDef parameter contents
SQL_CHAR, SQL_VARCHAR	Maximum length
SQL_DECIMAL	Precision (maximum number of digits possible)
SQL_DOUBLE, SQL_FLOAT	Precision
SQL_INTEGER	Precision
SQL_NUMERIC	Precision
SQL_REAL	Precision
SQL_SMALLINT	Precision

pibScale is the total number of digits to the right of the decimal point for the column referenced. pibScale is defined only for the SQL_DECIMAL and SQL_NUMERIC data types. For example, if the Oterro database data type of the column is NUMERIC(5,3), pcbColDef would contain the value five and pibScale would contain the value three. Zero is returned where pibScale is either zero or undefined.

Due to the possible presence of any RULES that are created in an Oterro database as part of the column definition, pfNullable is always SQL_NULLABLE_UNKNOWN.

Related Functions

Function	Description
SQLColAttributes	Returns column attributes in a result set.
SQLExtendedFetch	Fetches one row of a result set; allows scrolling.
SQLFetch	Fetches one row of a result set.
SQLGetData	Gets result data for a column in a result set.
SQLGetTypeInfo	Returns information about the data types in the database.
SQLNumResultCols	Returns the number of columns in a result set.
SQLTables	Returns the tables in a database.

Errors

SQLSTATE	Description
01004	The data was truncated (SQL_SUCCESS_WITH_INFO was returned).
24000	An invalid cursor state: A cursor is currently open on the statement handle.
S1000	An error has occurred that has no defined SQLSTATE—see the error message text.
S1001	A memory allocation failure.
S1002	An invalid column number.
S1090	An invalid string or buffer length.

Visual Basic Example

SQLAPI.BAS:
Declare Function SQLDescribeCol Lib "ODBC32.DLL" (ByVal hstmt&, ByVal icol%, ByVal szColName$, ByVal cbColNameMax%, pcbColName%, pfSqlType%, pcbColDef&, pibScale%, pfNullable%) As Integer

CODE:
```
Global xarray1(50) As Variant
Global sqlstring As String

Private Sub ms2descr_Click()
     Dim i As Integer
     Dim n As Integer
     Dim szcol11 As String * 18
     Dim cbcol11 As Integer
     Dim cbcol12 As Integer
     Dim cblong1 As Long
     Dim cbcol13 As Integer
     Dim cbcol14 As Integer
     sqlstring = "select * from texts" & vbNullChar
     retcode = SQLExecDirect(hstmt&, sqlstring, SQL_NTS)
          errorcheck retcode
     retcode = SQLNumResultCols(hStmt&, colnum)
          errorcheck retcode
     i = 1
     Do While i <= colnum
          retcode = SQLDescribeCol(hStmt&, i, szcol11, 18, cbcol11,
cbcol12, cblong1, cbcol13, cbcol14)
               errorcheck retcode
          xarray1(i) = Chop(szcol11)
          retcode = SQLColAttributes(hStmt&, i, SQL_COLUMN_DISPLAY_SIZE,
szcol11, 18, cbcol11, cblong1)
               errorcheck retcode
          xarray1(i) = xarray1(i) & "; " & cblong1
          retcode = SQLColAttributes(hStmt&, i, SQL_COLUMN_SEARCHABLE,
szcol11, 18, cbcol11, cblong1)
               errorcheck retcode
          xarray1(i) = xarray1(i) & "; " & cblong1
          i = i + 1
     Loop
     view2.List1.Clear
     i = 1
     n = 1
     Do While n <> 0
          view2.List1.AddItem xarray1(i)
          i = i + 1
          n = Len(xarray1(i))
     Loop
     retcode = SQLFreeStmt(hStmt&, SQL_CLOSE)
End Sub
```

4.17 SQLDescribeParam

Note: This function is normally used in conjunction with SQLBindParameter. Since Visual Basic does not support SQLBindParameter, this function is included here with the syntax for using the C or C++ programming language.

SQLDescribeParam returns the description of a parameter marker associated with a prepared SQL statement.

Syntax

RETCODE PASCAL SQLDescribeParam (hStmt, ipar, pfSqlType, pcbColDef, pibScale, pfNullable)

Arguments

Type	Argument	Use	Description
HSTMT	hStmt	Input	The statement handle.
UWORD	ipar	Input	The parameter to describe, starting at 1.
SWORD FAR*	pfSqlType	Output	The SQL data type of the parameter.
UDWORD FAR*	pcbColDef	Output	The length or precision for this column's data.
SWORD FAR*	pibScale	Output	The scale for this column's data type.
SWORD FAR*	pfNullable	Output	Returns one of the following: SQL_NO_NULLS when the parameter does not allow NULL values, SQL_NULLABLE when NULL values are allowed, SQL_NULLABLE_UNKNOWN when unknown if the parameter allows NULL values.

Return Values

SQL_SUCCESS, SQL_SUCCESS_ WITH_INFO, SQL_STILL_EXECUTING, SQL_ERROR, or SQL_INVALID_HANDLE.

Comments

Parameter markers are numbered from left to right in the order they appear in the SQL statement.

SQLDescribeParam does not return the type of a parameter in an SQL statement. All parameters are input except in calls to procedures. Call SQLProcedureColumns to determine the type of each parameter in a call to a procedure.

4.18 SQLDisconnect

SQLDisconnect closes the database connection associated with a specific connection handle.

Syntax

```
RETCODE = SQLDisconnect (hdbc)
```

Arguments

Type	Argument	Use	Description
Long	hdbc	Input	The database connection handle.

Return Values

SQL_SUCCESS, SQL_SUCCESS_WITH_INFO, SQL_ERROR, or SQL_INVALID_HANDLE

Comments

When this function is called and there are uncommitted transactions, SQL_ERROR is returned. The connection to the database cannot be removed until SQLTransact has been called to commit or rollback the transaction(s).

When there are no outstanding transactions, any statement handles that have been allocated are released in a manner equivalent to calling SQLFreeStmt with the SQL_DROP option.

Related Functions

Function	Description
SQLAllocConnect	Allocates a connection handle.
SQLBrowseConnect	Retrieves values required to connect to a data source.
SQLConnect	Opens a connection to a database.
SQLDriverConnect	Prompts for information to open a connection to a database.
SQLFreeConnect	Frees the connection handle.

Errors

SQLSTATE	Description
08003	No database has been connected.

25000	A commit or rollback must be executed before disconnecting from this database.
S1000	An error has occurred that has no defined SQLSTATE—see the error message text.
S1001	A memory allocation failure.

Visual Basic Example

SQLAPI.BAS:
Declare Function SQLDisconnect Lib "ODBC32.DLL" (ByVal hdbc&) As Integer

CODE:
```
Global Const xdisco = "No Database Connected"
Global Const xexit = "end"
Global dbstr As String

quit xdisco

Sub quit(xend As String)    'xend specifies whether to disconnect or exit
     retcode = SQLFreeStmt(hstmt&, SQL_DROP)
     retcode = SQLDisconnect(hdbc&)
     retcode = SQLFreeConnect(hdbc&)
     If hdbc2 <> 0 Then
          retcode = SQLDisconnect(hdbc2&)
          retcode = SQLFreeConnect(hdbc2&)
     End If
     dbstr = xdisco
     If xend = "end" Then
          retcode = SQLFreeEnv (henv&)
          End
     End If
End Sub
```

4.19 SQLDriverConnect

SQLDriverConnect connects to a database through the ODBC Driver Manager. SQLDriverConnect can display an ODBC data source dialog box in which the currently defined data sources are listed.

Syntax

RETCODE = SQLDriverConnect(hdbc, hwnd, szConnStrIn, cbConnStrIn, szConnStrOut, cbConnStrOutMax, pcbConnStrOut, fDriverCompletion)

Arguments

Type	Argument	Use	Description
Long	hdbc	Input	The database connection handle.
Long	hwnd	Input	The window handle of the calling process.
String	szConnStrIn	Input	The connection string.
Integer	cbConnStrIn	Input	The length of the connection string.
String	szConnStrOut	Output	A pointer to the filled connection string.

Integer	cbConnStrOutMax	Input	The maximum length of the output connection string.
Integer	pcbConnStrOut	Output	The length of the returned connection string.
integer	fDriverCompletion	Input	How the driver is supposed to treat the connection.

Return Values

`SQL_SUCCESS, SQL_SUCCESS_WITH_INFO, SQL_ERROR,` or `SQL_INVALID_HANDLE`

Information Types

fDriverCompletion Constant	Value	Description
SQL_DRIVER_NOPROMPT	0	Attempt the connection with the given information.
SQL_DRIVER_COMPLETE	1	If all required information is not available, prompt for the information.
SQL_DRIVER_COMPLETE_REQUIRED	2	If the connection cannot be made with the information given, prompt for the information.
SQL_DRIVER_PROMPT	3	Prompt regardless of what information is given.

When fDriverCompletion is SQL_DRIVER_COMPLETE_REQUIRED, the Oterro Engine does not prompt for any missing or incorrect parameters because the Data Source Name (DSN) is produced by the ODBC driver manager.

Comments

By default, transaction processing is set ON and AUTOCOMMIT is set ON for this function. You can override these defaults by calling the function SQLSetConnectOption with the appropriate arguments, or by setting AUTOCOMMIT OFF or TRANSACT OFF in the OTERRO11.CFG file.

When the ODBC driver manager is used, a dialog box is presented to select a data source, user identifier, and password, if necessary.

When the user identifier is already known, you might have a connection string as follows: "DSN=DATABASE;UID=USERID;PWD=PASSWORD".

When SQLDriverConnect is called without releasing the connection handle, the connection options that were available in the previous connection are available in the new one.

This function requires a window handle. Therefore, it must be called from within a Visual Basic form or the window handle must be specifically passed to the function.

Related Functions

Function	Description
SQLAllocConnect	Allocates a connection handle.
SQLBrowseConnect	Retrieves values required to connect to a data source.
SQLConnect	Opens a connection to a database.
SQLDataSources	Returns the data source names.
SQLDisconnect	Closes the connection to a database.
SQLDrivers	Returns the driver descriptions.
SQLFreeConnect	Frees the connection handle.
SQLGetConnectOption	Queries the status of a connection option.
SQLSetConnectOption	Sets a database connection option.

Errors

SQLSTATE	Description
01004	The data was truncated (SQL_SUCCESS_WITH_INFO was returned).
01S00	An unknown connection attribute was specified. The connection was completed successfully.
08001	Unable to connect to the data source.
08002	The connection is already in use.
08004	The data source rejected the connection.
28000	No access is available for this user.
S1000	An error has occurred that has no defined SQLSTATE—see the error message text.
S1001	A memory allocation failure.
S1090	An invalid string or buffer length.
S1110	An invalid value for fDriverCompletion.

Visual Basic Example

SQLAPI.BAS:
Declare Function SQLDriverConnect Lib "ODBC32.DLL" (ByVal hdbc&, ByVal hwnd&, ByVal szConnStrIn$, ByVal cbConnStrIn%, ByVal szConnStrOut$, ByVal cbConnStrOutMax%, pcbConnStrOut%, ByVal fDriverCompletion%) As Integer

CODE:
```
Global szConnectOut As String * 512
Global cbConnectOut As Integer
Global dbstr As String
Global dbdir As String

Private Sub fdrvconn_Click()
     Dim i As Integer
     retcode = SQLAllocEnv(henv&)
     retcode = SQLAllocConnect(henv&, hdbc&)
          errorcheck retcode
     retcode = SQLDriverConnect(hdbc&, hwnd&, dbstr, SQL_NTS, szConnectOut,
255, cbConnectOut, SQL_DRIVER_COMPLETE)
     If retcode <> 0 Then
          errorcheck retcode
          GoTo lend
     End If
     'get the database path
     retcode = SQLGetInfo(hdbc&, SQL_DATABASE_NAME, szConnectOut, 512,
cbConnectOut)
          errorcheck retcode
     dbstr = Chop(szConnectOut)
     i = InStr(dbstr, "\")
     dbdir = Left$(dbstr, i)
     Do While i <> 0
          i = InStr(i + 1, dbstr, "\")
          If i <> 0 Then
               dbdir = Left$(dbstr, i)
          End If
     Loop
     retcode = SQLAllocStmt(hdbc&, hstmt&)
          errorcheck retcode
lend:
```

```
End Sub
```

4.20 SQLDrivers

SQLDrivers lists driver descriptions and driver attribute keywords. This function is implemented by the ODBC Driver Manager and returns information only on drivers currently listed by the ODBC Administrator.

Syntax

RETCODE = SQLDrivers(hEnv, fDirection, pucDesc, sDescMaxLen, psDescLen, pucAttribs, sAttribsMaxLen, psAttribsLen)

Arguments

Type	Argument	Use	Description
Long	hEnv	Input	The environment handle.
Integer	fDirection	Input	Determines which driver description is fetched: SQL_FETCH_NEXT, SQL_FETCH_FIRST
String	pucDesc	Output	Pointer to storage for the driver description.
Integer	sDescMaxLen	Input	The maximum length of the pucDesc buffer.
Integer	sDescLen	Output	Total number of bytes available to return pucDesc.
String	pucAttribs	Output	Pointer to storage for the list of driver attribute value pairs.
Integer	sAttribsMaxLen	Input	Maximum length of the pucAttribs buffer.
Integer	psAttribsLen	Output	Total number of bytes available to return in pucAttribs.

Return Values

SQL_SUCCESS, SQL_SUCCESS_WITH_INFO, SQL_NO_DATA_FOUND, SQL_ERROR, or SQL_INVALID_HANDLE.

Comments

SQLDrivers returns the driver description in the pucDesc argument. It returns additional information about the driver in the pucAttribs argument as a list of keyword-value pairs.

Because SQLDrivers is implemented by the ODBC Driver Manager, it is supported for all drivers.

Related Functions

Function	Description
SQLBrowseConnect	Retrieves values required to connect to a data source.
SQLConnect	Opens a connection to a database.
SQLDataSources	Returns the data source names.
SQLDriverConnect	Prompts for information to open a connection to a database.

Errors

SQLSTATE	Description
01000	Driver Manager-specific informational message (Function returns SQL_SUCCESS_WITH_INFO).

01004	The data was truncated (SQL_SUCCESS_WITH_INFO was returned).
S1000	An error has occurred that has no defined SQLSTATE—see the error message text.
S1001	A memory allocation failure.
S1090	An invalid string or buffer length.
S1103	The direction option was out of range.

Visual Basic Example

SQLAPI.BAS:
Declare Function SQLDrivers Lib "ODBC32.DLL" (ByVal henv&, ByVal fDirection%, ByVal pucDesc$, ByVal sDescMaxLen%, psDescLen%, ByVal pucAttribs$, ByVal sAttribsMaxLen%, psAttribsLen%) As Integer

CODE:
```
Global szOut1 As String * 512
Global cbOut1 As Integer
Global szOut2 As String * 512
Global cbOut2 As Integer
Global xarray1(50) As Variant

Private Sub mdr1drv_Click()
     retcode = SQLDrivers(hEnv&, SQL_FETCH_FIRST, szOut1, 512, cbOut1,
szOut2, 512, cbOut2)
          errorcheck retcode
     xarray1(1) = Chop(szOut1)
     i = 2
     Do While retcode = 0
          retcode = SQLDrivers(hEnv&, SQL_FETCH_NEXT, szOut1, 512, cbOut1,
szOut2, 512, cbOut2)
          If retcode <> 100 Then
               errorcheck retcode
               xarray1(i) = Chop(szOut1)
               If i < 50 Then
                    i = i + 1
               End If
          End If
     Loop
     view2.List1.Clear
     i = 1
     n = 1
     Do While n <> 0
          view2.List1.AddItem xarray1(i)
          i = i + 1
          n = Len(xarray1(i))
     Loop
     retcode = SQLFreeStmt(hStmt&, SQL_CLOSE)
End Sub
```

4.21 SQLEndTran

SQLEndTran requests a commit or rollback operation for all active operations on all statements associated with a connection. SQLEndTran can also request that a commit or rollback operation be performed for all connections associated with an environment.

Syntax

```
RETCODE = SQLEndTran(HandleType, Handle, CompletionType)
```

Argumants

Type	Argument	Use	Description
Integer	HandleType	Iput	Handle type identifier. Contains either SQL_HANDLE_ENV (if Handle is an environment handle) or SQL_HANDLE_DBC (if Handle is a connection handle).
Long	Handle	Input	The handle, of the type indicated by HandleType, indicating the scope of the transaction. See "Comments" for more information.
Integer	CompletionType	Input	One of the following two values: SQL_COMMIT SQL_ROLLBACK

Return Values

SQL_SUCCESS, SQL_SUCCESS_WITH_INFO, SQL_ERROR, or SQL_INVALID_HANDLE

Errors

SQLSTATE	Error	Description
01000	General warning	Driver-specific informational message. (Function returns SQL_SUCCESS_WITH_INFO.)
08003	Connection not open	(DM) The HandleType was SQL_HANDLE_DBC, and the Handle was not in a connected state.
08007	Connection failure during transaction	The HandleType was SQL_HANDLE_DBC, and the connection associated with the Handle failed during the execution of the function, and it cannot be determined whether the requested COMMIT or ROLLBACK occurred before the failure.
25S01	Transaction state unknown	One or more of the connections in Handle failed to complete the transaction with the outcome specified, and the outcome is unknown.
25S02	Transaction is still active	The driver was not able to guarantee that all work in the global transaction could be completed atomically, and the transaction is still active.
25S03	Transaction is rolled back	The driver was not able to guarantee that all work in the global transaction could be completed atomically, and all work in the transaction active in Handle was rolled back.
40001	Serialization failure	The transaction was rolled back due to a resource deadlock with another transaction.
40002	Integrity constraint violation	The CompletionType was SQL_COMMIT, and the commitment of changes caused integrity constraint violation. As a result, the transaction was rolled back.
HY000	General error	An error occurred for which there was no specific SQLSTATE and for which no implementation-specific SQLSTATE was defined. The error message returned by SQLGetDiagRec in the *szMessageText buffer describes the error and its cause.

HY001	Memory allocation error	The driver was unable to allocate memory required to support execution or completion of the function.
HY010	Function sequence error	(DM) An asynchronously executing function was called for a StatementHandle associated with the ConnectionHandle and was still executing when SQLEndTran was called. (DM) SQLExecute, SQLExecDirect, SQLBulkOperations, or SQLSetPos was called for a StatementHandle associated with the ConnectionHandle and returned SQL_NEED_DATA. This function was called before data was sent for all data-at-execution parameters or columns.
HY012	Invalid transaction operation code	(DM) The value specified for the argument CompletionType was neither SQL_COMMIT nor SQL_ROLLBACK.
HY013	Memory management error	The function call could not be processed because the underlying memory objects could not be accessed, possibly because of low memory conditions.
HY092	Invalid attribute/option identifier	(DM) The value specified for the argument HandleType was neither SQL_HANDLE_ENV nor SQL_HANDLE_DBC.
HYC00	Optional feature not implemented	The driver or data source does not support the ROLLBACK operation.
HYT01	Connection timeout expired	The connection timeout period expired before the data source responded to the request. The connection timeout period is set through SQLSetConnectAttr, SQL_ATTR_CONNECTION_TIMEOUT.
IM001	Driver does not support this function	(DM) The driver associated with the ConnectionHandle does not support the function.

Comments

For an ODBC 3.x driver, if HandleType is SQL_HANDLE_ENV and Handle is a valid environment handle, then the Driver Manager will call SQLEndTran in each driver associated with the environment. The Handle argument for the call to a driver will be the driver's environment handle. For an ODBC 2.x driver, if HandleType is SQL_HANDLE_ENV and Handle is a valid environment handle, and there are multiple connections in a connected state in that environment, then the Driver Manager will call SQLTransact in the driver once for each connection in a connected state in that environment. The Handle argument in each call will be the connection's handle. In either case, the driver will attempt to commit or roll back transactions, depending on the value of CompletionType, on all connections that are in a connected state on that environment. Connections that are not active do not affect the transaction.

Note: SQLEndTran cannot be used to commit or roll back transactions on a shared environment. SQLSTATE HY092 (Invalid attribute/option identifier) will be returned if SQLEndTran is called with Handle set to either the handle of a shared environment or the handle of a connection on a shared environment.

The Driver Manager will return SQL_SUCCESS only if it receives SQL_SUCCESS for each connection. If the Driver Manager receives SQL_ERROR on one or more connections, it returns SQL_ERROR to the application, and the diagnostic information is placed in the diagnostic data structure of the environment. To determine which connection or connections failed during the commit or rollback operation, the application can call SQLGetDiagRec for each connection.

Related Functions

Function	Description
SQLGetInfo	Returning information about a driver or data source
SQLFreeHandle	Freeing a handle
SQLFreeStmt	Freeing a statement handle

4.22 SQLError

Error information is associated with the environment handle, connection handle, and statement handle. Error information is retrieved from SQLError one handle at a time.

Syntax

RETCODE = SQLError(henv, hdbc, hstmt, szSqlState, pfNativeError, szErrorMsg, cbErrorMsgMax, pcbErrorMsg)

Arguments

Type	Argument	Use	Description
Long	henv	Input	The environment handle.
Long	hdbc	Input	The database connection handle.
Long	hstmt	Input	The statement handle.
String	szSqlState	Output	The SQL state string.
Long	pfNativeError	Output	The Oterro error code.
String	szErrorMsg	Output	The Oterro error message text.
Integer	cbErrorMsgMax	Input	The maximum length of the error message text buffer.
Integer	pcbErrorMsg	Output	The actual length of the text.

Return Values

SQL_SUCCESS, SQL_SUCCESS_WITH_INFO, SQL_ERROR, SQL_INVALID_HANDLE, or SQL_NO_DATA_FOUND

Comments

Once error information for a certain function call has been retrieved by SQLError, it cannot be read again. If a function call returns multiple errors, SQLError must be called multiple times.

SQL_NULL_HDBC and SQL_NULL_HSTMT need to be used for hdbc and hstmt when an error has occurred during a function call that requires a valid handle for either hdbc or hstmt, such as SQLAllocConnet, SQLAllocStmt, SQLBrowseConnect, SQLConnect, SQLDisconnect, or SQLDriverConnect.

Errors

SQLError does not post any extended error information.

Visual Basic Example

SQLAPI.BAS:
Declare Function SQLError Lib "ODBC32.DLL" (ByVal henv&, ByVal hdbc&, ByVal hstmt&, ByVal szsqlstate$, pfnativeerror&, ByVal szErrorMsg$, ByVal cbErrorMsgMax%, pcberrormsg%) As Integer

CODE:
```
Global retcode As Integer
Global dbstr As String
Global szsqlstate As String
Global pfnativeerror As String
Global aucerrortext As String

dbstr = "select * from artist"
retcode = SQLExecDirect(hStmt&, dbstr, SQL_NTS)
     errorcheck retcode

Sub errorcheck(retcode As Integer)
     Dim errorval As Integer
     Dim errormsg As String
     errorval = retcode
     If errorval <> SQL_SUCCESS Then
          Select Case errorval
               Case SQL_INVALID_HANDLE
                    MsgBox "Invalid Handle", 0, "API Error"
               Case SQL_SUCCESS_WITH_INFO
                    retcode = SQLError(henv&, hdbc&, hstmt&, szsqlstate,
pfnativeerror, aucerrortext, 512, 0)
                    errormsg = "SQLState: " & Chop(szsqlstate) & Chr$(13) &
"NativeError: " & pfnativeerror & Chr$(13) & Chop(aucerrortext)
                    MsgBox errormsg, 0, "API Error"
               Case Else
                    retcode = SQLError(henv&, hdbc&, hstmt&, szsqlstate,
pfnativeerror, aucerrortext, 512, 0)
                    errormsg = "SQLState: " & Chop(szsqlstate) & Chr$(13) &
"NativeError: " & pfnativeerror & Chr$(13) & Chop(aucerrortext)
                    MsgBox errormsg, 0, "API Error"
          End Select
     End If
End Sub
```

4.23 SQLExecDirect

SQLExecDirect executes the statement in the szSqlStr buffer by sending the statement to the DBMS for processing.

Syntax

RETCODE = SQLExecDirect (hstmt, szSqlStr, cbSqlStr)

Arguments

Type	Argument	Use	Description
Long	hstmt	Input	The statement handle.
String	szSqlStr	Input	The SQL statement.
Long	cbSqlStr	Input	The length of the SQL statement.

Return Values

SQL_SUCCESS, SQL_SUCCESS_WITH_INFO, SQL_ERROR, SQL_INVALID_HANDLE, or SQL_NEED_DATA (when parameter markers are used)

Comments

SQLExecDirect is the basic method by which data is retrieved and modified. SQL commands are passed to the Oterro Engine by the SQLExecDirect function.

The statement in the szSqlStr buffer can be any SQL command that the Oterro database supports; those commands are listed in "How to Use the Oterro Engine" in Chapter Two.

When the statement in the szSqlStr buffer contains the Oterro database command SELECT, a cursor name is generated by the Oterro Engine. When the function SQLSetCursorName is used to associate a cursor name with an hstmt, that cursor name is used instead of the cursor name generated by the Oterro Engine. To retrieve the cursor name generated by the Oterro Engine, use the function SQLGetCursorName.

When a prepared statement exists on the statement handle, the statement is released and all associated memory is released.

While parameter markers are allowed within calls to SQLExecDirect, they slow down the process. Use parameter markers with the functions SQLPrepare and SQLExecute instead of with SQLExecDirect.

Related Functions

Function	Description
SQLAllocStmt	Allocates a new statement handle.
SQLCancel	Ends processing on a statement.
SQLExecute	Executes a prepared SQL statement.
SQLExtendedFetch	Fetches one row of a result set; allows scrolling.
SQLFetch	Fetches one row of a result set.
SQLGetCursorName	Gets the name of the cursor associated with a statement handle.
SQLGetData	Gets result data for a column in a result set.
SQLPrepare	Prepares an SQL statement for execution.
SQLRowCount	Returns the number of rows affected by the update, insert, or delete.
SQLSetCursorName	Sets a cursor name for a statement handle.

Errors

SQLSTATE	Description
01006	The privilege was not revoked: The user did not have the permission to revoke.
07001	The wrong number of parameters.
21001	The string parameter was truncated.
21003	The numeric parameter was out of range.
21S01	The number of values in the insert list did not match the table.
21S02	The number of columns listed in the view did not match the actual number.
22008	An invalid date, time, or timestamp parameter.
22012	Division by zero has occurred.
23000	An integrity constraint violation: A rule has been violated.
24000	An invalid cursor state: A cursor is currently open on the statement handle.
34000	An invalid cursor name: The cursor specified in the statement does not exist.
37000	A syntax error or access violation: An incorrect SQL statement.
40001	A serialization failure: The entire transaction has been rolled back to prevent deadlock.
42000	A syntax error or access violation: Did not have sufficient permissions to execute.
S0001	A table or view already exists.
S0002	A table or view does not exist.
S0011	An index already exists.
S0012	An index does not exist.
S0021	A column already exists.
S0022	A column does not exist.
S1000	An error has occurred that has no defined SQLSTATE—see the error message text.

S1001	A memory allocation failure.
S1009	An invalid argument value—a null pointer was passed.
S1090	An invalid string or buffer length.

Visual Basic Example

SQLAPI.BAS:
Declare Function SQLExecDirect Lib "ODBC32.DLL" (ByVal hstmt&, ByVal szSqlStr$, ByVal cbSqlStr&) As Integer

CODE:
```
Global sqlstring As String

sqlstring = "select * from artists"
execdir sqlstring

Sub execdir(sqlstring As String)
     retcode = SQLExecDirect(hstmt&, sqlstring, SQL_NTS)
          errorcheck retcode
End Sub
```

4.24 SQLExecute

SQLExecute executes the prepared SQL statement associated with hstmt.

Syntax

RETCODE = SQLExecute (hstmt)

Arguments

Type	Argument	Use	Description
Long	hstmt	Input	The statement handle.

Return Values

SQL_SUCCESS, SQL_SUCCESS_WITH_INFO, SQL_ERROR, SQL_INVALID_HANDLE, SQL_NEED_DATA (when parameter markers are used)

Comments

When the statement contains the Oterro command SELECT, a cursor name is generated by the Oterro Engine. When the function SQLSetCursorName is used to associate a cursor name with an hstmt, that cursor name is used instead of the cursor name generated by the Oterro Engine. To retrieve the cursor name generated by the Oterro Engine, use the function SQLGetCursorName.

Related Functions

Function	Description
SQLAllocStmt	Allocates a new statement handle.
SQLCancel	Ends processing on a statement.

SQLExecDirect	Executes an SQL statement.
SQLExtendedFetch	Fetches one row of a result set; allows scrolling.
SQLFetch	Fetches one row of a result set.
SQLGetCursorName	Gets the name of the cursor associated with a statement handle.
SQLGetData	Gets result data for a column in a result set.
SQLPrepare	Prepares an SQL statement for execution.
SQLRowCount	Returns the number of rows affected by the update, insert, or delete.
SQLSetCursorName	Sets a cursor name for a statement handle.

Errors

SQLSTATE	Description
01006	The privilege was not revoked: The user did not have the permission to revoke.
07001	The wrong number of parameters.
21001	The string parameter was truncated.
21003	The numeric parameter was out of range.
21S01	The number of values in the insert list did not match the table.
21S02	The number of columns listed in the view did not match the actual number.
22008	An invalid date, time, or timestamp parameter.
22012	Division by zero has occurred.
23000	An integrity constraint violation: A rule has been violated.
24000	An invalid cursor state: A cursor is currently open on the statement handle.
34000	An invalid cursor name: The cursor specified in the statement does not exist.
37000	A syntax error or access violation: An incorrect SQL statement.
40001	A serialization failure: The entire transaction has been rolled back to prevent deadlock.
42000	A syntax error or access violation: Did not have sufficient permissions to execute.
S0001	A table or view already exists.
S0002	A table or view does not exist.
S0011	An index already exists.
S0012	An index does not exist.
S0021	A column already exists.
S0022	A column does not exist.
S1000	An error has occurred that has no defined SQLSTATE—see the error message text.
S1001	A memory allocation failure.
S1009	An invalid argument value—a null pointer was passed.
S1010	The statement was not prepared.
S1090	An invalid string or buffer length.

Visual Basic Example

SQLAPI.BAS:
Declare Function SQLExecute Lib "ODBC32.DLL" (ByVal hstmt&) As Integer

CODE:
```
Global sqlstring As String

Private Sub ms2extfetch_Click()
     Dim cblong1 As Long
     Dim cbint1 As Integer
     Dim i As Integer
     Dim n As Integer
     sqlstring = "select int1,real1,doub1 from numbers" & vbNullChar
     retcode = SQLPrepare(hStmt&, sqlstring, SQL_NTS)
          errorcheck retcode
     retcode = SQLExecute(hStmt&)
          errorcheck retcode
     retcode = SQLSetStmtOption(hStmt&, SQL_CURSOR_TYPE,
SQL_CURSOR_DYNAMIC)
     retcode = SQLGetStmtOption(hStmt&, SQL_CURSOR_TYPE, cblong1)
     xarray1(1) = "cursor type = " & cblong1
     retcode = SQLExtendedFetch(hStmt&, SQL_FETCH_FIRST, 1, cblong1,
cbint1)
     retcode = SQLGetData(hStmt&, 1, SQL_C_CHAR, colresults, 5000,
cbcolresults)
     xarray1(2) = Chop(colresults)
     retcode = SQLExtendedFetch(hStmt&, SQL_FETCH_LAST, 1, cblong1, cbint1)
     retcode = SQLGetData(hStmt&, 1, SQL_C_CHAR, colresults, 5000,
cbcolresults)
     xarray1(3) = Chop(colresults)
     retcode = SQLExtendedFetch(hStmt&, SQL_FETCH_PRIOR, 1, cblong1,
cbint1)
     retcode = SQLGetData(hStmt&, 1, SQL_C_CHAR, colresults, 5000,
cbcolresults)
     xarray1(4) = Chop(colresults)
     view2.List1.Clear
     i = 1
     n = 1
     Do While n <> 0
          view2.List1.AddItem xarray1(i)
          i = i + 1
          n = Len(xarray1(i))
     Loop
     retcode = SQLFreeStmt(hStmt&, SQL_CLOSE)
End Sub
```

4.25 SQLExtendedFetch

SQLExtendedFetch returns both a set of data as a row and scrolls through the result set according to the current scroll arguments.

Syntax

RETCODE = SQLExtendedFetch (hstmt, fFetchType, irow, pcrow, rfgRowStatus)

Arguments

Type	Argument	Use	Description
Long	hstmt	Input	The statement handle.
Integer	fFetchType	Input	The type of fetch; accepts one of the following: SQL_FETCH_NEXT SQL_FETCH_FIRST SQL_FETCH_LAST SQL_FETCH_PRIOR SQL_FETCH_ABSOLUTE SQL_FETCH_RELATIVE SQL_FETCH_BOOKMARK
Long	irow	Input	The number of the row to fetch.
Long	pcrow	Output	The number of rows actually fetched. When an error occurs during processing, pcrow points to the row that precedes the row with the error.
Integer	rfgRowStatus	Output	An array of status values. The number of elements must equal the number of rows in the rowset (as defined by the SQL_ROWSET_SIZE statement option). The driver returns a status value for each row fetched, which can be one of

			the following: SQL_ROW_SUCCESS, SQL_ROW_DELETED, or SQL_ROW_UPDATED. When the number of rows fetched is less than the number of elements in the status array, the driver sets remaining status elements to SQL_ROW_NOROW.

Return Values

```
SQL_SUCCESS, SQL_ERROR, SQL_MAX_ROWS, or SQL_INVALID_HANDLE
```

Comments

The SQLExtendedFetch function uses scrollable cursors only when the statement has been defined as a scrollable cursor using SQLSetStmtOption. When SQLSetStmtOption is not used to define scrollable cursors, SQLExtendedFetch behaves like SQLFetch. SQLFetch can use scrollable cursors but only fetches in the direction SQL_FETCH_NEXT.

Related Functions

Function	Description
SQLCancel	Ends processing on a statement.
SQLColumnPrivileges	Returns the privileges assigned to the columns of a table.
SQLDescribeCol	Describes a column in a result set.
SQLExecDirect	Executes an SQL statement.
SQLExecute	Executes a prepared SQL statement.
SQLFetch	Fetches one row of a result set.
SQLGetStmtOption	Queries the status of a statement option.
SQLNumResultCols	Returns the number of columns in a result set.
SQLProcedureColumns	Returns the columns for the procedures.
SQLProcedures	Returns the list of procedure names in the database.
SQLSetStmtOption	Sets options for a statement handle.
SQLSpecialColumns	Returns information about a set of columns.
SQLStatistics	Returns statistics for tables and indexes.
SQLTablePrivileges	Returns the privileges assigned to the table.
SQLTables	Returns the tables in a database.

Errors

SQLSTATE	Description
01000	A driver-specific informational message (The function returns SQL_SUCCESS_WITH_INFO).
01004	The data was truncated (SQL_SUCCESS_WITH_INFO was returned). String values are right truncated. For numeric values, the fractional part of number is truncated.
07006	The specified conversion is illegal.
08S01	The data source connection failed before the function completed processing.
22003	The numeric value is out of range: A significant truncation would have occurred.
22012	Division by zero has occurred.
24000	An invalid cursor state: A cursor is currently open on the statement handle.
40001	A serialization failure: The entire transaction has been rolled back to prevent deadlock.
IM001	The driver associated with the hstmt does not support the function.
S1000	An error has occurred that has no defined SQLSTATE—see the error message text.
S1001	A memory allocation failure.
S1002	An invalid column number.
S1106	The value specified for the argument fFetchType was not equal to: SQL_FETCH_NEXT SQL_FETCH_FIRST SQL_FETCH_LAST SQL_FETCH_PRIOR SQL_FETCH_ABSOLUTE SQL_FETCH_RELATIVE

S1107	The value specified with the SQL_CURSOR_TYPE statement option was SQL_CURSOR_KEYSET_DRIVEN, but the value specified with the SQL_KEYSET_SIZE statement option was greater than 0 and less than the value specified with the SQL_ROWSET_SIZE statement option.
S1C00	The driver or data source does not support the specified type.

Visual Basic Example

SQLAPI.BAS:
Declare Function SQLExtendedFetch Lib "ODBC32.DLL" (ByVal hstmt&, ByVal fFetchType%, ByVal irow&, pcrow&, rgfRowStatus%) As Integer

CODE:
```
Global sqlstring As String

Private Sub ms2extfetch_Click()
     Dim cblong1 As Long
     Dim cbint1 As Integer
     Dim i As Integer
     Dim n As Integer
     sqlstring = "select int1,real1,doub1 from numbers" & vbNullChar
     retcode = SQLPrepare(hStmt&, sqlstring, SQL_NTS)
          errorcheck retcode
     retcode = SQLExecute(hStmt&)
          errorcheck retcode
     retcode = SQLSetStmtOption(hStmt&, SQL_CURSOR_TYPE,
SQL_CURSOR_DYNAMIC)
     retcode = SQLGetStmtOption(hStmt&, SQL_CURSOR_TYPE, cblong1)
     xarray1(1) = "cursor type = " & cblong1
     retcode = SQLExtendedFetch(hStmt&, SQL_FETCH_FIRST, 1, cblong1,
cbint1)
     retcode = SQLGetData(hStmt&, 1, SQL_C_CHAR, colresults, 5000,
cbcolresults)
     xarray1(2) = Chop(colresults)
     retcode = SQLExtendedFetch(hStmt&, SQL_FETCH_LAST, 1, cblong1, cbint1)
     retcode = SQLGetData(hStmt&, 1, SQL_C_CHAR, colresults, 5000,
cbcolresults)
     xarray1(3) = Chop(colresults)
     retcode = SQLExtendedFetch(hStmt&, SQL_FETCH_PRIOR, 1, cblong1,
cbint1)
     retcode = SQLGetData(hStmt&, 1, SQL_C_CHAR, colresults, 5000,
cbcolresults)
     xarray1(4) = Chop(colresults)
     view2.List1.Clear
     i = 1
     n = 1
     Do While n <> 0
          view2.List1.AddItem xarray1(i)
          i = i + 1
          n = Len(xarray1(i))
     Loop
     retcode = SQLFreeStmt(hStmt&, SQL_CLOSE)
End Sub
```

4.26 SQLFetch

SQLFetch fetches the next row from the result set.

Syntax

```
RETCODE = SQLFetch (hstmt)
```

Arguments

Type	Argument	Use	Description
Long	hstmt	Input	The statement handle.

Return Values

```
SQL_SUCCESS, SQL_ERROR, SQL_INVALID_HANDLE, SQL_SUCCESS_WITH_INFO,
SQL_MAX_ROWS, or SQL_NO_DATA_FOUND
```

Comments

When designing an application using Visual Basic, after calling SQLFetch, you must use SQLGetData to retrieve the data from the result set for each column needed.

Related Functions

Function	Description
SQLColAttributes	Returns column attributes in a result set.
SQLColumnPrivileges	Returns the privileges assigned to the columns of a table.
SQLDescribeCol	Describes a column in a result set.
SQLExecDirect	Executes an SQL statement.
SQLExecute	Executes a prepared SQL statement.
SQLExtendedFetch	Fetches one row of a result set; allows scrolling.
SQLGetData	Gets result data for a column in a result set.
SQLNumResultCols	Returns the number of columns in a result set.
SQLProcedureColumns	Returns the columns for the procedures.
SQLProcedures	Returns the list of procedure names in the database.
SQLStatistics	Returns statistics for tables and indexes.
SQLTablePrivileges	Returns the privileges assigned to the table.
SQLTables	Returns the tables in a database.

Errors

SQLSTATE	Description
01004	The data was truncated (SQL_SUCCESS_WITH_INFO was returned).
07006	The specified conversion is illegal.
22003	The numeric value is out of range: A significant truncation would have occurred.
22012	Division by zero has occurred.
24000	An invalid cursor state: A cursor is currently open on the statement handle.
S1000	An error has occurred that has no defined SQLSTATE—see the error message text.
S1001	A memory allocation failure.

S1002	An invalid column number.

Visual Basic Example
SQLAPI.BAS:
Declare Function SQLFetch Lib "ODBC32.DLL" (ByVal hstmt&) As Integer

```
CODE:
Global colnum As Integer
Global bufstring As String
Global cbcol1 As Long
Global colresults As String * 5000
Global starttime As Date
Global endtime As Date
Global elapsed As Integer

Sub getall()
     Dim i As Integer
     retcode = SQLNumResultCols(hstmt, colnum)
          errorcheck retcode
     bufstring = sql1.Text
     results.AddItem UCase(bufstring)
     starttime = Time
     Do While SQLFetch(hstmt&) = SQL_SUCCESS
          i = 1
          Do While i <= colnum
               retcode = SQLGetData(hstmt&, i, SQL_C_CHAR, colresults,
5000, cbcol1)
                    errorcheck retcode
               If i = 1 Then
                    bufstring = Chop(colresults)
               Else
                    bufstring = bufstring & "," & Chop(colresults)
               End If
               i = i + 1
          Loop
          results.AddItem bufstring
     Loop
     endtime = Time
     elapsed = DateDiff("s", starttime, endtime)
     bufstring = "Elapsed Time:   " & elapsed & "   seconds"
     results.AddItem bufstring
     retcode = SQLFreeStmt(hstmt&, SQL_CLOSE)
End Sub
```

4.27 SQLFetchScroll

SQLFetchScroll fetches the specified rowset of data from the result set and returns data for all bound columns. Row sets can be specified at an absolute or relative position or by bookmark. SQLFreeHandle frees resources associated with a specific environment, connection, statement, or descriptor handle. This new function in Oterro 3.0 for freeing handles is in addition to the ODBC 2.0 functions SQLFreeConnect (for freeing a connection handle) and SQLFreeEnv (for freeing an environment handle). SQLFreeConnect and SQLFreeEnv are both deprecated in ODBC 3.x.

Syntax

```
RETCODE = SQLFetchScroll(StatementHandle, FetchOrientation, FetchOffset)
```

Arguments

Type	Argument	Use	Description
Long	StatementHandle	Input	Statement handle.
Integer	FetchOrientation	Input	Type of fetch: SQL_FETCH_NEXT SQL_FETCH_PRIOR SQL_FETCH_FIRST SQL_FETCH_LAST SQL_FETCH_ABSOLUTE SQL_FETCH_RELATIVE SQL_FETCH_BOOKMARK
Integer	FetchOffset	Input	Number of the row to fetch. The interpretation of this argument depends on the value of the FetchOrientation argument.

Return Values

```
SQL_SUCCESS, SQL_SUCCESS_WITH_INFO, SQL_NO_DATA, SQL_STILL_EXECUTING,
SQL_ERROR, or SQL_INVALID_HANDLE
```

Errors

When SQLFetchScroll returns either SQL_ERROR or SQL_SUCCESS_WITH_INFO, an associated SQLSTATE value can be obtained by calling SQLGetDiagRec with a HandleType of SQL_HANDLE_STMT and a Handle of StatementHandle. The following table lists the SQLSTATE values commonly returned by SQLFetchScroll and explains each one in the context of this function; the notation "(DM)" precedes the descriptions of SQLSTATEs returned by the Driver Manager. The return code associated with each SQLSTATE value is SQL_ERROR, unless noted otherwise.

For all those SQLSTATEs that can return SQL_SUCCESS_WITH_INFO or SQL_ERROR (except 01xxx SQLSTATEs), SQL_SUCCESS_WITH_INFO is returned if an error occurs on one or more, but not all, rows of a multirow operation, and SQL_ERROR is returned if an error occurs on a single-row operation.

SQLSTATE	Error	Description
01000	General warning	Driver-specific informational message. (Function returns SQL_SUCCESS_WITH_INFO.)
01004	String data, right truncated	String or binary data returned for a column resulted in the truncation of nonblank character or non-NULL binary data. If it was a string value, it was right-truncated.
01S01	Error in row	An error occurred while fetching one or more rows. (If this SQLSTATE is returned when an ODBC 3.x application is working with an ODBC 2.x driver, it can be ignored.)
01S06	Attempt to fetch before the result set returned the first rowset	The requested rowset overlapped the start of the result set when FetchOrientation was SQL_FETCH_PRIOR, the current position was beyond the first row, and the number of the current row is less than or equal to the rowset size. The requested rowset overlapped the start of the result set when FetchOrientation was SQL_FETCH_PRIOR, the current position was beyond the end of the result set, and the rowset size was greater than the result set size. The requested rowset overlapped the start of the result set when FetchOrientation was SQL_FETCH_RELATIVE, FetchOffset was negative, and the absolute value of FetchOffset was less than or equal to the rowset size.

		The requested rowset overlapped the start of the result set when FetchOrientation was SQL_FETCH_ABSOLUTE, FetchOffset was negative, and the absolute value of FetchOffset was greater than the result set size but less than or equal to the rowset size. (Function returns SQL_SUCCESS_WITH_INFO.)
01S07	Fractional truncation	The data returned for a column was truncated. For numeric data types, the fractional part of the number was truncated. For time, timestamp, and interval data types containing a time component, the fractional portion of the time was truncated. (Function returns SQL_SUCCESS_WITH_INFO.)
07006	Restricted data type attribute violation	The data value of a column in the result set could not be converted to the data type specified by TargetType in SQLBindCol. Column 0 was bound with a data type of SQL_C_BOOKMARK, and the SQL_ATTR_USE_BOOKMARKS statement attribute was set to SQL_UB_VARIABLE. Column 0 was bound with a data type of SQL_C_VARBOOKMARK, and the SQL_ATTR_USE_BOOKMARKS statement attribute was not set to SQL_UB_VARIABLE.
07009	Invalid descriptor index	The driver was an ODBC 2.x driver that does not support SQLExtendedFetch, and a column number specified in the binding for a column was 0. Column 0 was bound, and the SQL_ATTR_USE_BOOKMARKS statement attribute was set to SQL_UB_OFF.
08S01	Communication link failure	The communication link between the driver and the data source to which the driver was connected failed before the function completed processing.
22001	String data, right truncated	A variable-length bookmark returned for a column was truncated.
22002	Indicator variable required but not supplied	NULL data was fetched into a column whose StrLen_or_IndPtr set by SQLBindCol (or SQL_DESC_INDICATOR_PTR set by SQLSetDescField or SQLSetDescRec) was a null pointer.
22003	Numeric value out of range	Returning the numeric value (as numeric or string) for one or more bound columns would have caused the whole (as opposed to fractional) part of the number to be truncated.
22007	Invalid datetime format	A character column in the result set was bound to a date, time, or timestamp C structure, and a value in the column was, respectively, an invalid date, time, or timestamp.
22012	Division by zero	A value from an arithmetic expression was returned, which resulted in division by zero.
22015	Interval field overflow	Assigning from an exact numeric or interval SQL type to an interval C type caused a loss of significant digits in the leading field. When fetching data to an interval C type, there was no representation of the value of the SQL type in the interval C type.
22018	Invalid character value for cast specification	A character column in the result set was bound to a character C buffer, and the column contained a character for which there was no representation in the character set of the buffer. The C type was an exact or approximate numeric, a datetime, or an interval data type; the SQL type of the column was a character data type; and the value in the column was not a valid literal of the bound C type.
24000	Invalid cursor state	The StatementHandle was in an executed state but no result set was associated with the StatementHandle.
40001	Serialization failure	The transaction in which the fetch was executed was terminated to prevent deadlock.
40003	Statement completion unknown	The associated connection failed during the execution of this function, and the state of the transaction cannot be determined.

HY000	General error	An error occurred for which there was no specific SQLSTATE and for which no implementation-specific SQLSTATE was defined. The error message returned by SQLGetDiagRec in the *MessageText buffer describes the error and its cause.
HY001	Memory allocation error	The driver was unable to allocate memory required to support execution or completion of the function.
HY008	Operation canceled	Asynchronous processing was enabled for the StatementHandle. The function was called, and before it completed execution, SQLCancel was called on the StatementHandle. Then the function was called again on the StatementHandle. The function was called, and before it completed execution, SQLCancel was called on the StatementHandle from a different thread in a multithread application.
HY010	Function sequence error	(DM) The specified StatementHandle was not in an executed state. The function was called without first calling SQLExecDirect, SQLExecute or a catalog function. (DM) An asynchronously executing function (not this one) was called for the StatementHandle and was still executing when this function was called. (DM) SQLExecute, SQLExecDirect, SQLBulkOperations, or SQLSetPos was called for the StatementHandle and returned SQL_NEED_DATA. This function was called before data was sent for all data-at-execution parameters or columns. (DM) SQLFetch was called for the StatementHandle after SQLExtendedFetch was called and before SQLFreeStmt with the SQL_CLOSE option was called.
HY013	Memory management error	The function call could not be processed because the underlying memory objects could not be accessed, possibly because of low memory conditions.
HY090	Invalid string or buffer length	The SQL_ATTR_USE_BOOKMARK statement attribute was set to SQL_UB_VARIABLE, and column 0 was bound to a buffer whose length was not equal to the maximum length for the bookmark for this result set. (This length is available in the SQL_DESC_OCTET_LENGTH field of the IRD and can be obtained by calling SQLDescribeCol, SQLColAttribute, or SQLGetDescField.)
HY106	Fetch type out of range	(DM) The value specified for the argument FetchOrientation was invalid. (DM) The argument FetchOrientation was SQL_FETCH_BOOKMARK, and the SQL_ATTR_USE_BOOKMARKS statement attribute was set to SQL_UB_OFF. The value of the SQL_ATTR_CURSOR_TYPE statement attribute was SQL_CURSOR_FORWARD_ONLY, and the value of argument FetchOrientation was not SQL_FETCH_NEXT. The value of the SQL_ATTR_CURSOR_SCROLLABLE statement attribute was SQL_NONSCROLLABLE, and the value of argument FetchOrientation was not SQL_FETCH_NEXT.
HY107	Row value out of range	The value specified with the SQL_ATTR_CURSOR_TYPE statement attribute was SQL_CURSOR_KEYSET_DRIVEN, but the value specified with the SQL_ATTR_KEYSET_SIZE statement attribute was greater than 0 and less than the value specified with the SQL_ATTR_ROW_ARRAY_SIZE statement attribute.
HY111	Invalid bookmark value	The argument FetchOrientation was SQL_FETCH_BOOKMARK, and the bookmark pointed to by the value in the SQL_ATTR_FETCH_BOOKMARK_PTR statement attribute was not valid or was a null pointer.
HYC00	Optional feature not implemented	The driver or data source does not support the conversion specified by the combination of the TargetType in SQLBindCol and the SQL

		data type of the corresponding column.
HYT01	Connection timeout expired	The connection timeout period expired before the data source responded to the request. The connection timeout period is set through SQLSetConnectAttr, SQL_ATTR_CONNECTION_TIMEOUT.

Comments

SQLFetchScroll returns a specified rowset from the result set. Rowsets can be specified by absolute or relative position or by bookmark. SQLFetchScroll can be called only while a result set exists — that is, after a call that creates a result set and before the cursor over that result set is closed. If any columns are bound, it returns the data in those columns. If the application has specified a pointer to a row status array or a buffer in which to return the number of rows fetched, SQLFetchScroll returns this information as well. Calls to SQLFetchScroll can be mixed with calls to SQLFetch but cannot be mixed with calls to SQLExtendedFetch.

Related Functions

Function	Description
SQLBindCol	Binding a buffer to a column in a result set
SQLBulkOperations	Performing bulk insert, update, or delete operations
SQLCancel	Canceling statement processing
SQLDescribeCol	Returning information about a column in a result set
SQLExecDirect	Executing an SQL statement
SQLExecute	Executing a prepared SQL statement
SQLFetch	Fetching a single row or a block of data in a forward-only direction
SQLFreeStmt	Closing the cursor on the statement
SQLNumResultCols	Returning the number of result set columns
SQLSetPos	Positioning the cursor, refreshing data in the rowset, or updating or deleting data in the result set
SQLSetStmtAttr	Setting a statement attribute

Code Example

```
#define ROW_ARRAY_SIZE 10

SQLUINTEGER    OrderIDArray[ROW_ARRAY_SIZE], NumRowsFetched;
SQLCHAR        SalesPersonArray[ROW_ARRAY_SIZE][11],
               StatusArray[ROW_ARRAY_SIZE][7];
SQLINTEGER     OrderIDIndArray[ROW_ARRAY_SIZE],
               SalesPersonLenOrIndArray[ROW_ARRAY_SIZE],
               StatusLenOrIndArray[ROW_ARRAY_SIZE];
SQLUSMALLINT   RowStatusArray[ROW_ARRAY_SIZE], i;
SQLRETURN      rc;
SQLHSTMT       hstmt;

// Set the SQL_ATTR_ROW_BIND_TYPE statement attribute to use
// column-wise binding. Declare the rowset size with the
// SQL_ATTR_ROW_ARRAY_SIZE statement attribute. Set the
// SQL_ATTR_ROW_STATUS_PTR statement attribute to point to the
// row status array. Set the SQL_ATTR_ROWS_FETCHED_PTR statement
// attribute to point to cRowsFetched.
```

```
SQLSetStmtAttr(hstmt, SQL_ATTR_ROW_BIND_TYPE, SQL_BIND_BY_COLUMN, 0);
SQLSetStmtAttr(hstmt, SQL_ATTR_ROW_ARRAY_SIZE, ROW_ARRAY_SIZE, 0);
SQLSetStmtAttr(hstmt, SQL_ATTR_ROW_STATUS_PTR, RowStatusArray, 0);
SQLSetStmtAttr(hstmt, SQL_ATTR_ROWS_FETCHED_PTR, &NumRowsFetched, 0);

// Bind arrays to the OrderID, SalesPerson, and Status columns.
SQLBindCol(hstmt, 1, SQL_C_ULONG, OrderIDArray, 0, OrderIDIndArray);
SQLBindCol(hstmt, 2, SQL_C_CHAR, SalesPersonArray,
sizeof(SalesPersonArray[0]),
            SalesPersonLenOrIndArray);
SQLBindCol(hstmt, 3, SQL_C_CHAR, StatusArray, sizeof(StatusArray[0]),
            StatusLenOrIndArray);

// Execute a statement to retrieve rows from the Orders table.
SQLExecDirect(hstmt, "SELECT OrderID, SalesPerson, Status FROM Orders",
SQL_NTS);

// Fetch up to the rowset size number of rows at a time. Print the actual
// number of rows fetched; this number is returned in NumRowsFetched.
// Check the row status array to print only those rows successfully
// fetched. Code to check if rc equals SQL_SUCCESS_WITH_INFO or
// SQL_ERROR not shown.
while ((rc = SQLFetchScroll(hstmt,SQL_FETCH_NEXT,0)) != SQL_NO_DATA) {
   for (i = 0; i < NumRowsFetched; i++) {
      if ((RowStatusArray[i] == SQL_ROW_SUCCESS) ||
            (RowStatusArray[i] == SQL_ROW_SUCCESS_WITH_INFO)) {
         if (OrderIDIndArray[i] == SQL_NULL_DATA)
            printf(" NULL      ");
         else
            printf("%d\t", OrderIDArray[i]);
         if (SalesPersonLenOrIndArray[i] == SQL_NULL_DATA)
            printf(" NULL      ");
         else
            printf("%s\t", SalesPersonArray[i]);
         if (StatusLenOrIndArray[i] == SQL_NULL_DATA)
            printf(" NULL\n");
         else
            printf("%s\n", StatusArray[i]);
      }
   }
}

// Close the cursor.
SQLCloseCursor(hstmt);
```

4.28 SQLForeignKeys

SQLForeignKeys returns both a list of foreign keys for a specified table and a list of foreign keys in other tables that refer to the primary key in the specified table.

Syntax

RETCODE = SQLForeignKeys (hstmt, szPkTableQualifier, cbPkTableQualifier, szPkTableOwner, cbPkTableOwner, szPkTableName, cbPkTableName, szFkTableQualifer, cbFkTableQualifer, szFkTableOwner, cbFkTableOwner, szFkTableName, cbFkTableName)

Arguments

Type	Argument	Use	Description
Long	hstmt	Input	The statement handle.
String	szPkTableQualifier	Input	The buffer containing the primary key table qualifier.
Integer	cbPkTableQualifier	Input	The length of the primary key table qualifier.
String	szPkTableOwner	Input	The buffer containing the primary key table-owner name.
Integer	cbPkTableOwner	Input	The length of the primary key table owner.
String	szPkTableName	Input	The buffer containing the primary key table name.
Integer	cbPkTableName	Input	The length of the primary key table name.
String	szFkTableQualifer	Input	The buffer containing the foreign key table qualifier.
Integer	cbFkTableQualifer	Input	The length of the foreign key table qualifier.
String	szFkTableOwner	Input	The buffer containing the foreign key owner name.
Integer	cbFkTableOwner	Input	The length of the foreign key table owner.
String	szFkTableName	Input	The buffer containing the foreign key table name.
Integer	cbFkTableName	Input	The length of the foreign key table name.

Return Values

SQL_SUCCESS, SQL_ERROR, or SQL_INVALID_HANDLE

Result Set

Column Name	Data Type	Comments
PKTABLE_QUALIFIER	TEXT 18	The primary key table qualifier. The Oterro Engine returns NULL.
PKTABLE_OWNER	TEXT 18	The primary key table owner. The Oterro Engine returns NULL.
PKTABLE_NAME	TEXT 18	The primary key table name.
PKCOLUMN_NAME	TEXT 18	The primary key column name.
FKTABLE_QUALIFIER	TEXT 18	The foreign key table qualifier. The Oterro Engine returns NULL.
FKTABLE_OWNER	TEXT 18	The foreign key table owner. The Oterro Engine returns NULL.
FKTABLE_NAME	TEXT 18	The foreign key table name.
FKCOLUMN_NAME	TEXT 18	The foreign key column name.
KEY_SEQ	INTEGER	The column-sequence number in key (starting with 1) for multi column keys.
UPDATE_RULE	INTEGER	The action to be applied to the foreign key when the SQL operation is UPDATE: SQL_CASCADE = 1 SQL_RESTRICT = 0
DELETE_RULE		The action to be applied to the foreign key when the SQL operation is DELETE: SQL_CASCADE = 1 SQL_RESTRICT = 0
FK_NAME	TEXT 18	The foreign key name.
PF_NAME	TEXT 18	The primary key name.

The lengths of text columns shown in the table are maximums; to determine the actual lengths, use the SQLGetInfo function.

Comments

When szPkTableName contains a table name, SQLForeignKeys returns a result set containing the primary key of the specified table and all of the foreign keys (in other tables) that refer to it.

When szFkTableName contains a table name, SQLForeignKeys returns a result set containing all of the foreign keys in the specified table and the primary keys (in other tables) to which they refer.

When both szPkTableName and szFkTableName contain table names, SQLForeignKeys returns the foreign keys in the table specified in szFkTableName that refer to the primary key in the table specified in szPkTableName.

SQLForeignKeys returns a standard result set; when the foreign keys associated with a primary key are requested, the result set is ordered by FKTABLE_QUALIFIER, FKTABLE_OWNER, FKTABLE_NAME, and KEY_SEQ. When the primary keys associated with a foreign key are requested, the result set is ordered by PKTABLE_QUALIFIER, PKTABLE_OWNER, PKTABLE_NAME, and KEY_SEQ.

Related Functions

Function	Description
SQLPrimaryKeys	Returns the columns defined as primary keys.
SQLStatistics	Returns statistics for tables and indexes.

Errors

SQLSTATE	Description
01000	A driver-specific informational message. (The function returns SQL_SUCCESS_WITH_INFO.)
08S01	The data source connection failed before the function completed processing.
24000	An invalid cursor state: A cursor is currently open on the statement handle.
IM001	The driver associated with the hstmt does not support the function.
S1000	An error has occurred that has no defined SQLSTATE—see the error message text.
S1001	A memory allocation failure.
S1009	An invalid argument value—a null pointer was passed.
S1090	An invalid string or buffer length.
S1C00	The driver or data source does not support the specified type.

Visual Basic Example

SQLAPI.BAS:
Declare Function SQLForeignKeys Lib "ODBC32.DLL" (ByVal hstmt&, ByVal szPkTableQualifier$, ByVal cbPkTableQualifier%, ByVal szPkTableOwner$, ByVal cbPkTableOwner%, ByVal szPkTableName$, ByVal cbPkTableName%, ByVal
    szFkTableQualifier$, ByVal cbFkTableQualifier%, ByVal szFkTableOwner$, ByVal cbFkTableOwner%, ByVal szFkTableName$, ByVal cbFkTableName%) As Integer

CODE:
```
Global colnum As Integer
Global szTableName As String * 20
Global cbTableName As Integer
Global szFirst As String * 1500
Global cbFirst As Long

Sub fkconst()
     Dim i As Integer
     Dim n As Integer
     Dim xarray1(13) As Variant
     retcode = SQLForeignKeys(hstmt&, "", 0, "", 0, "", 0, "", 0, "", 0,
szTableName, cbTableName)
          errorcheck retcode
     n = 0
     n = n + 1
     i = 1
     retcode = SQLNumResultCols(hstmt&, colnum)
          errorcheck retcode
     Do While SQLFetch(hstmt&) = SQL_SUCCESS
          Do While i <= colnum
               retcode = SQLGetData(hstmt&, i, SQL_C_CHAR, szFirst, 255,
cbFirst)
               xarray1(i) = Chop(szFirst)
               i = i + 1
          Loop
          dbstr1.Grid4.Row = n
          dbstr1.Grid4.Col = 4             'fk col
          dbstr1.Grid4.Text = xarray1(8)
          dbstr1.Grid4.Col = 5             'pk table
          dbstr1.Grid4.Text = xarray1(3)
          dbstr1.Grid4.Col = 6             'fk seq
          dbstr1.Grid4.Text = xarray1(9)
          dbstr1.Grid4.Col = 7             'fk index name
          dbstr1.Grid4.Text = xarray1(12)
          dbstr1.Grid4.Col = 8             'pk col
          dbstr1.Grid4.Text = xarray1(4)
          dbstr1.Grid4.Col = 9             'pk index name
          dbstr1.Grid4.Text = xarray1(13)
          n = n + 1
          i = 1
     Loop
     retcode = SQLFreeStmt(hstmt&, SQL_CLOSE)
End Sub
```

4.29 SQLFreeConnect

SQLFreeConnect frees the database connection handle. All memory associated with the connection handle is released.

Syntax

RETCODE = SQLFreeConnect (hdbc)

Arguments

Type	Argument	Use	Description
Long	hdbc	Input	The database connection handle.

Return Values

SQL_SUCCESS, SQL_ERROR, or SQL_INVALID_HANDLE

Comments

When SQL_ERROR is returned, the connection handle is not released. When SQL_INVALID_HANDLE is returned, the handle is invalid and the connection handle is not released.

An error occurs when SQLFreeConnect is called before SQLDisconnect and the connection is not released.

SQLFreeConnect does not validate the handles that are passed to it. Because SQLFreeConnect does not validate handles, when you try to pass it an undefined or already freed handle, you could get unpredictable results.

Related Functions

Function	Description
SQLAllocConnect	Allocates a connection handle.
SQLBrowseConnect	Retrieves values required to connect to a data source.
SQLConnect	Opens a connection to a database.
SQLDisconnect	Closes the connection to a database.
SQLDriverConnect	Prompts for information to open a connection to a database through the ODBC Driver Manager.

Errors

SQLSTATE	Description
S1000	An error has occurred that has no defined SQLSTATE—see the error message text.
S1010	An attempt to free the connection handle prior to disconnecting has been made.

Visual Basic Example

SQLAPI.BAS:
Declare Function SQLFreeConnect Lib "ODBC32.DLL" (ByVal hdbc&) As Integer

CODE:
```
Global Const xdisco = "No Database Connected"
Global Const xexit = "end"
Global dbstr As String

quit xdisco

Sub quit(xend As String)    'xend specifies whether to disconnect or exit
     retcode = SQLFreeStmt(hstmt&, SQL_DROP)
     retcode = SQLDisconnect(hdbc&)
     retcode = SQLFreeConnect(hdbc&)
     If hdbc2 <> 0 Then
          retcode = SQLDisconnect(hdbc2&)
          retcode = SQLFreeConnect(hdbc2&)
     End If
     dbstr = xdisco
     If xend = "end" Then
          retcode = SQLFreeEnv (henv&)
          End
     End If
End Sub
```

4.30 SQLFreeHandle

SQLFreeHandle also replaces the ODBC 2.0 function SQLFreeStmt (with the SQL_DROP Option) for freeing a statement handle.

This function is a generic function for freeing handles. It replaces the ODBC 2.0 functions SQLFreeConnect (for freeing a connection handle) and SQLFreeEnv (for freeing an environment handle). SQLFreeConnect and SQLFreeEnv are both deprecated in ODBC 3.x. SQLFreeHandle also replaces the ODBC 2.0 function SQLFreeStmt (with the SQL_DROP Option) for freeing a statement handle. For more information, see "Comments."

Syntax

```
RETCODE = SQLFreeHandle(HandleType,Handle)
```

Arguments

Type	Argument	Use	Description
Integer	HandleType	Input	The type of handle to be freed by SQLFreeHandle. Must be one of the following values: SQL_HANDLE_ENV SQL_HANDLE_DBC SQL_HANDLE_STMT SQL_HANDLE_DESC If HandleType is not one of these values, SQLFreeHandle returns SQL_INVALID_HANDLE.
Long	Handle	Input	The handle to be freed.

Return Values

SQL_SUCCESS, SQL_ERROR, or SQL_INVALID_HANDLE

If SQLFreeHandle returns SQL_ERROR, the handle is still valid.

Errors

When SQLFreeHandle returns SQL_ERROR, an associated SQLSTATE value may be obtained from the diagnostic data structure for the handle that SQLFreeHandle tried to free but could not. The following table lists the SQLSTATE values typically returned by SQLFreeHandle and explains each one in the context of this function; the notation "(DM)" precedes the descriptions of SQLSTATEs returned by the Driver Manager. The return code associated with each SQLSTATE value is SQL_ERROR, unless noted otherwise.

SQLSTATE	Error	Description
HY000	General error	An error occurred for which there was no specific SQLSTATE and for which no implementation-specific SQLSTATE was defined. The error message returned by SQLGetDiagRec in the *MessageText buffer describes the error and its cause.
HY001	Memory allocation error	The driver was unable to allocate memory that is required to support execution or completion of the function.
HY010	Function sequence error	(DM) The HandleType argument was SQL_HANDLE_ENV, and at least one connection was in an allocated or connected state. SQLDisconnect and SQLFreeHandle with a HandleType of SQL_HANDLE_DBC must be called for each connection before calling SQLFreeHandle with a HandleType of SQL_HANDLE_ENV.

		(DM) The HandleType argument was SQL_HANDLE_DBC, and the function was called before calling SQLDisconnect for the connection. (DM) The HandleType argument was SQL_HANDLE_STMT; an asynchronously executing function was called on the statement handle; and the function was still executing when this function was called. (DM) The HandleType argument was SQL_HANDLE_STMT; SQLExecute, SQLExecDirect, SQLBulkOperations, or SQLSetPos was called with the statement handle and returned SQL_NEED_DATA. This function was called before data was sent for all data-at-execution parameters or columns. (DM) All subsidiary handles and other resources were not released before SQLFreeHandle was called.
HY013	Memory management error	The HandleType argument was SQL_HANDLE_STMT or SQL_HANDLE_DESC, and the function call could not be processed because the underlying memory objects could not be accessed, possibly because of low memory conditions.
HY017	Invalid use of an automatically allocated descriptor handle.	(DM) The Handle argument was set to the handle for an automatically allocated descriptor.
HYT01	Connection timeout expired	The connection timeout period expired before the data source responded to the request. The connection timeout period is set through SQLSetConnectAttr, SQL_ATTR_CONNECTION_TIMEOUT.
IM001	Driver does not support this function	(DM) The HandleType argument was SQL_HANDLE_DESC, and the driver was an ODBC 2.x driver. (DM) The HandleType argument was SQL_HANDLE_STMT, and the driver was not a valid ODBC driver.

Comments

SQLFreeHandle is used to free handles for environments, connections, statements, and descriptors, as described in the following sections. For general information about handles, see Handles.

An application should not use a handle after it has been freed; the Driver Manager does not check the validity of a handle in a function call.

Related Functions

Function	Description
SQLAllocHandle	Allocating a handle
SQLCancel	Canceling statement processing
SQLSetCursorName	Setting a cursor name

Code Example

```
// SQLConnect_ref.cpp
// compile with: odbc32.lib
#include <windows.h>
#include <sqlext.h>

int main() {
   SQLHENV henv;
   SQLHDBC hdbc;
   SQLHSTMT hstmt;
   SQLRETURN retcode;
   SQLPOINTER rgbValue;
   int i = 5;
   rgbValue = &i;

   SQLCHAR * OutConnStr = (SQLCHAR * )malloc(255);
   SQLSMALLINT * OutConnStrLen = (SQLSMALLINT *)malloc(255);

   // Allocate environment handle
   retcode = SQLAllocHandle(SQL_HANDLE_ENV, SQL_NULL_HANDLE, &henv);

   // Set the ODBC version environment attribute
   if (retcode == SQL_SUCCESS || retcode == SQL_SUCCESS_WITH_INFO) {
      retcode = SQLSetEnvAttr(henv, SQL_ATTR_ODBC_VERSION, (void*)
SQL_OV_ODBC3, 0);

      // Allocate connection handle
      if (retcode == SQL_SUCCESS || retcode == SQL_SUCCESS_WITH_INFO) {
         retcode = SQLAllocHandle(SQL_HANDLE_DBC, henv, &hdbc);

         // Set login timeout to 5 seconds
         if (retcode == SQL_SUCCESS || retcode == SQL_SUCCESS_WITH_INFO) {
            SQLSetConnectAttr(hdbc, SQL_LOGIN_TIMEOUT, (SQLPOINTER)
(rgbValue), 0);

            // Connect to data source
            retcode = SQLConnect(hdbc, (SQLCHAR*) "NorthWind", SQL_NTS,
(SQLCHAR*) NULL, 0, NULL, 0);

            // Allocate statement handle
            if (retcode == SQL_SUCCESS || retcode == SQL_SUCCESS_WITH_INFO)
{
               retcode = SQLAllocHandle(SQL_HANDLE_STMT, hdbc, &hstmt);

               // Process data
               if (retcode == SQL_SUCCESS || retcode ==
SQL_SUCCESS_WITH_INFO) {
                  SQLFreeHandle(SQL_HANDLE_STMT, hstmt);
               }

               SQLDisconnect(hdbc);
            }

            SQLFreeHandle(SQL_HANDLE_DBC, hdbc);
         }
      }
      SQLFreeHandle(SQL_HANDLE_ENV, henv);
   }
}
```

Copyright © 1982-2025 R:BASE Technologies, Inc.

4.31 SQLFreeEnv

SQLFreeEnv frees the environment handle. All memory associated with the handle is released.

Syntax

```
RETCODE = SQLFreeEnv (henv)
```

Arguments

Type	Argument	Use	Description
Long	henv	Input	The environment handle.

Return Values

`SQL_SUCCESS, SQL_ERROR, or SQL_INVALID_HANDLE`

Comments

When SQL_ERROR is returned, the environment handle is not released. When SQL_INVALID_HANDLE is returned, the handle is invalid and the environment handle is not released.

All connections must be released before calling SQLFreeEnv.

SQLFreeEnv does not validate the handles that are passed to it. Because SQLFreeEnv does not validate handles, when you try to pass it an undefined or already freed handle, you could get unpredictable results.

Related Functions

Function	Description
SQLAllocEnv	Allocates an environment handle.

Errors

SQLSTATE	Description
S1000	An error has occurred that has no defined SQLSTATE—see the error message text.
S1010	An attempt to free the environment handle prior to disconnecting and freeing all current connection handles has been made.

Visual Basic Example

SQLAPI.BAS:
Declare Function SQLFreeEnv Lib "ODBC32.DLL" (ByVal henv&) As Integer

CODE:
```
Global Const xdisco = "No Database Connected"
Global Const xexit = "end"
Global dbstr As String
```

```
quit xdisco

Sub quit(xend As String)    'xend specifies whether to disconnect or exit
     retcode = SQLFreeStmt(hstmt&, SQL_DROP)
     retcode = SQLDisconnect(hdbc&)
     retcode = SQLFreeConnect(hdbc&)
     If hdbc2 <> 0 Then
          retcode = SQLDisconnect(hdbc2&)
          retcode = SQLFreeConnect(hdbc2&)
     End If
     dbstr = xdisco
     If xend = "end" Then
          retcode = SQLFreeEnv (henv&)
          End
     End If
End Sub
```

4.32 SQLFreeStmt

SQLFreeStmt ends processing on the statement hstmt as specified by the fOption argument.

Syntax

RETCODE = SQLFreeStmt (hstmt, fOption)

Arguments

Type	Argument	Use	Description
Long	hstmt	Input	The statement handle.
Integer	fOption	Input	The end option; accepts one of the following: SQL_CLOSE SQL_DROP SQL_UNBIND SQL_RESET_PARAMS

Return Values

SQL_SUCCESS, SQL_ERROR, or SQL_INVALID_HANDLE

Comments

Following is an explanation of the fOption arguments:

When SQL_CLOSE is specified, any cursor associated with the hstmt is closed and the result set associated with the statement handle is discarded. The cursor can be reopened by reissuing SQLExecDirect or SQLExecute with the same or different parameters.

When SQL_DROP is specified, the memory allocated for the statement handle is released and the hstmt can't be accessed. Any open cursors are closed and the result set associated with the statement handle is discarded. This option frees all resources associated with the hstmt.

When SQL_UNBIND is specified, all buffers for columns bound by SQLBindCol are released.

When SQL_RESET_PARAMS is specified, all parameters bound by SQLBindParameter are released.

SQLFreeStmt does not validate the handles that are passed to it. Because SQLFreeStmt does not validate handles, when you try to pass it an undefined or already freed handle, you could get unpredictable results.

When using the ODBC Driver Manager, you cannot do two SQLFreeStmt commands referencing the same statement handle in a row. Instead of doing an SQLFreeStmt (hstmt,SQL_CLOSE) and then an SQLFreeStmt (hstmt,SQL_DROP), just do the SQLFreeStmt (hstmt,SQL_DROP). The result is the same. Then reallocate the statement as neccessary.

Related Functions

Function	Description
SQLAllocStmt	Allocates a new statement handle.
SQLCancel	Ends processing on a statement.

Errors

SQLSTATE	Description
S1000	An error has occurred that has no defined SQLSTATE—see the error message text.
S1001	A memory allocation failure.
S1092	An option type was out of range.

Visual Basic Example

SQLAPI.BAS:
Declare Function SQLFreeStmt Lib "ODBC32.DLL" (ByVal hstmt&, ByVal fOption%) As Integer

CODE:

```
Global colnum As Integer
Global bufstring As String
Global cbcol1 As Long
Global colresults As String * 5000
Global starttime As Date
Global endtime As Date
Global elapsed As Integer

Sub getall()
     Dim i As Integer
     retcode = SQLNumResultCols(hstmt, colnum)
          errorcheck retcode
     bufstring = sql1.Text
     results.AddItem UCase(bufstring)
     starttime = Time
     Do While SQLFetch(hstmt&) = SQL_SUCCESS
          i = 1
          Do While i <= colnum
               retcode = SQLGetData(hstmt&, i, SQL_C_CHAR, colresults,
5000, cbcol1)
                    errorcheck retcode
               If i = 1 Then
                    bufstring = Chop(colresults)
               Else
```

```
                    bufstring = bufstring & "," & Chop(colresults)
               End If
               i = i + 1
          Loop
          results.AddItem bufstring
     Loop
     endtime = Time
     elapsed = DateDiff("s", starttime, endtime)
     bufstring = "Elapsed Time:   " & elapsed & "   seconds"
     results.AddItem bufstring
     retcode = SQLFreeStmt(hstmt&, SQL_CLOSE)
End Sub
```

4.33 SQLGetConnectAttr

SQLGetConnectAttr returns the current setting of a connection attribute.

Syntax

RETCODE = SQLGetConnectAttr(ConnectionHandle,Attribute,ValuePtr,BufferLength,StringLengthPtr);

Arguments

Type	Argument	Use	Description
Long	ConnectionHandle	Input	Connection handle.
Integer	Attribute	Input	Attribute to retrieve.
Long	ValuePtr	Output	A pointer to memory in which to return the current value of the attribute specified by Attribute.
Integer	BufferLength	Input	If Attribute is an ODBC-defined attribute and ValuePtr points to a character string or a binary buffer, this argument should be the length of *ValuePtr. If Attribute is an ODBC-defined attribute and *ValuePtr is an integer, BufferLength is ignored. If the value in *ValuePtr is a Unicode string (when calling SQLGetConnectAttrW), the BufferLength argument must be an even number. If Attribute is a driver-defined attribute, the application indicates the nature of the attribute to the Driver Manager by setting the BufferLength argument. BufferLength can have the following values: If *ValuePtr is a pointer to a character string, BufferLength is the length of the string. If *ValuePtr is a pointer to a binary buffer, the application places the result of the SQL_LEN_BINARY_ATTR(length) macro in BufferLength. This places a negative value in BufferLength. If *ValuePtr is a pointer to a value other than a character string or binary string, BufferLength should have the value SQL_IS_POINTER. If *ValuePtr contains a fixed-length data type, BufferLength is either SQL_IS_INTEGER or SQL_IS_UINTEGER, as appropriate.
Integer	StringLengthPtr	Output	A pointer to a buffer in which to return the total number of bytes (excluding the null-termination character) available to return in *ValuePtr. If *ValuePtr is a null pointer, no length is returned. If the attribute value is a character string and the number of bytes available to return is greater than BufferLength minus the length of the null-

			termination character, the data in *ValuePtr is truncated to BufferLength minus the length of the null-termination character and is null-terminated by the driver.

Return Values

SQL_SUCCESS, SQL_SUCCESS_WITH_INFO, SQL_NO_DATA, SQL_ERROR, or SQL_INVALID_HANDLE

Errors

When SQLGetConnectAttr returns SQL_ERROR or SQL_SUCCESS_WITH_INFO, an associated SQLSTATE value can be obtained from the diagnostic data structure by calling SQLGetDiagRec with a HandleType of SQL_HANDLE_DBC and a Handle of ConnectionHandle. The following table lists the SQLSTATE values typically returned by SQLGetConnectAttr and explains each one in the context of this function; the notation "(DM)" precedes the descriptions of SQLSTATEs returned by the Driver Manager. The return code associated with each SQLSTATE value is SQL_ERROR, unless noted otherwise.

SQLSTATE	**Error**	**Description**
01000	General warning	Driver-specific informational message. (Function returns SQL_SUCCESS_WITH_INFO.)
01004	String data, right truncated	The data returned in *ValuePtr was truncated to be BufferLength minus the length of a null-termination character. The length of the untruncated string value is returned in *StringLengthPtr. (Function returns SQL_SUCCESS_WITH_INFO.)
08003	Connection does not exist	(DM) An Attribute value that required an open connection was specified.
08S01	Communication link failure	The communication link between the driver and the data source to which the driver was connected failed before the function completed processing.
HY000	General error	An error occurred for which there was no specific SQLSTATE and for which no implementation-specific SQLSTATE was defined. The error message returned from the diagnostic data structure by the argument MessageText in SQLGetDiagField describes the error and its cause.
HY001	Memory allocation error	The driver was unable to allocate memory that is required to support execution or completion of the function.
HY010	Function sequence error	(DM) SQLBrowseConnect was called for the ConnectionHandle and returned SQL_NEED_DATA. This function was called before SQLBrowseConnect returned SQL_SUCCESS_WITH_INFO or SQL_SUCCESS.
HY013	Memory management error	The function call could not be processed because the underlying memory objects could not be accessed, possibly because of low memory conditions.
HY090	Invalid string or buffer length	(DM) *ValuePtr is a character string, and BufferLength was less than zero but not equal to SQL_NTS.
HY092	Invalid attribute/option identifier	The value specified for the argument Attribute was not valid for the version of ODBC supported by the driver.
HYC00	Optional feature not implemented	The value specified for the argument Attribute was a valid ODBC connection attribute for the version of ODBC supported by the driver, but was not supported by the driver.
HYT01	Connection timeout expired	The connection timeout period expired before the data source responded to the request. The connection timeout period is set through SQLSetConnectAttr, SQL_ATTR_CONNECTION_TIMEOUT.
IM001	Driver does not support this function	(DM) The driver that corresponds to the ConnectionHandle does not support the function.

Comments

For general information about connection attributes, see Connection Attributes.

For a list of attributes that can be set, see SQLSetConnectAttr. Notice that if Attribute specifies an attribute that returns a string, ValuePtr must be a pointer to a buffer for the string. The maximum length of the returned string, including the null-termination character, will be BufferLength bytes.

Depending on the attribute, an application does not have to establish a connection before calling SQLGetConnectAttr. However, if SQLGetConnectAttr is called and the specified attribute does not have a default and has not been set by a prior call to SQLSetConnectAttr, SQLGetConnectAttr will return SQL_NO_DATA.

If Attribute is SQL_ATTR_ TRACE or SQL_ATTR_ TRACEFILE, ConnectionHandle does not have to be valid, and SQLGetConnectAttr will not return SQL_ERROR or SQL_INVALID_HANDLE if ConnectionHandle is invalid. These attributes apply to all connections. SQLGetConnectAttr will return SQL_ERROR or SQL_INVALID_HANDLE if another argument is invalid.

Although an application can set statement attributes by using SQLSetConnectAttr, an application cannot use SQLGetConnectAttr to retrieve statement attribute values; it must call SQLGetStmtAttr to retrieve the setting of statement attributes.

Both SQL_ATTR_AUTO_IPD and SQL_ATTR_CONNECTION_DEAD connection attributes can be returned by a call to SQLGetConnectAttr but cannot be set by a call to SQLSetConnectAttr.

Related Functions

Function	Description
SQLGetStmtAttr	Returning the setting of a statement attribute
SQLSetConnectAttr	Setting a connection attribute
SQLSetEnvAttr	Setting an environment attribute
SQLSetStmtAttr	Setting a statement attribute

4.34 SQLGetConnectOption

SQLGetConnectOption queries database connection options.

Syntax

```
RETCODE = SQLGetConnectOption (hdbc, usOption, pulParam)
```

Arguments

Type	Argument	Use	Description
Long	hdbc	Input	The database connection handle.
Integer	usOption	Input	The information option to retrieve.
Long	pulParam	Output	The value for usOption; it is a 32-bit integer.

Return Values

```
SQL_SUCCESS, SQL_ERROR, or SQL_INVALID_HANDLE
```

Information Types

All connection options must be set before calling SQLDriverConnect. Connection options for the Oterro Engine are as follows:

usOption Constant	Value	Description
SQL_ACCESS_MODE	101	Can either be: SQL_MODE_READ_WRITE (0) SQL_MODE_READ_ONLY (1)
SQL_AUTOCOMMIT	102	1 = AUTOCOMMIT 0 = manual commit
SQL_MICRORIM_AUTOCONVERT_MODE	1010	1 = convert on 0 = convert off
SQL_MICRORIM_AUTORECOVER_MODE	1009	1 = recover on 0 = recover off
SQL_MICRORIM_AUTOROWVER_MODE	1013	1 = on 0 = off
SQL_MICRORIM_AUTOSYNC_MODE	1011	1 = sync on 0 = sync off
SQL_MICRORIM_AUTOUPGRADE_MODE	1012	1 = upgrade on 0 = upgrade off
SQL_MICRORIM_COMPATIBILITY_MODE	1001	1 = on 0 = off
SQL_MICRORIM_FASTLOCKS_MODE	1008	1 = fastlocks on 0 = fastlocks off
SQL_MICRORIM_MAX_TRANSACTIONS	1003	A number from 1 to 255 designating the maximum number of transactions the Oterro database should allow.
SQL_MICRORIM_MULTIUSER_MODE	1004	1 = multi-user on 0 = multi-user off
SQL_MICRORIM_TRANSACTION_MODE	1006	1 = transactions on 0 = transactions off
SQL_MICRORIM_STATICDB_MODE	1007	1 = static on 0 = static off

Related Functions

Function	Description
SQLAllocConnect	Allocates a connection handle.
SQLConnect	Opens a connection to a database.
SQLDriverConnect	Prompts for information to open a connection to a database.
SQLSetConnectOption	Sets a database connection option.

Errors

SQLSTATE	Description
S1000	An error has occurred that has no defined SQLSTATE—see the error message text.
S1001	A memory allocation failure.
S1092	An option type was out of range.
S1C00	The driver or data source does not support the specified type.

Visual Basic Example

SQLAPI.BAS:
Declare Function SQLGetConnectOption Lib "ODBC32.DLL" (ByVal hdbc&, ByVal usOption%, pulParam&) As Integer

CODE:
```
Global cbcol1 As Long

Private Sub Form_Load()
     If iType = 1 Then
          conn2.check1.Enabled = False
          conn2.Check2.Enabled = False
          retcode = SQLGetConnectOption(hdbc&, SQL_MICRORIM_MULTIUSER_MODE,
cbcol1)
               errorcheck retcode
          If cbcol1 = 1 Then
               conn2.check1.Value = 1
          End If
          retcode = SQLGetConnectOption(hdbc&,
SQL_MICRORIM_TRANSACTION_MODE,     cbcol1)
               errorcheck retcode
          If cbcol1 = 1 Then
               conn2.Check2.Value = 1
          End If
     Else
          conn2.check1.Value = 1
          conn2.Check2.Value = 0
     End If
End Sub
```

4.35 SQLGetCursorName

SQLGetCursorName retrieves the name of the cursor associated with this statement handle.

Syntax

RETCODE = SQLGetCursorName (hstmt, szCursor, cbCursorMax, pcbCursor)

Arguments

Type	Argument	Use	Description
Long	hstmt	Input	The statement handle.
String	szCursor	Output	The buffer containing the cursor name.
Integer	cbCursorMax	Input	The maximum length of the cursor name buffer.
Integer	pcbCursor	Output	The actual length of the cursor name.

Return Values

SQL_SUCCESS, SQL_ERROR, or SQL_INVALID_HANDLE

Comments

A cursor name can only be retrieved when one of the following has been done:

- A SELECT statement has been executed.
- A cursor name has been set with SQLSetCursorName.

Related Functions

Function	Description
SQLExecDirect	Executes an SQL statement.
SQLExecute	Executes a prepared SQL statement.
SQLPrepare	Prepares an SQL statement for execution.
SQLSetCursorName	Sets a cursor name for a statement handle.

Errors

Copyright © 1982-2025 R:BASE Technologies, Inc.

SQLSTATE	Description
01004	The data was truncated.
S1000	An error has occurred that has no defined SQLSTATE—see the error message text.
S1001	A memory allocation failure.
S1015	A cursor name was not available.
S1090	An invalid string or buffer length.

Visual Basic Example

SQLAPI.BAS:
Declare Function SQLGetCursorName Lib "ODBC32.DLL" (ByVal hstmt&, ByVal szCursor$, ByVal cbCursorMax%, pcbCursor%) As Integer

CODE:
```
Global xarray1(50) As Variant
Global colresults As String * 5000
Global cbcolresults As Long
Global sqlstring As String

Private Sub ms2cursor_Click()
     Dim cblong1 As Long
     Dim cbint1 As Integer
     Dim szcur1 As String
     Dim cbcur1 As Integer
     Dim szcur2 As String * 50
     Dim cbcur2 As Integer
     Dim hstmtselect As Long
     Dim hstmtupd As Long
     'to perform a positioned update, you need to define 2 statement
handles
     retcode = SQLAllocStmt(hdbc&, hstmtselect)
     retcode = SQLAllocStmt(hdbc&, hstmtupd)
     'define a cursor name and set it
     szcur1 = "testCursor9x9" & vbNullChar
     cbcur1 = Len(Chop(szcur1))
     retcode = SQLSetStmtOption(hstmtselect, SQL_ROWSET_SIZE, 1)
          errorcheck retcode
     retcode = SQLSetCursorName(hstmtselect, szcur1, cbcur1)
          errorcheck retcode
     'perform a select to activate the cursor
     sqlstring = "select real1 from numbers" & vbNullChar
     retcode = SQLExecDirect(hstmtselect, sqlstring, SQL_NTS)
          errorcheck retcode
     'fetch the third row
     retcode = SQLExtendedFetch(hstmtselect, SQL_FETCH_NEXT, 1, cblong1,
cbint1)
          errorcheck retcode
     retcode = SQLExtendedFetch(hstmtselect, SQL_FETCH_NEXT, 1, cblong1,
cbint1)
          errorcheck retcode
     retcode = SQLExtendedFetch(hstmtselect, SQL_FETCH_NEXT, 1, cblong1,
cbint1)
          errorcheck retcode
     'get the cursor name to use with the update
     retcode = SQLGetCursorName(hstmtselect, szcur2, 18, cbcur2)
          errorcheck retcode
     'execute the update and close both statements
     sqlstring = "update numbers set real1 = 9.9 where current of "
Chop(szcur2) & vbNullChar
     retcode = SQLExecDirect(hstmtupd, sqlstring, SQL_NTS)
           errorcheck retcode
     retcode = SQLFreeStmt(hstmtselect, SQL_CLOSE)
     retcode = SQLFreeStmt(hstmtupd, SQL_CLOSE)
     'verify the update was performed
     sqlstring = "select real1 from numbers" & vbNullChar
     retcode = SQLExecDirect(hstmtselect, sqlstring, SQL_NTS)
          errorcheck retcode
     i = 1
     Do While SQLFetch(hstmtselect) = SQL_SUCCESS
          retcode = SQLGetData(hstmtselect, 1, SQL_C_CHAR, colresults,
5000, cbcolresults)
               errorcheck retcode
          xarray1(i) = Chop(colresults)
          i = i + 1
     Loop
     view2.List1.Clear
     i = 1
     n = 1
     Do While n <> 0
          view2.List1.AddItem xarray1(i)
          i = i + 1
          n = Len(xarray1(i))
     Loop
     retcode = SQLFreeStmt(hstmtselect, SQL_DROP)
     retcode = SQLFreeStmt(hstmtupd, SQL_DROP)
End Sub
```

4.36 SQLGetData

SQLGetData retrieves result data for a single column in the current row.

Syntax

RETCODE = SQLGetData (hstmt, icol, fCType, rgbValue, cbValueMax, pcbValue)

Arguments

Type	Argument	Use	Description
Long	hstmt	Input	The statement handle.
Integer	icol	Input	The column number in the result data, starting at 1.
Integer	fCType	Input	Convert to this C data type in the value buffer.
String	rgbValue	Output	A pointer to storage for the data.
Long	cbValueMax	Input	The maximum length of the rgbValue buffer.
Long	pcbValue	Output	The number of bytes placed in the rgbValue buffer.

Return Values

```
SQL_SUCCESS, SQL_NO_DATA_FOUND, or SQL_INVALID_HANDLE
```

Comments

The maximum amount of data received in a call is determined by the parameter cbValueMax. However, numeric data types not translated to text are written to the buffer regardless of length.

The actual number of bytes available, or the number of bytes written to rgbValue is returned in the parameter pcbValue. Character string buffers are truncated when pcbValue is greater than or equal to cbValueMax, and the return value from SQLGetData is SQL_SUCCESS_WITH_INFO. When this occurs, the number of bytes left awaiting retrieval is determined by pcbValue minus cbValueMax minus 1. A future call to SQLGetData produces data that can be concatenated with the truncated data retrieved in the previous call to SQLGetData. You must make the two calls to retrieve the whole string.

SQLFetch must be called prior to calling SQLGetData.

Related Functions

Function	Description
SQLColAttributes	Returns column attributes in a result set.
SQLDescribeCol	Describes a column in a result set.
SQLExecDirect	Executes an SQL statement.
SQLExecute	Executes a prepared SQL statement.
SQLFetch	Fetches one row of a result set.
SQLNumResultCols	Returns the number of columns in a result set.
SQLPrepare	Prepares an SQL statement for execution.

Errors

SQLSTATE	Description
01004	The data was truncated (SQL_SUCCESS_WITH_INFO was returned).
07006	The specified conversion is illegal.
22003	The numeric value is out of range: A significant truncation would have occurred.
24000	An invalid cursor state: A cursor is currently open on the statement handle.
S1000	An error has occurred that has no defined SQLSTATE—see the error message text.
S1001	A memory allocation failure.
S1002	An invalid column number.
S1003	A program type is out of range: The C type was not valid.
S1009	An invalid argument value—a null pointer was passed.
S1090	An invalid string or buffer length.

Visual Basic Example

SQLAPI.BAS:
Declare Function SQLGetData Lib "ODBC32.DLL" (ByVal hstmt&, ByVal icol%, ByVal fCType%, ByVal rgbValue$, ByVal cbValueMax&, pcbValue&) As Integer

CODE:

```
Global colnum As Integer
Global bufstring As String
Global cbcol1 As Long
Global colresults As String * 5000
Global starttime As Date
Global endtime As Date
Global elapsed As Integer

Sub getall()
     Dim i As Integer
     retcode = SQLNumResultCols(hstmt, colnum)
          errorcheck retcode
     bufstring = sql1.Text
     results.AddItem UCase(bufstring)
     starttime = Time
     Do While SQLFetch(hstmt&) = SQL_SUCCESS
          i = 1
          Do While i <= colnum
               retcode = SQLGetData(hstmt&, i, SQL_C_CHAR, colresults,
5000, cbcol1)
                    errorcheck retcode
               If i = 1 Then
                    bufstring = Chop(colresults)
               Else
                    bufstring = bufstring & "," & Chop(colresults)
               End If
               i = i + 1
          Loop
          results.AddItem bufstring
     Loop
     endtime = Time
     elapsed = DateDiff("s", starttime, endtime)
     bufstring = "Elapsed Time:   " & elapsed & "   seconds"
     results.AddItem bufstring
     retcode = SQLFreeStmt(hstmt&, SQL_CLOSE)
End Sub
```

4.37 SQLGetDiagRec

SQLGetDiagRec returns the current value of the SQLSTATE field of a diagnostic record that contains error, warning, and status information.

A connection handle must be allocated using SQLAllocHandle() before calling this function.

Syntax

RETCODE = SQLGetDiagRec(HandleType, Handle, RecNumber, *SQLState, *NativeErrorPtr, *MessageText, BufferLength, *TextLengthPtr)

Arguments

Type	Argument	Use	Description
Integer	HandleType	Input	A handle-type identifier that describes the type of handle for which diagnostics are desired. Can be SQL_HANDLE_STMT or

			SQL_HANDLE_DBC.
Long	Handle	Input	A handle for the diagnostic data structure, of the type indicated by HandleType.
Integer	RecNumber	Input	Indicates the status record from which the application seeks information. Status records must be 1.
String	SQLState	Output	Pointer to a buffer in which to return a 5 character SQLSTATE code pertaining to the diagnostic record RecNumber. The first two characters indicate the class; the next three indicate the subclass.
Integer	NativeErrorPtr	Output	Pointer to a buffer in which to return the native error code, specific to the data source.
String	MessageText	Output	Pointer to a buffer in which to return the error message text. The fields returned by SQLGetDiagRec() are contained in a text string.
Integer	BufferLength	Input	Length (in bytes) of the MessageText buffer.
Integer	TextLengthPtr	Output	Pointer to a buffer in which to return the total number of bytes (excluding the number of bytes required for the null termination character) available to return in MessageText. If the number of bytes available to return is greater than BufferLength, then the error message text in MessageText is truncated to BufferLength minus the length of the null termination character.

Return Values

SQL_SUCCESS, SQL_SUCCESS_WITH_INFO, SQL_ERROR, or SQL_INVALID_HANDLE

Comments

An application typically calls SQLGetDiagRec when a previous call to an ODBC function returns anything other than SQL_SUCCESS.

SQLGetDiagRec returns a character string containing multiple fields of the diagnostic data structure record.

The following SQLSTATEs can now be returned : 57011, HY024, HY092, HY000, HY012.

SQLGetDiagRec retrieves only the diagnostic information most recently associated with the handle specified in the Handle argument. If the application calls any function, except SQLGetDiagRec, any diagnostic information from the previous calls on the same handle is lost.

HandleType argument

Each handle type can have diagnostic information associated with it. The HandleType argument denotes the handle type of Handle.

Diagnostics

SQLGetDiagRec() does not post error values for itself. It uses the following return values to report the outcome of its own execution:

SQL_SUCCESS	The function successfully returned diagnostic information.
SQL_SUCCESS_WITH_INFO	The MessageText buffer is too small to hold the requested diagnostic message. No diagnostic records are generated. To determine that a truncation occurred, the application must compare BufferLength to the actual number of bytes available, which is written to StringLengthPtr.
SQL_INVALID_HANDLE	The handle indicated by HandleType and Handle is not a valid handle.
SQL_ERROR	One of the following situations occurred: • RecNumber is negative or 0. • BufferLength is less than zero.
SQL_NO_DATA	RecNumber is greater than the number of diagnostic records that existed for the handle specified in Handle. The function also returns SQL_NO_DATA for any positive RecNumber if there are no diagnostic records for Handle.

4.38 SQLGetFunctions

SQLGetFunctions returns information about whether the Oterro Engine supports a specific ODBC function.

Syntax

```
RETCODE = SQLGetFunctions (hdbc, fFunction, pfExists)
```

Arguments

Type	Argument	Use	Description
Long	hdbc	Input	The database connection handle.
Integer	fFunction	Input	A specific function constant, or the constant SQL_API_ALL_FUNCTIONS.
Integer	pfExists	Output	Returns TRUE = 1 or FALSE = 0.

Return Values

```
SQL_SUCCESS, SQL_ERROR, or SQL_INVALID_HANDLE
```

Result Set

fFunction Constant	**Condition**
SQL_API_SQLALLOCCONNECT	TRUE
SQL_API_SQLALLOCENV	TRUE
SQL_API_SQLALLOCSTMT	TRUE
SQL_API_SQLBINDPARAMETER	TRUE
SQL_API_SQLBINDCOL	TRUE
SQL_API_SQLBROWSECONNECT	TRUE
SQL_API_SQLCANCEL	TRUE
SQL_API_SQLCOLATTRIBUTES	TRUE
SQL_API_SQLCOLUMNPRIVILEGES	TRUE
SQL_API_SQLCOLUMNS	TRUE
SQL_API_SQLCONNECT	TRUE
SQL_API_SQLDATASOURCES	TRUE
SQL_API_SQLDESCRIBECOL	TRUE
SQL_API_SQLDESCRIBEPARAM	TRUE
SQL_API_SQLDISCONNECT	TRUE
SQL_API_SQLDRIVERCONNECT	TRUE
SQL_API_SQLDRIVERS	TRUE
SQL_API_SQLERROR	TRUE
SQL_API_SQLEXECDIRECT	TRUE
SQL_API_SQLEXECUTE	TRUE
SQL_API_SQLEXTENDEDFETCH	TRUE
SQL_API_SQLFETCH	TRUE
SQL_API_SQLFOREIGNKEYS	TRUE
SQL_API_SQLFREECONNECT	TRUE
SQL_API_SQLFREEENV	TRUE
SQL_API_SQLFREESTMT	TRUE
SQL_API_SQLGETCONNECTOPTION	TRUE
SQL_API_SQLGETCURSORNAME	TRUE
SQL_API_SQLGETDATA	TRUE

Comments

For SQL_API_ALL_FUNCTIONS, the storage area pointed to by pfExists must be an INTEGER array of 100 elements.

Related Functions

Function	Description
SQLGetInfo	Queries information about a driver or data source.

Errors

SQLSTATE	Description
01000	A driver-specific informational message. (The function returns SQL_SUCCESS_WITH_INFO.)
S1000	An error has occurred that has no defined SQLSTATE—see the error message text.
S1001	A memory allocation failure.
S1095	An invalid fFunction value was specified.

Visual Basic Example

SQLAPI.BAS:
Declare Function SQLGetFunctions Lib "ODBC32.DLL" (ByVal hdbc&, ByVal fFunction%, pfExists%) As Integer

CODE:
```
Private Sub md1func_Click()
     Dim i As Integer
     Dim msg As String
     retcode = SQLGetFunctions(hdbc&, SQL_API_SQLBROWSECONNECT, i)
          errorcheck retcode
     msg = "SQL_API_SQLBROWSECONNECT = " & i
     MsgBox msg, 64, "SQLGetFunctions"
End Sub
```

4.39 SQLGetStmtAttr

SQLGetStmtAttr returns the current setting of a statement attribute.

Syntax

SQLRETURN =
SQLGetStmtAttr(StatementHandle, Attribute, ValuePtr, BufferLength, StringLengthPtr)

Arguments

Type	Argument	Use	Description
Long	StatementHandle	Input	Statement handle.
Integer	Attribute	Input	Attribute to retrieve.
Long	ValuePtr	Output	Pointer to a buffer in which to return the value of the attribute specified in Attribute.
Integer	BufferLength	Input	If Attribute is an ODBC-defined attribute and ValuePtr points to a character string or a binary buffer, this argument should be the length

			of *ValuePtr. If Attribute is an ODBC-defined attribute and *ValuePtr is an integer, BufferLength is ignored. If the value returned in *ValuePtr is a Unicode string (when calling SQLGetStmtAttrW), the BufferLength argument must be an even number. If Attribute is a driver-defined attribute, the application indicates the nature of the attribute to the Driver Manager by setting the BufferLength argument. BufferLength can have the following values: If *ValuePtr is a pointer to a character string, then BufferLength is the length of the string or SQL_NTS. If *ValuePtr is a pointer to a binary buffer, then the application places the result of the SQL_LEN_BINARY_ATTR(length) macro in BufferLength. This places a negative value in BufferLength. If *ValuePtr is a pointer to a value other than a character string or binary string, then BufferLength should have the value SQL_IS_POINTER. If *ValuePtr is contains a fixed-length data type, then BufferLength is either SQL_IS_INTEGER or SQL_IS_UINTEGER, as appropriate.
Integer	StringLengthPtr	Output	A pointer to a buffer in which to return the total number of bytes (excluding the null-termination character) available to return in *ValuePtr. If ValuePtr is a null pointer, no length is returned. If the attribute value is a character string, and the number of bytes available to return is greater than or equal to BufferLength, the data in *ValuePtr is truncated to BufferLength minus the length of a null-termination character and is null-terminated by the driver.

Return Values

SQL_SUCCESS, SQL_SUCCESS_WITH_INFO, SQL_ERROR, or SQL_INVALID_HANDLE

Errors

When SQLGetStmtAttr returns SQL_ERROR or SQL_SUCCESS_WITH_INFO, an associated SQLSTATE value may be obtained by calling SQLGetDiagRec with a HandleType of SQL_HANDLE_STMT and a Handle of StatementHandle. The following table lists the SQLSTATE values commonly returned by SQLGetStmtAttr and explains each one in the context of this function; the notation "(DM)" precedes the descriptions of SQLSTATEs returned by the Driver Manager. The return code associated with each SQLSTATE value is SQL_ERROR, unless noted otherwise.

SQLSTATE	**Error**	**Description**
01000	General warning	Driver-specific informational message. (Function returns SQL_SUCCESS_WITH_INFO.)
01004	String data, right truncated	The data returned in *ValuePtr was truncated to be BufferLength minus the length of a null-termination character. The length of the untruncated string value is returned in *StringLengthPtr. (Function returns SQL_SUCCESS_WITH_INFO.)
24000	Invalid cursor state	The argument Attribute was SQL_ATTR_ROW_NUMBER and the cursor was not open, or the cursor was positioned before the start of the result set or after the end of the result set.
HY000	General error	An error occurred for which there was no specific SQLSTATE and for which no implementation-specific SQLSTATE was defined. The error message returned by SQLGetDiagRec in the argument MessageText describes the error and its cause.
HY001	Memory allocation error	The driver was unable to allocate memory required to support execution or completion of the function.

HY010	Function sequence error	(DM) An asynchronously executing function was called for the StatementHandle and was still executing when this function was called.(DM) SQLExecute, SQLExecDirect, SQLBulkOperations, or SQLSetPos was called for the StatementHandle and returned SQL_NEED_DATA. This function was called before data was sent for all data-at-execution parameters or columns.
HY013	Memory management error	The function call could not be processed because the underlying memory objects could not be accessed, possibly because of low memory conditions.
HY090	Invalid string or buffer length	(DM) *ValuePtr is a character string, and BufferLength was less than zero, but not equal to SQL_NTS.
HY092	Invalid attribute/option identifier	The value specified for the argument Attribute was not valid for the version of ODBC supported by the driver.
HY109	Invalid cursor position	The Attribute argument was SQL_ATTR_ROW_NUMBER and the row had been deleted or could not be fetched.
HYC00	Optional feature not implemented	The value specified for the argument Attribute was a valid ODBC statement attribute for the version of ODBC supported by the driver, but was not supported by the driver.
HYT01	Connection timeout expired	The connection timeout period expired before the data source responded to the request. The connection timeout period is set through SQLSetConnectAttr, SQL_ATTR_CONNECTION_TIMEOUT.
IM001	Driver does not support this function	(DM) The driver corresponding to the StatementHandle does not support the function.

Comments

For general information about statement attributes, see Statement Attributes.

A call to SQLGetStmtAttr returns in *ValuePtr the value of the statement attribute specified in Attribute. That value can either be a 32-bit value or a null-terminated character string. If the value is a null-terminated string, the application specifies the maximum length of that string in the BufferLength argument, and the driver returns the length of that string in the *StringLengthPtr buffer. If the value is a 32-bit value, the BufferLength and StringLengthPtr arguments are not used.

To allow applications calling SQLGetStmtAttr to work with ODBC 2.x drivers, a call to SQLGetStmtAttr is mapped in the Driver Manager to SQLGetStmtOption.

The following statement attributes are read-only, so can be retrieved by SQLGetStmtAttr, but not set by SQLSetStmtAttr:

- SQL_ATTR_IMP_PARAM_DESC
- SQL_ATTR_IMP_ROW_DESC
- SQL_ATTR_ROW_NUMBER

For a list of attributes that can be set and retrieved, see SQLSetStmtAttr.

Related Functions

Function	**Description**
SQLGetConnectAttr	Returning the setting of a connection attribute
SQLSetConnectAttr	Setting a connection attribute
SQLSetStmtAttr	Setting a statement attribute

4.40 SQLGetInfo

SQLGetInfo returns general information about the Oterro Engine and the database specified by hdbc.

Syntax

RETCODE=SQLGetInfo(hdbc, fInfoType, rgbInfoValue, cbInfoValueMax, pcbInfoValue)

Arguments

Type	Argument	Use	Description
Long	hdbc	Input	The database connection handle.
Integer	fInfoType	Input	The type of information.
String	rgbInfoValue	Output	The buffer to contain information.
Integer	cbInfoValueMax	Input	The length of the rgbInfoValue buffer.
Integer	pcbInfoValue	Output	The number of bytes placed in the rgbInfoValue buffer.

Return Values

SQL_ERROR, SQL_SUCCESS, or SQL_INVALID_HANDLE

Information Types

fInfoType Constant	**Value**	**Description**
SQL_ACCESSIBLE_PROCEDURE	20	A character string that returns "Y" or "N" depending on whether a user can access all procedures listed by a call to SQLProcedures. The Oterro Engine returns "Y".
SQL_ACCESSIBLE_TABLES	19	A character string: "Y", if it is guaranteed that any tables produced by a call to SQLTables will be accessible by making the call. The Oterro Engine returns "N".
SQL_ACTIVE_CONNECTIONS	0	A 16-bit integer specifying number of active connections possible on this driver (0 signifies no limit or the limit is not known). The Oterro Engine returns 0.
SQL_ACTIVE_STATEMENTS	1	A 16-bit integer specifying number of active statements possible on this driver (0 signifies no limit or the limit is not known). The Oterro Engine returns 0.
SQL_ALTER_TABLE	81	A 32-bit bitmask enumerating the clauses supported by the data source in the ALTER TABLE command. The Oterro Engine returns the following: SQL_AT_ADD_COLUMN SQL_AT_DROP_COLUMN
SQL_BOOKMARK_PERSISTENCE	139	Indicates whether bookmarks persist through operations. The Oterro Engine returns the following: SQL_BP_DELETE SQL_BP_SCROLL SQL_BP_UPDATE
SQL_COLUMN_ALIAS	82	A character string indicating whether the data source supports column aliases. The Oterro Engine returns "N".
SQL_CONCAT_NULL_BEHAVIOR	22	A 16-bit integer denoting the results of concatenating a null with a value. When the value is returned, the driver returns 1. The driver returns 0 when a NULL is returned. The Oterro Engine returns 1.
SQL_CONVERT_FUNCTIONS	48	A 32-bit bitmask enumerating conversion functions supported by the driver. The Oterro Engine returns 0.
SQL_CONVERT_BIGINT SQL_CONVERT_BINARY SQL_CONVERT_BIT SQL_CONVERT_CHAR SQL_CONVERT_DATE SQL_CONVERT_DECIMAL SQL_CONVERT_DOUBLE SQL_CONVERT_FLOAT SQL_CONVERT_INTEGER SQL_CONVERT_LONGVARCHAR	53 54 55 56 57 58 59 60 61 62	A 32-bit bitmask indicating a set of conversions allowed for the data type named in the fInfotype. If the bitmask equals zero, conversions are not supported, including conversion to the same data type. The Oterro Engine returns 0.

SQL_CONVERT_NUMERIC SQL_CONVERT_REAL SQL_CONVERT_SMALLINT SQL_CONVERT_TIME SQL_CONVERT_TIMESTAMP SQL_CONVERT_TINYINT SQL_CONVERT_VARBINARY SQL_CONVERT_VARCHAR SQL_CONVERT_LONGVARBINARY	63 64 65 66 67 68 69 70 71	
SQL_CORRELATION_NAME	74	A 16-bit integer denoting whether table correlation names are supported. The Oterro Engine returns SQL_CN_DIFFERENT, the correlation names must be different from the table names.
SQL_CURSOR_COMMIT_BEHAVIOR	23	A 16-bit integer denoting the behavior of cursors when a commit instruction is executed. 0 Close and delete cursors, 1 Close cursors, 2 Preserve cursors at their present position. The Oterro Engine returns 1.
SQL_CURSOR_ROLLBACK_BEHAVIOR	24	The operation is ROLLBACK, the results are the same as option 23.
SQL_DATA_SOURCE_NAME	2	A string containing the drive, path, and name of the currently connected database.
SQL_DATA_SOURCE_READ_ONLY	25	The driver returns a "Y" if the database is connected in a read-only mode, or an "N" if not.
SQL_DATABASE_NAME	16	A string containing the drive, path, and name of the currently connected database.
SQL_DBMS_NAME	17	A string containing the name of the DBMS upon which the driver is running. The driver returns Oterro.
SQL_DBMS_VER	18	A string containing the version of the DBMS upon which the driver is running.
SQL_DEFAULT_TXN_ISOLATION	26	The level of transaction isolation provided by the data source. The Oterro Engine returns SQL_TXN_REPEATABLE_READ, since changes are not seen by any other users until they are committed.
SQL_DRIVER_HDBC	3	The actual HDBC implemented by the ODBC Driver Manager.
SQL_DRIVER_HENV	4	The actual HENV implemented by the ODBC Driver Manager.
SQL_DRIVER_HLIB	76	The actual library handle returned to the ODBC Driver Manager when it loaded the Oterro DLL.
SQL_DRIVER_HSTMT	5	The actual HSTMT implemented by the ODBC Driver Manager.
SQL_DRIVER_NAME	6	A string containing the name of the driver DLL.
SQL_DRIVER_ODBC_VER	77	The ODBC version supported by the driver.
SQL_DRIVER_VER	7	The driver's version string.
SQL_EXPRESSIONS_IN_ORDERBY	27	A "Y" is returned if the data source allows expressions in ORDER BY clauses. The Oterro Engine returns "N".
SQL_FETCH_DIRECTION	8	A 32-bit bitmask describing the direction in which fetches move. The Oterro Engine returns the following: SQL_FD_FETCH_NEXT SQL_FD_FETCH_FIRST SQL_FD_FETCH_LAST SQL_FD_FETCH_PRIOR SQL_FD_FETCH_ABSOLUTE SQL_FD_FETCH_RELATIVE SQL_FD_FETCH_BOOKMARK
SQL_FILE_USAGE	78	A 16-bit integer indicating how database files are handled. The Oterro Engine returns SQL_FILE_QUALIFIER, each file is a complete database.

Related Functions

Function	Description
SQLGetFunctions	Returns information about whether the Oterro Engine supports a function.

Errors

SQLSTATE	Description
01004	The data was truncated.
08003	No database has been connected.
22003	The numeric value is out of range: A significant truncation would have occurred.
S1000	An error has occurred that has no defined SQLSTATE—see the error message text.
S1001	A memory allocation failure.
S1090	An invalid string or buffer length.
S1096	An information type was out of range: fInfoType was invalid.
S1C00	The driver or data source does not support the specified type.

Visual Basic Example

SQLAPI.BAS:
Declare Function SQLGetInfo Lib "ODBC32.DLL" (ByVal hdbc&, ByVal fInfoType%, ByVal rgbInfoValue$, ByVal cbInfoValueMax%, pcbInfoValue%) As Integer

CODE:

```
Global szConnectOut As String * 512
Global cbConnectOut As Integer
Global dbstr As String
Global dbdir As String

Private Sub fdrvconn_Click()
     Dim i As Integer
     retcode = SQLAllocEnv(henv&)
     retcode = SQLAllocConnect(henv&, hdbc&)
          errorcheck retcode
     retcode = SQLDriverConnect(hdbc&, hwnd&, dbstr, SQL_NTS, szConnectOut,
255, cbConnectOut, SQL_DRIVER_COMPLETE)
     If retcode <> 0 Then
          errorcheck retcode
          GoTo lend
     End If
     'get the database path
     retcode = SQLGetInfo(hdbc&, SQL_DATABASE_NAME, szConnectOut, 512,
cbConnectOut)
          errorcheck retcode
     dbstr = Chop(szConnectOut)
     i = InStr(dbstr, "\")
     dbdir = Left$(dbstr, i)
     Do While i <> 0
          i = InStr(i + 1, dbstr, "\")
          If i <> 0 Then
               dbdir = Left$(dbstr, i)
          End If
     Loop
     retcode = SQLAllocStmt(hdbc&, hstmt&)
          errorcheck retcode
lend:
End Sub
```

4.41 SQLGetStmtOption

SQLGetStmtOption returns the current setting of a statement option.

Syntax

```
RETCODE = SQLGetStmtOption (hstmt, sSQLType, pvParam)
```

Arguments

Type	Argument	Use	Description
Long	hstmt	Input	The statement handle.
Integer	sSQLType	Input	The information option to retrieve.
Long	pvParam	Output	The value for sSQLType; it is a 32-bit integer or a pointer to a NULL terminated string.

Return Values

```
SQL_SUCCESS, SQL_ERROR, or SQL_INVALID_HANDLE
```

Information Types

Statement options that are supported by the Oterro Engine:

sSQLType Constant	Value	Return Codes
SQL_ASYNC_ENABLE	4	Specifies whether a statement is executed asyncronously.
SQL_BIND_TYPE	5	Specifies the binding orientation.
SQL_CURSOR_TYPE	6	May be one of the following: SQL_CURSOR_FORWARD_ONLY = 0 SQL_CURSOR_KEYSET_DRIVEN = 1 SQL_CURSOR_STATIC = 2 SQL_CURSOR_DYNAMIC = 3
SQL_CONCURRENCY	7	SQL_CONCUR_LOCK = 2 SQL_CONCUR_READ_ONLY = 1 SQL_CONCUR_ROWVER = 3 SQL_CONCUR_VALUES = 4
SQL_GET_BOOKMARK	21	The bookmark for the current row.
SQL_KEYSET_SIZE	8	Numbers of rows in a KEYSET.
SQL_MAX_LENGTH	3	The maximum length of data from a text column.
SQL_MAX_ROWS	1	The maximum number of rows to be returned.
SQL_NOSCAN	2	Specifies whether SQL strings are scanned for escape clauses.
SQL_QUERY_TIMEOUT	0	Number of seconds before timing out on an SQL statement.
SQL_RETRIEVE_DATA	15	Specifies whether an SQLExtendedFetch retrieves data after positioning.
SQL_ROW_NUMBER	18	The number of the current row in the result set.
SQL_ROWSET_SIZE	9	Number of rows returned.
SQL_SIMULATE_CURSOR	10	Specifies whether positioned update and delete affect only one row.

Related Functions

Function	Description
SQLExtendedFetch	Fetches one row of a result set; allows scrolling.
SQLSetStmtOption	Sets options for a statement handle.

Errors

SQLSTATE	Description
01000	A driver-specific informational message. (The function returns SQL_SUCCESS_WITH_INFO.)
24000	An invalid cursor state: A cursor is currently open on the statement handle.
IM001	The driver associated with the hstmt does not support the function.
S1000	An error has occurred that has no defined SQLSTATE—see the error message text.
S1001	A memory allocation failure.
S1092	An option type was out of range.
S1C00	The driver or data source does not support the specified type.

Visual Basic Example

SQLAPI.BAS:
Declare Function SQLGetStmtOption Lib "ODBC32.DLL" (ByVal hstmt&, ByVal sSQLType%, pvParam&) As Integer

CODE:
```
Global sqlstring As String

Private Sub ms2extfetch_Click()
     Dim cblong1 As Long
     Dim cbint1 As Integer
     Dim i As Integer
     Dim n As Integer
     sqlstring = "select int1,real1,doub1 from numbers" & vbNullChar
     retcode = SQLPrepare(hStmt&, sqlstring, SQL_NTS)
          errorcheck retcode
     retcode = SQLExecute(hStmt&)
          errorcheck retcode
     retcode = SQLSetStmtOption(hStmt&, SQL_CURSOR_TYPE,
SQL_CURSOR_DYNAMIC)
     retcode = SQLGetStmtOption(hStmt&, SQL_CURSOR_TYPE, cblong1)
     xarray1(1) = "cursor type = " & cblong1
     retcode = SQLExtendedFetch(hStmt&, SQL_FETCH_FIRST, 1, cblong1,
cbint1)
     retcode = SQLGetData(hStmt&, 1, SQL_C_CHAR, colresults, 5000,
cbcolresults)
     xarray1(2) = Chop(colresults)
     retcode = SQLExtendedFetch(hStmt&, SQL_FETCH_LAST, 1, cblong1, cbint1)
     retcode = SQLGetData(hStmt&, 1, SQL_C_CHAR, colresults, 5000,
cbcolresults)
     xarray1(3) = Chop(colresults)
     retcode = SQLExtendedFetch(hStmt&, SQL_FETCH_PRIOR, 1, cblong1,
cbint1)
     retcode = SQLGetData(hStmt&, 1, SQL_C_CHAR, colresults, 5000,
cbcolresults)
     xarray1(4) = Chop(colresults)
     view2.List1.Clear
     i = 1
     n = 1
     Do While n <> 0
          view2.List1.AddItem xarray1(i)
          i = i + 1
          n = Len(xarray1(i))
     Loop
     retcode = SQLFreeStmt(hStmt&, SQL_CLOSE)
End Sub
```

4.42 SQLGetTypeInfo

SQLGetTypeInfo retrieves information about the data types available in the database to which you are connected.

Syntax

RETCODE = SQLGetTypeInfo (hstmt, fSqlType)

Arguments

Type	Argument	Use	Description
Long	hstmt	Input	The statement handle.
Integer	fSqlType	Input	The ODBC data type, or SQL_ALL_TYPES to return information about all data types.

Return Values

SQL_SUCCESS, SQL_ERROR, or SQL_INVALID_HANDLE

Information Types

Valid SQL data types for the Oterro Engine are as follows:

Constant	Value	Oterro Data Types	Value
SQL_BINARY	-2	BIT	12
SQL_CHAR	1	TEXT	3
SQL_CURRENCY	-16	CURRENCY	6
SQL_DATE	9	DATE	4
SQL_DECIMAL	3	NUMERIC	9
SQL_DOUBLE	8	DOUBLE	7
SQL_FLOAT	6	DOUBLE	7
SQL_INTEGER	4	INTEGER	1
SQL_LONGVARBINARY	-4	VARBIT	14
SQL_LONGVARCHAR	-1	VARCHAR	11
SQL_NUMERIC	2	NUMERIC	9
SQL_REAL	7	REAL	2
SQL_SMALLINT	5	INTEGER	1
SQL_TIME	10	TIME	5
SQL_TIMESTAMP	11	DATETIME	10
SQL_VARBINARY	-3	BITNOTE	13
SQL_VARCHAR	12	NOTE	8

Result Set

Column Name	Data Type	Comments
TYPE_NAME	TEXT 8	The textual name of the ODBC data type, matches the Oterro database data type name.
DATA_TYPE	INTEGER	The integer number representing the corresponding Oterro database data type.
PRECISION	INTEGER	The maximum length allowed for the ODBC data type.
LITERAL_PREFIX	TEXT 1	The quote prefix character, for example ' (single quote), or NULL.
LITERAL_SUFFIX	TEXT 1	The quote suffix character, for example ' (single quote), or NULL.
CREATE_PARAMS	NOTE	The parameters required for CREATE, if any, otherwise NULL.
NULLABLE	INTEGER	Whether the data type accepts a NULL value: 0 = SQL_NO_NULLS if the data type does not accept NULL values 1 = SQL_NULLABLE if the data type accepts NULL values 2 = SQL_NULLABLE_UNKNOWN if it is not known whether the data type accepts NULL values The Oterro Engine returns SQL_NULLABLE_UNKNOWN.
CASE_SENSITIVE	INTEGER	Whether the data type allows case sensitivity: 1 if the data type can be case sensitive 0 if the data type cannot be case sensitive
SEARCHABLE	INTEGER	How the data type is used in a WHERE clause: 0 = SQL_UNSEARCHABLE if the data type cannot be used in a WHERE clause 1 = SQL_LIKE_ONLY if the data type can be used in a WHERE clause only with the LIKE predicate 2 = SQL_ALL_EXCEPT_LIKE if the data type can be used in a WHERE clause with all comparison operators except LIKE 3 = SQL_SEARCHABLE if the data type can be used in a WHERE clause with any comparison operator The Oterro Engine returns either 2 or 3.
UNSIGNED_ATTRIBUTE	INTEGER	1 if the data type is unsigned 0 if the data type is signed NULL is returned for non-numeric data types
MONEY	INTEGER	Whether the data type is a money data type: 1 if it is a money data type 0 if it is not a money data type Oterro database currency data type returns 1
AUTO_INCREMENT	INTEGER	Whether the data type is auto-incrementing: 1 if ODBC autonumber 0 if not ODBC autonumber The Oterro Engine returns 0. Oterro database autonumbering is different from ODBC autonumbering.
LOCAL_TYPE_NAME	TEXT 18	The localized version of the data source-dependent name of the data type. This is always NULL for the Oterro Engine.
MINIMUM_SCALE	INTEGER	The minimum scale for the specified data type.
MAXIMUM_SCALE	INTEGER	The maximum scale for the specified data type.

Related Functions

Function	Description
SQLColAttributes	Returns column attributes in a result set.
SQLDescribeCol	Describes a column in a result set.

Errors

SQLSTATE	Description
24000	An invalid cursor state: A cursor is currently open on the statement handle.
S1000	An error has occurred that has no defined SQLSTATE—see the error message text.
S1001	A memory allocation failure.
S1004	An SQL data type is out of range: An invalid fSqlType was specified.

Visual Basic Example

SQLAPI.BAS:
Declare Function SQLGetTypeInfo Lib "ODBC32.DLL" (ByVal hstmt&, ByVal fSqlType%) As Integer

CODE:
```
Global colresults As String * 5000
Global cbcolresults As Long
Global colnum As Integer

Private Sub mdb1type_Click()
     retcode = SQLGetTypeInfo(hStmt&, SQL_ALL_TYPES)
          errorcheck retcode
     loadtest
End Sub


Sub loadtest()
     Dim i As Integer
     Dim cblong1 As Long
     Dim cbint1 As Integer
     i = 0
     retcode = SQLNumResultCols(hStmt&, colnum)
          errorcheck retcode
     i = 1
     Do While SQLFetch(hStmt&) = SQL_SUCCESS
          Do While i <= colnum
               retcode = SQLGetData(hStmt&, i, SQL_C_CHAR, colresults,
5000, cbcolresults)
               view1.text1.Text = view1.text1.Text & vbCrLf & "Col" & i &
": " Chop(colresults)
               i = i + 1
          Loop
          i = 1
     Loop
     retcode = SQLFreeStmt(hStmt&, SQL_CLOSE)
End Sub
```

4.43 SQLMoreResults

Note: This function is normally used in conjunction with SQLSetPos. Since Visual Basic does not support SQLSetPos, this function is included here with the syntax for using the C or C++ programming language.

SQLMoreResults determines if there are more results available on an hstmt containing SELECT, UPDATE, INSERT, or DELETE statements and initializes processing for those results.

Syntax

```
RETCODE PASCAL SQLMoreResults (hstmt)
```

Arguments

Type	Argument	Use	Description
HSTMT	hstmt	Input	The statement handle.

Return Values

```
SQL_SUCCESS, SQL_SUCCESS_WITH_INFO, SQL_STILL_EXECUTING,
SQL_NO_DATA_FOUND, SQL_ERROR, or SQL_INVALID_HANDLE.
```

Comments

If another result set or count is available, SQLMoreResults returns SQL_SUCCESS and initializes the result set or count for more processing.

If all results have been processed, SQLMoreResults returns SQL_NO_DATA_FOUND.

4.44 SQLNativeSql

SQLNativeSql returns the SQL string as translated by the driver.

Syntax

```
RETCODE = SQLNativeSql(hdbc, pucSQLStr, lSQLStrLen, pucSQLStrOut, lSQLStrOutMax,
plSQLStrOut)
```

Arguments

Type	Argument	Use	Description
Long	hdbc	Input	The database connection handle.
String	pucSQLStr	Input	The SQL string to be translated.
Long	lSQLStrLen	Input	The length of the text string.
String	pucSQLStrOut	Output	Pointer to storage for the translated string.
Long	lSQLStrOutMax	Input	Maximum length of the pucSQLStrOut buffer.
Long	plSQLStrOut	Output	The total number of bytes available to return.

Return Values

```
SQL_SUCCESS, SQL_SUCCESS_WITH_INFO, SQL_ERROR, or SQL_INVALID_HANDLE
```

Errors

SQLSTATE	Description
01000	A driver-specific informational message. (Function returns SQL_SUCCESS_WITH_INFO.)

Copyright © 1982-2025 R:BASE Technologies, Inc.

01004	The data was truncated. (SQL_SUCCESS_WITH_INFO was returned.)
08003	No database has been connected.
37000	A syntax error or access violation.
IM001	The driver does not support this function.
S1000	An error has occurred that has no defined SQLSTATE—see the error message text.
S1001	A memory allocation failure.
S1009	An invalid argument value—a null pointer was passed.
S1090	An invalid string or buffer length.

Visual Basic Example

SQLAPI.BAS:
Declare Function SQLNativeSql Lib "ODBC32.DLL" (ByVal hdbc&, ByVal pucSQLStr$, ByVal lSQLStrLen&, ByVal pucSQLStrOut$, ByVal lSQLStrOutMax&, plSQLStrOut&) As Integer

CODE:
```
Private Sub ms2native_Click()
     Dim sqlstring as String
     Dim cblong1 As Long
     Dim nativesql As String * 512
     sqlstring = "SELECT * FROM numbers" & vbNullChar
     retcode = SQLNativeSql(hdbc&, sqlstring, SQL_NTS, nativesql, 512,
cblong1)
          errorcheck retcode
     bufstring = InputBox(sqlstring, "SQLNativeSql", nativesql)
     retcode = SQLCancel(hStmt&)
End Sub
```

4.45 SQLNumParams

Note: This function is normally used in conjunction with SQLBindParameter. Since Visual Basic does not support SQLBindParameter, this function is included here with the syntax for using the C or C++ programming language.

SQLNumParams returns the number of parameters in an SQL statement.

Syntax

RETCODE = SQLNumParams (hstmt, psNumParams)

Arguments

Type	Argument	Use	Description
HSTMT	hstmt	Input	The statement handle.
SWORD FAR*	psNumParams	Output	The number of parameters in the statement.

Return Values

SQL_SUCCESS, SQL_SUCCESS_WITH_INFO, SQL_STILL_EXECUTING, SQL_ERROR, or SQL_INVALID_HANDLE

Comments

SQLNumParams can only be called after SQLPrepare has been called.

If the statement associated with hstmt does not contain parameters, SQLNumParams sets psNumParams to 0.

4.46 SQLNumResultCols

SQLNumResultCols sets the value of the argument pcCol to the number of columns in the result set associated with the statement handle hstmt.

Syntax

```
RETCODE = SQLNumResultCols (hstmt, pcCol)
```

Arguments

Type	Argument	Use	Description
Long	hstmt	Input	The statement handle.
Integer	pcCol	Output	The number of columns in the result set.

Return Values

```
SQL_SUCCESS, SQL_ERROR, or SQL_INVALID_HANDLE
```

Comments

The argument pcCol is set to zero when the hstmt is associated with a statement that does not return a result set, for example, an UPDATE or INSERT command.

Related Functions

Function	Description
SQLColAttributes	Returns column attributes in a result set.
SQLDescribeCol	Describes a column in a result set.
SQLExtendedFetch	Fetches one row of a result set; allows scrolling.
SQLFetch	Fetches one row of a result set.
SQLGetData	Gets result data for a column in a result set.

Errors

SQLSTATE	Description
S1000	An error has occurred that has no defined SQLSTATE—see the error message text.
S1001	A memory allocation failure.
S1002	An invalid column number.

Visual Basic Example

SQLAPI.BAS:
Declare Function SQLNumResultCols Lib "ODBC32.DLL" (ByVal hstmt&, pcCol%) As Integer

CODE:
```
Global colnum As Integer
Global bufstring As String
Global cbcol1 As Long
Global colresults As String * 5000
Global starttime As Date
Global endtime As Date
Global elapsed As Integer

Sub getall()
     Dim i As Integer
     retcode = SQLNumResultCols(hstmt, colnum)
          errorcheck retcode
     bufstring = sql1.Text
     results.AddItem UCase(bufstring)
     starttime = Time
     Do While SQLFetch(hstmt&) = SQL_SUCCESS
          i = 1
          Do While i <= colnum
               retcode = SQLGetData(hstmt&, i, SQL_C_CHAR, colresults,
5000, cbcol1)
                     errorcheck retcode
               If i = 1 Then
                     bufstring = Chop(colresults)
               Else
                     bufstring = bufstring & "," & Chop(colresults)
               End If
               i = i + 1
          Loop
          results.AddItem bufstring
     Loop
     endtime = Time
     elapsed = DateDiff("s", starttime, endtime)
     bufstring = "Elapsed Time:   " & elapsed & "   seconds"
     results.AddItem bufstring
     retcode = SQLFreeStmt(hstmt&, SQL_CLOSE)
End Sub
```

4.47 SQLParamOptions

Note: This function is normally used in conjunction with SQLBindParameter. Since Visual Basic does not support SQLBindParameter, this function is included here with the syntax for using the C or C++ programming language.

SQLParamOptions allows an application to specify multiple values for the set of parameters assigned by SQLBindParameter.

Syntax

`RETCODE PASCAL` SQLParamOptions (hstmt, ulSetSize, pulRow)

Arguments

Type	Argument	Use	Description
HSTMT	hstmt	Input	The statement handle.
UDWORD	ulSetSize	Input	The number of values for each parameter.
UDWORD FAR *	pulRow	Input	A pointer to storage for the current row number.

Return Values

```
SQL_SUCCESS, SQL_SUCCESS_WITH_INFO, SQL_ERROR, or SQL_INVALID_HANDLE
```

Comments

If SQLExecute or SQLExecDirect returns an error when trying to execute a parameterized query, the value in pulRow returns which row of parameters failed.

4.48 SQLPrepare

SQLPrepare prepares the statement that will be executed later by the function SQLExecute.

Syntax

```
RETCODE = SQLPrepare (hstmt, szSqlStr, cbSqlStr)
```

Arguments

Type	Argument	Use	Description
Long	hstmt	Input	The statement handle.
String	szSqlStr	Input	The SQL string.
Long	cbSqlStr	Input	The length of the SQL statement.

Return Values

```
SQL_SUCCESS, SQL_SUCCESS_WITH_INFO, SQL_ERROR, or SQL_INVALID_HANDLE
```

Comments

For a list of the available Oterro database commands, see "How to Use the Oterro Engine".

Related Functions

Function	Description
SQLAllocStmt	Allocates a new statement handle.
SQLExecDirect	Executes an SQL statement.
SQLExecute	Executes a prepared SQL statement.
SQLGetCursorName	Gets the name of the cursor associated with a statement handle.
SQLGetData	Gets result data for a column in a result set.
SQLSetCursorName	Sets a cursor name for a statement handle.

Errors

SQLSTATE	Description
S1000	An error has occurred that has no defined SQLSTATE—see the error message text.
S1001	A memory allocation failure.
S1009	An invalid argument value—a null pointer was passed.
S1090	An invalid string or buffer length.

Visual Basic Example

SQLAPI.BAS:
Declare Function SQLPrepare Lib "ODBC32.DLL" (ByVal hstmt&, ByVal szSqlStr$, ByVal cbSqlStr&) As Integer

CODE:

```
Global sqlstring As String

Private Sub ms2extfetch_Click()
     Dim cblong1 As Long
     Dim cbint1 As Integer
     Dim i As Integer
     Dim n As Integer
     sqlstring = "select int1,real1,doub1 from numbers" & vbNullChar
     retcode = SQLPrepare(hStmt&, sqlstring, SQL_NTS)
          errorcheck retcode
     retcode = SQLExecute(hStmt&)
          errorcheck retcode
     retcode = SQLSetStmtOption(hStmt&, SQL_CURSOR_TYPE,
SQL_CURSOR_DYNAMIC)
     retcode = SQLGetStmtOption(hStmt&, SQL_CURSOR_TYPE, cblong1)
     xarray1(1) = "cursor type = " & cblong1
     retcode = SQLExtendedFetch(hStmt&, SQL_FETCH_FIRST, 1, cblong1,
cbint1)
     retcode = SQLGetData(hStmt&, 1, SQL_C_CHAR, colresults, 5000,
cbcolresults)
     xarray1(2) = Chop(colresults)
     retcode = SQLExtendedFetch(hStmt&, SQL_FETCH_LAST, 1, cblong1, cbint1)
     retcode = SQLGetData(hStmt&, 1, SQL_C_CHAR, colresults, 5000,
cbcolresults)
     xarray1(3) = Chop(colresults)
     retcode = SQLExtendedFetch(hStmt&, SQL_FETCH_PRIOR, 1, cblong1,
cbint1)
     retcode = SQLGetData(hStmt&, 1, SQL_C_CHAR, colresults, 5000,
cbcolresults)
     xarray1(4) = Chop(colresults)
     view2.List1.Clear
     i = 1
     n = 1
     Do While n <> 0
          view2.List1.AddItem xarray1(i)
          i = i + 1
          n = Len(xarray1(i))
     Loop
     retcode = SQLFreeStmt(hStmt&, SQL_CLOSE)
End Sub
```

4.49 SQLPrimaryKeys

SQLPrimaryKeys returns the primary key column(s) for a given table.

Syntax

RETCODE = SQLPrimaryKeys (hstmt, szPkTableQualifier, cbPkTableQualifier, szPkTableOwner, cbPkTableOwner, szPkTableName, cbPkTableName)

Arguments

Type	Argument	Use	Description
Long	hstmt	Input	The statement handle.
String	szPkTableQualifier	Input	The buffer containing the table qualifier.
Integer	cbPkTableQualifier	Input	The length of the table qualifier.
String	szPkTableOwner	Input	The buffer containing the table-owner name.
Integer	cbPkTableOwner	Input	The length of the table-owner name.
String	szPkTableName	Input	The buffer containing the table name.
integer	cbPkTableName	Input	The length of the table name.

Return Values

`SQL_SUCCESS, SQL_ERROR, or SQL_INVALID_HANDLE`

Result Set

Column Name	Data Type	Comments
TABLE_QUALIFIER	TEXT 18	The primary key table qualifier. The Oterro Engine returns NULL.
TABLE_OWNER	TEXT 18	The primary key table owner. The Oterro Engine returns NULL.
TABLE_NAME	TEXT 18	The primary key table name.
COLUMN_NAME	TEXT 18	The primary key column name.
KEY_SEQ	INTEGER	The column-sequence number in the key (starting with 1).
PK_NAME	TEXT 18	The primary key name.

The lengths of text columns shown in the table are maximums; to determine the actual lengths, use the SQLGetInfo function.

Comments

One row is returned for each column defined as a primary key.

Related Functions

Function	Description
SQLForeignKeys	Returns the columns defined as foreign keys.
SQLSpecialColumns	Returns information about a set of columns.
SQLStatistics	Returns statistics for tables and indexes.

Errors

SQLSTATE	Description
01000	A driver-specific informational message. (The function returns SQL_SUCCESS_WITH_INFO.)
08S01	The data source connection failed before the function completed processing.
24000	An invalid cursor state: A cursor is currently open on the statement handle.
IM001	The driver associated with the hstmt does not support the function.
S1000	An error has occurred that has no defined SQLSTATE—see the error message text.
S1001	A memory allocation failure.
S1090	An invalid string or buffer length.
S1C00	The driver or data source does not support the specified type.

Visual Basic Example

SQLAPI.BAS:
Declare Function SQLPrimaryKeys Lib "ODBC32.DLL" (ByVal hstmt&, ByVal szPkTableQualifier$, ByVal cbPkTableQualifier%, ByVal szPkTableOwner$, ByVal cbPkTableOwner%, ByVal szPkTableName$, ByVal cbPkTableName%) As Integer

CODE:
```
Global colnum As Integer
Global szTableName As String * 20
Global cbTableName As Integer
Global szFirst As String * 1500
Global cbFirst As Long

Sub pkconstr()
     Static xnum As Integer
     retcode = SQLPrimaryKeys(hstmt&, "", 0, "", 0, szTableName,
cbTableName)
          errorcheck retcode
     xnum = 1
     load3grid xnum
End Sub

Sub load3grid(xnum As Integer)
     Dim i As Integer
     Dim n As Integer
     n = xnum
     i = 1
     retcode = SQLNumResultCols(hstmt&, colnum)
          errorcheck retcode
     Do While SQLFetch(hstmt&) = SQL_SUCCESS
          dbstr1.Grid3.Row = n
          Do While i <= colnum
               retcode = SQLGetData(hstmt&, i, SQL_C_CHAR, szFirst, 255,
cbFirst)
               dbstr1.Grid3.Col = i
               dbstr1.Grid3.Text = Chop(szFirst)
               i = i + 1
          Loop
          n = n + 1
          i = 1
     Loop
     xnum = n
     retcode = SQLFreeStmt(hstmt&, SQL_CLOSE)
End Sub
```

4.50 SQLProcedureColumns

SQLProcedureColumns returns the list of input and output parameters, as well as the columns that make up the result set for the specified procedures. The driver returns the information as a result set on the specified hStmt.

Syntax

RETCODE = SQLProcedureColumns (hStmt, szProcQualifier, cbProcQualifier, szProcOwner, cbProcOwner, szProcName, cbProcName, szColumnName, cbColumnName)

Arguments

Type	Argument	Use	Description
Long	hStmt	Input	The statement handle.
String	szProcQualifier	Input	The procedure qualifier name.
Integer	cbProcQualifier	Input	The length of procedure qualifier name.
String	szProcOwner	Input	The procedure owner name.
Integer	cbProcOwner	Input	The length of the procedure owner name.
String	szProcName	Input	The procedure name.
Integer	cbProcName	Input	The length of the procedure name.
String	szColumnName	Input	The column name.
Integer	cbColumnName	Input	The length of the column name.

Return Values

SQL_SUCCESS, SQL_SUCCESS_WITH_INFO, SQL_STILL_EXECUTING, SQL_ERROR, or SQL_INVALID_HANDLE

Result Set

Column Name	Data Type	Comments
PROCEDURE_QUALIFIER	TEXT 18	The procedure qualifier name. Always NULL in the Oterro Engine.
PROCEDURE_OWNER	TEXT 18	The procedure owner name. Always NULL in the Oterro Engine.
PROCEDURE_NAME	TEXT 18	The procedure name.
COLUMN_NAME	TEXT 18	The procedure column name.
COLUMN_TYPE	INTEGER	Returns one of the following: 0 = SQL_PARAM_TYPE_UNKNOWN The procedure column is a parameter whose type is unknown. 1 = SQL_PARAM_INPUT The procedure column is an input parameter. 2 = SQL_PARAM_INPUT_OUTPUT The procedure column is an input/output parameter. 3 = SQL_RESULT_COL The procedure column is a result set column. 4 = SQL_PARAM_OUTPUT The procedure column is an output parameter. 5 = SQL_RETURN_VALUE The procedure column is the return value of the procedure.

DATA_TYPE	INTEGER	The number representing the Oterro database data type.
TYPE_NAME	TEXT 18	The Oterro database data type name.
PRECISION	INTEGER	The precision of the procedure column.
LENGTH	INTEGER	The length in bytes of data transferred.
SCALE	INTEGER	The scale of the procedure column.
RADIX	INTEGER	The base of the number system used in the database: 10 for all numeric data types (INTEGER, DOUBLE, etc.) NULL for all non-numeric data types (TEXT, DATE, etc.)
NULLABLE	INTEGER	Returns one of the following: 0 = SQL_NO_NULLS The procedure column does not accept NULL values. 1 = SQL_NULLABLE The procedure column accepts NULL values. 2 = SQL_NULLABLE_UNKNOWN Not known if the procedure column accepts NULL values.
REMARKS	NOTE	A description of the procedure column.

Related Functions

Function	**Description**
SQLExtendedFetch	Fetches one row of a result set; allows scrolling.
SQLFetch	Fetches one row of a result set.
SQLProcedures	Returns the list of procedure names in the database.

Errors

SQLSTATE	Description
01000	A driver-specific informational message. (Function returns SQL_SUCCESS_WITH_INFO.)
08S01	A communication link failure.
24000	An invalid cursor state.
IM001	The driver does not support this function.
S1000	An error has occurred that has no defined SQLSTATE —see the error message text.
S1001	A memory allocation failure.
S1008	The operation was canceled.
S1010	A function sequence error occurred.
S1090	An invalid string or buffer length.
S1C00	The driver or data source does not support the specified type.
S1T00	The timeout period expired before the data source returned the result.

Visual Basic Example

SQLAPI.BAS:
Declare Function SQLProcedureColumns Lib "ODBC32.DLL" (ByVal hstmt&, ByVal szProcQualifier$, ByVal cbProcQualifier%, ByVal szProcOwner$, ByVal cbProcOwner%, ByVal szProcName$, ByVal cbProcName%, ByVal szColumnName$, ByVal cbColumnName%) As Integer

CODE:
```
Global colresults As String * 5000
Global cbcolresults As Long
Global colnum As Integer

Private Sub mdb1cpriv_Click()
     'this will return all columns for the proc1 procedure
     retcode = SQLProcedureColumns(hStmt&, "", 0, "", 0, "proc1", 5,"",0)
```

```
          errorcheck retcode
     loadtest
End Sub

Sub loadtest()
     Dim i As Integer
     Dim cblong1 As Long
     Dim cbint1 As Integer
     i = 0
     retcode = SQLNumResultCols(hStmt&, colnum)
          errorcheck retcode
     i = 1
     Do While SQLFetch(hStmt&) = SQL_SUCCESS
          Do While i <= colnum
               retcode = SQLGetData(hStmt&, i, SQL_C_CHAR, colresults,
5000, cbcolresults)
               view1.text1.Text = view1.text1.Text & vbCrLf & "Col" & i &
": " Chop(colresults)
               i = i + 1
          Loop
          i = 1
     Loop
     retcode = SQLFreeStmt(hStmt&, SQL_CLOSE)
End Sub
```

4.51 SQLProcedures

SQLProcedures returns the list of procedure names stored in a specific data source.

Syntax

`RETCODE` = SQLProcedures (hstmt, szProcQualifier, cbProcQualifier, szProcOwner, cbProcOwner, szProcName, cbProcName)

Arguments

Type	Argument	Use	Description
Long	hstmt	Input	The statement handle.
String	szProcQualifier	Input	The procedure qualifier.
Integer	cbProcQualifier	Input	The length of the procedure qualifier.
String	szProcOwner	Input	The procedure owner name.
Integer	cbProcOwner	Input	The length of the procedure owner name.
String	szProcName	Input	The procedure name.
Integer	cbProcName	Input	The length of the procedure name.

Return Values

`SQL_SUCCESS, SQL_SUCCESS_WITH_INFO, SQL_STILL_EXECUTING, SQL_ERROR, SQL_INVALID_HANDLE.`

Result Set

Column Name	Data Type	Comments
PROCEDURE_QUALIFIER	TEXT 18	The procedure qualifier name. Always NULL in the Oterro Engine.
PROCEDURE_OWNER	TEXT 18	The procedure owner name. Always NULL in the Oterro Engine.
PROCEDURE_NAME	TEXT 18	The procedure name.
NUM_INPUT_PARAMS	INTEGER	Reserved for future use.
NUM_OUTPUT_PARAMS	INTEGER	Reserved for future use.
NUM_RESULT_SETS	INTEGER	Reserved for future use.
REMARKS	TEXT 254	A description of the procedure.
PROCEDURE_TYPE	INTEGER	Returns one of the following: 0 = SQL_PT_UNKNOWN when it cannot be determined if the procedure returns a value. 1 = SQL_PT_PROCEDURE when the returned object is a procedure. 2 = SQL_PT_FUNCTION when the returned object is a function. The Oterro Engine returns 0.

Related Functions

Function	Description
SQLCancel	Ends processing of a statement.
SQLExtendedFetch	Fetches one row of a result set; allows scrolling.
SQLFetch	Fetches one row of a result set.
SQLGetInfo	Queries information about a driver or data source.
SQLProcedureColumns	Returns the columns for the procedures.

Errors

SQLSTATE	Description
01000	A driver-specific informational message. (Function returns SQL_SUCCESS_WITH_INFO)
08S01	A communication link failure.
24000	An invalid cursor state.
IM001	The driver does not support this function.
S1000	An error has occurred that has no defined SQLSTATE—see the error message text.
S1001	A memory allocation failure.
S1008	The operation was canceled.
S1010	A function sequence error occurred.
S1090	An invalid string or buffer length.
S1C00	The driver or data source does not support the specified type.
S1T00	The timeout period expired before the data source returned the requested results.

Visual Basic Example

SQLAPI.BAS:
```
Declare Function SQLProcedures Lib "ODBC32.DLL" (ByVal hstmt&, ByVal  szProcQualifier$, ByVal cbProcQualifier%, ByVal szProcOwner$, ByVal  cbProcOwner%, ByVal szProcName$, ByVal cbProcName%) As Integer
```

```
CODE:
Global colresults As String * 5000
Global cbcolresults As Long
Global colnum As Integer

Private Sub mdb1cpriv_Click()
     'this will return all procedures for the database
     retcode = SQLProcedures(hStmt&, "", 0, "", 0, "", 0)
          errorcheck retcode
     loadtest
End Sub

Sub loadtest()
     Dim i As Integer
     Dim cblong1 As Long
     Dim cbint1 As Integer
     i = 0
     retcode = SQLNumResultCols(hStmt&, colnum)
          errorcheck retcode
     i = 1
     Do While SQLFetch(hStmt&) = SQL_SUCCESS
          Do While i <= colnum
               retcode = SQLGetData(hStmt&, i, SQL_C_CHAR, colresults,
5000, cbcolresults)
               view1.text1.Text = view1.text1.Text & vbCrLf & "Col" & i &
": " Chop(colresults)
               i = i + 1
          Loop
          i = 1
     Loop
     retcode = SQLFreeStmt(hStmt&, SQL_CLOSE)
End Sub
```

4.52 SQLPutData

Note: This function is not used in Visual Basic applications. This function cannot be called from Visual Basic because it uses a pointer to a data structure as an input argument but does not use that pointer immediately. Since Visual Basic moves data around, the pointers would become invalid. It is included here with the syntax for using the C and C++ programming languages.

SQLPutData places the data in the statement after SQLBindParameter sets up the parameters for the statement.

Syntax

`RETCODE PASCAL` SQLPutData (hstmt, rgbValue, cbValue)

Arguments

Type	Argument	Use	Description
HSTMT	hstmt	Input	The statement handle.
PTR	rgbValue	Input	A pointer to storage for the data.

Copyright © 1982-2025 R:BASE Technologies, Inc.

SDWORD	cbValue	Input	The length of the rgbValue area, or either SQL_NULL_DATA or SQL_DEFAULT_PARAM.

Return Values

SQL_SUCCESS, SQL_ERROR, or SQL_INVALID_HANDLE

4.53 SQLRowCount

SQLRowCount returns the number of rows that were affected by the update, insert, or delete in the given hstmt, and sets pcrow to the number of rows affected.

Syntax

```
RETCODE = SQLRowCount (hstmt, pcrow)
```

Arguments

Type	Argument	Use	Description
Long	hstmt	Input	The statement handle.
Long	pcrow	Output	The number of rows affected.

Return Values

SQL_SUCCESS, SQL_ERROR, SQL_SUCCESS_WITH_INFO, or SQL_INVALID_HANDLE

Comments

SQLRowCount sets the pcrow argument to -1 when it is called before an update, insert, or delete statement has been executed for this hstmt, or when the last statement executed by this hstmt was not an update, insert, or delete.

Related Functions

Function	Description
SQLExecDirect	Executes an SQL statement.
SQLExecute	Executes a prepared SQL statement.

Errors

SQLSTATE	Description
S1000	An error has occurred that has no defined SQLSTATE—see the error message text.
S1001	A memory allocation failure.

Visual Basic Example

```
SQLAPI.BAS:
Declare Function SQLRowCount Lib "ODBC32.DLL" (ByVal hstmt&, pcrow&) As Integer
```

CODE:
```
Private Sub mt1transact_Click()
     Dim cblong1 As Long
     Dim sqlstring as String
     sqlstring = "update numbers set int1 = 45 where int1 = 99" &
vbNullChar
     retcode = SQLExecDirect(hStmt&, sqlstring, SQL_NTS)
          errorcheck retcode
     retcode = SQLRowCount(hStmt&, cblong1)
          errorcheck retcode
     If cblong1 = 1 Then
          retcode = SQLTransact(hEnv&, hdbc&, SQL_COMMIT)
               errorcheck retcode
     Else
          retcode = SQLTransact(hEnv&, hdbc&, SQL_ROLLBACK)
               errorcheck retcode
     End If
     retcode = SQLCancel(hStmt&)
End Sub
```

4.54 SQLSetConnectAttr

SQLSetConnectAttr sets attributes that govern aspects of connections.

Oterro calls SQLSetConnectAttr with the SQL_ATTR_ODBC_CURSORS attribute to specify whether the cursor library is always used. Oterro supports scrollable cursors and if it returns SQL_CA1_RELATIVE for the SQL_STATIC_CURSOR_ATTRIBUTES1 information type in SQLGetInfo, Oterro calls SQLSetConnectAttr to specify the cursor library usage after it calls SQLAllocHandle with a HandleType of SQL_HANDLE_DBC to allocate the connection and before it connects to the data source. If Oterro calls SQLSetConnectAttr with the SQL_ATTR_ODBC_CURSORS attribute while the connection is still active, the cursor library returns an error.

To set a statement attribute supported by the cursor library for all statements associated with a connection, Oterro calls SQLSetConnectAttr for that statement attribute after it connects to the data source and before it opens the cursor. If Oterro calls SQLSetConnectAttr with a statement attribute and a cursor is open on a statement associated with the connection, the statement attribute will not be applied to that statement until the cursor is closed and reopened.

Syntax

RETCODE = SQLSetConnectAttr(ConnectionHandle,Attribute,ValuePtr,StringLength)

Arguments

Type	Argument	Use	Description
Long	ConnectionHandle	Input	Connection handle.
Integer	Attribute	Input	Attribute to set, listed in "Comments."
Long	ValuePtr	Input	Pointer to the value to be associated with Attribute. Depending on the value of Attribute, ValuePtr will be a 32-bit unsigned integer value or will point to a null-terminated character string. Note that if the Attribute argument is a driver-specific value, the value in ValuePtr may be a signed integer.
Integer	StringLength	Input	If Attribute is an ODBC-defined attribute and ValuePtr points to a character string or a binary buffer, this argument should be the length of *ValuePtr. For character string data, this argument should contain the number of bytes in the string.

			If Attribute is an ODBC-defined attribute and ValuePtr is an integer, StringLength is ignored. If Attribute is a driver-defined attribute, the application indicates the nature of the attribute to the Driver Manager by setting the StringLength argument. StringLength can have the following values: If ValuePtr is a pointer to a character string, then StringLength is the length of the string or SQL_NTS. If ValuePtr is a pointer to a binary buffer, then the application places the result of the SQL_LEN_BINARY_ATTR(length) macro in StringLength. This places a negative value in StringLength. If ValuePtr is a pointer to a value other than a character string or a binary string, then StringLength should have the value SQL_IS_POINTER. If ValuePtr contains a fixed-length value, then StringLength is either SQL_IS_INTEGER or SQL_IS_UINTEGER, as appropriate.

Return Values

```
SQL_SUCCESS, SQL_SUCCESS_WITH_INFO, SQL_ERROR, or SQL_INVALID_HANDLE
```

Errors

When SQLSetConnectAttr returns SQL_ERROR or SQL_SUCCESS_WITH_INFO, an associated SQLSTATE value can be obtained by calling SQLGetDiagRec with a HandleType of SQL_HANDLE_DBC and a Handle of ConnectionHandle. The following table lists the SQLSTATE values commonly returned by SQLSetConnectAttr and explains each one in the context of this function; the notation "(DM)" precedes the descriptions of SQLSTATEs returned by the Driver Manager. The return code associated with each SQLSTATE value is SQL_ERROR, unless noted otherwise.

The driver can return SQL_SUCCESS_WITH_INFO to provide information about the result of setting an option.

SQLSTATE	Error	Description
01000	General warning	Driver-specific informational message. (Function returns SQL_SUCCESS_WITH_INFO.)
01S02	Option value changed	The driver did not support the value specified in ValuePtr and substituted a similar value. (Function returns SQL_SUCCESS_WITH_INFO.)
08002	Connection name in use	The Attribute argument was SQL_ATTR_ODBC_CURSORS, and the driver was already connected to the data source.
08003	Connection does not exist	(DM) An Attribute value was specified that required an open connection, but the ConnectionHandle was not in a connected state.
08S01	Communication link failure	The communication link between the driver and the data source to which the driver was connected failed before the function completed processing.
24000	Invalid cursor state	The Attribute argument was SQL_ATTR_CURRENT_CATALOG, and a result set was pending.
3D000	Invalid catalog name	The Attribute argument was SQL_CURRENT_CATALOG, and the specified catalog name was invalid.
HY000	General error	An error occurred for which there was no specific SQLSTATE and for which no implementation-specific SQLSTATE was defined. The error message returned by SQLGetDiagRec in the *MessageText buffer describes the error and its cause.
HY001	Memory allocation error	The driver was unable to allocate memory required to support execution or completion of the function.

HY009	Invalid use of null pointer	The Attribute argument identified a connection attribute that required a string value, and the ValuePtr argument was a null pointer.
HY010	Function sequence error	(DM) An asynchronously executing function was called for a StatementHandle associated with the ConnectionHandle and was still executing when SQLSetConnectAttr was called. (DM) SQLExecute, SQLExecDirect, SQLBulkOperations, or SQLSetPos was called for a StatementHandle associated with the ConnectionHandle and returned SQL_NEED_DATA. This function was called before data was sent for all data-at-execution parameters or columns. (DM) SQLBrowseConnect was called for the ConnectionHandle and returned SQL_NEED_DATA. This function was called before SQLBrowseConnect returned SQL_SUCCESS_WITH_INFO or SQL_SUCCESS.
HY011	Attribute cannot be set now	The Attribute argument was SQL_ATTR_TXN_ISOLATION, and a transaction was open.
HY013	Memory management error	The function call could not be processed because the underlying memory objects could not be accessed, possibly because of low memory conditions.
HY024	Invalid attribute value	Given the specified Attribute value, an invalid value was specified in ValuePtr. (The Driver Manager returns this SQLSTATE only for connection and statement attributes that accept a discrete set of values, such as SQL_ATTR_ACCESS_MODE or SQL_ATTR_ASYNC_ENABLE. For all other connection and statement attributes, the driver must verify the value specified in ValuePtr.) The Attribute argument was SQL_ATTR_TRACEFILE or SQL_ATTR_TRANSLATE_LIB, and ValuePtr was an empty string.
HY090	Invalid string or buffer length	(DM) *ValuePtr is a character string, and the StringLength argument was less than 0 but was not SQL_NTS.
HY092	Invalid attribute/option identifier	(DM) The value specified for the argument Attribute was not valid for the version of ODBC supported by the driver. (DM) The value specified for the argument Attribute was a read-only attribute.
HYC00	Optional feature not implemented	The value specified for the argument Attribute was a valid ODBC connection or statement attribute for the version of ODBC supported by the driver but was not supported by the driver.
HYT01	Connection timeout expired	The connection timeout period expired before the data source responded to the request. The connection timeout period is set through SQLSetConnectAttr, SQL_ATTR_CONNECTION_TIMEOUT.
IM001	Driver does not support this function	(DM) The driver associated with the ConnectionHandle does not support the function.
IM009	Unable to load translation DLL	The driver was unable to load the translation DLL that was specified for the connection. This error can be returned only when Attribute is SQL_ATTR_TRANSLATE_LIB.

Comments

For general information about connection attributes, see Connection Attributes.

The currently defined attributes and the version of ODBC in which they were introduced are shown in the table later in this section; it is expected that more attributes will be defined to take advantage of different data sources. A range of attributes is reserved by ODBC; driver developers must reserve values for their own driver-specific use from Open Group.

Note: The ability to set statement attributes at the connection level by calling SQLSetConnectAttr has been deprecated in ODBC 3.x. ODBC 3.x applications should never set statement attributes at the connection level. ODBC 3.x statement attributes cannot be set at the connection level, with the exception of the SQL_ATTR_METADATA_ID and SQL_ATTR_ASYNC_ENABLE attributes, which are both connection attributes and statement attributes and can be set at either the connection level or the statement level.

An application can call SQLSetConnectAttr at any time between the time the connection is allocated and freed. All connection and statement attributes successfully set by the application for the connection persist until SQLFreeHandle is called on the connection. For example, if an application calls SQLSetConnectAttr before connecting to a data source, the attribute persists even if SQLSetConnectAttr fails in the driver when the application connects to the data source; if an application sets a driver-specific attribute, the attribute persists even if the application connects to a different driver on the connection.

Some connection attributes can be set only before a connection has been made; others can be set only after a connection has been made. The following table indicates those connection attributes that must be set either before or after a connection has been made. Either indicates that the attribute can be set either before or after connection.

Attribute	**Set before or after connection?**
SQL_ATTR_ACCESS_MODE	Either**[1]**
SQL_ATTR_ASYNC_ENABLE	Either**[2]**
SQL_ATTR_AUTOCOMMIT	Either
SQL_ATTR_CONNECTION_TIMEOUT	Either
SQL_ATTR_CURRENT_CATALOG	Either**[1]**
SQL_ATTR_LOGIN_TIMEOUT	Before
SQL_ATTR_METADATA_ID	Either
SQL_ATTR_ODBC_CURSORS	Before
SQL_ATTR_PACKET_SIZE	Before
SQL_ATTR_QUIET_MODE	Either
SQL_ATTR_TRACE	Either
SQL_ATTR_TRACEFILE	Either
SQL_ATTR_TRANSLATE_LIB	After
SQL_ATTR_TRANSLATE_OPTION	After
SQL_ATTR_TXN_ISOLATION	Either[3]

[1] SQL_ATTR_ACCESS_MODE and SQL_ATTR_CURRENT_CATALOG can be set before or after connecting, depending on the driver. However, interoperable applications set them before connecting because some drivers do not support changing these after connecting.

[2] SQL_ATTR_ASYNC_ENABLE must be set before there is an active statement.

[3] SQL_ATTR_TXN_ISOLATION can be set only if there are no open transactions on the connection. Some connection attributes support substitution of a similar value if the data source does not support the value specified in *ValuePtr. In such cases, the driver returns SQL_SUCCESS_WITH_INFO and SQLSTATE 01S02 (Option value changed). For example, if Attribute is SQL_ATTR_PACKET_SIZE and *ValuePtr exceeds the maximum packet size, the driver substitutes the maximum size. To determine the substituted value, an application calls SQLGetConnectAttr.

The format of information set in the *ValuePtr buffer depends on the specified Attribute. SQLSetConnectAttr will accept attribute information in one of two different formats: a null-terminated character string or a 32-bit integer value. The format of each is noted in the attribute's description. Character strings pointed to by the ValuePtr argument of SQLSetConnectAttr have a length of StringLength bytes.

Related Functions

Function	Description
SQLAllocHandle	Allocating a handle
SQLGetConnectAttr	Returning the setting of a connection attribute

Code Example

```
// SQLConnect_ref.cpp
// compile with: odbc32.lib
#include <windows.h>
#include <sqlext.h>

int main() {
   SQLHENV henv;
   SQLHDBC hdbc;
   SQLHSTMT hstmt;
   SQLRETURN retcode;
   SQLPOINTER rgbValue;
   int i = 5;
   rgbValue = &i;

   SQLCHAR * OutConnStr = (SQLCHAR * )malloc(255);
   SQLSMALLINT * OutConnStrLen = (SQLSMALLINT *)malloc(255);

   // Allocate environment handle
   retcode = SQLAllocHandle(SQL_HANDLE_ENV, SQL_NULL_HANDLE, &henv);

   // Set the ODBC version environment attribute
   if (retcode == SQL_SUCCESS || retcode == SQL_SUCCESS_WITH_INFO) {
      retcode = SQLSetEnvAttr(henv, SQL_ATTR_ODBC_VERSION, (void*)
SQL_OV_ODBC3, 0);

      // Allocate connection handle
      if (retcode == SQL_SUCCESS || retcode == SQL_SUCCESS_WITH_INFO) {
         retcode = SQLAllocHandle(SQL_HANDLE_DBC, henv, &hdbc);

         // Set login timeout to 5 seconds
         if (retcode == SQL_SUCCESS || retcode == SQL_SUCCESS_WITH_INFO) {
            SQLSetConnectAttr(hdbc, SQL_LOGIN_TIMEOUT, (SQLPOINTER)
(rgbValue), 0);

            // Connect to data source
            retcode = SQLConnect(hdbc, (SQLCHAR*) "NorthWind", SQL_NTS,
(SQLCHAR*) NULL, 0, NULL, 0);

            // Allocate statement handle
            if (retcode == SQL_SUCCESS || retcode == SQL_SUCCESS_WITH_INFO)
{
               retcode = SQLAllocHandle(SQL_HANDLE_STMT, hdbc, &hstmt);

               // Process data
               if (retcode == SQL_SUCCESS || retcode ==
SQL_SUCCESS_WITH_INFO) {
                  SQLFreeHandle(SQL_HANDLE_STMT, hstmt);
               }

               SQLDisconnect(hdbc);
            }

            SQLFreeHandle(SQL_HANDLE_DBC, hdbc);
         }
      }
      SQLFreeHandle(SQL_HANDLE_ENV, henv);
```

Copyright © 1982-2025 R:BASE Technologies, Inc.

```
    }
}
```

4.55 SQLSetConnectOption

SQLSetConnectOption sets database connection options. Connection options can also be set in the OTERRO11.CFG file.

Syntax

```
RETCODE = SQLSetConnectOption (hdbc, usOption, ulParam)
```

Arguments

Type	Argument	Use	Description
Long	hdbc	Input	The database connection handle.
Integer	usOption	Input	The information option to set.
Long	ulParam	Input	A 32-bit integer value for usOption, or a null-terminated character string.

Return Values

`SQL_SUCCESS, SQL_ERROR,` or `SQL_INVALID_HANDLE`

Information Types

All connection options must be set before calling SQLDriverConnect. Connection options for the Oterro Engine are as follows:

usOption Constant	**Value**	**Description**
SQL_ACCESS_MODE	101	0 = SQL_MODE_READ_WRITE 1 = SQL_MODE_READ_ONLY
SQL_AUTOCOMMIT	102	1 = AUTOCOMMIT 0 = manual commit
SQL_MICRORIM_AUTOCONVERT_MODE	1010	1 = convert on 0 = convert off
SQL_MICRORIM_AUTORECOVER_MODE	1009	1 = recover on 0 = recover off
SQL_MICRORIM_AUTOROWVER_MODE	1013	1 = on 0 = off
SQL_MICRORIM_AUTOSYNC_MODE	1011	1 = sync on 0 = sync off
SQL_MICRORIM_AUTOUPGRADE_MODE	1012	1 = upgrade on 0 = upgrade off
SQL_MICRORIM_COMPATIBILITY_MODE	1001	1 = on 0 = off
SQL_MICRORIM_FASTLOCKS_MODE	1008	1 = fastlocks on 0 = fastlocks off
SQL_MICRORIM_MAX_TRANSACTIONS	1003	A number from 1 to 255 designating the maximum number of users Oterro should allow in transaction mode.
SQL_MICRORIM_MULTIUSER_MODE	1004	1 = multi-user on 0 = multi-user off
SQL_MICRORIM_STATICDB_MODE	1007	1 = static on 0 = static off
SQL_MICRORIM_TRANSACTION_MODE	1006	1 = transactions on 0 = transactions off

Comments

The ODBC specifications require that transaction processing be set to ON, and AUTOCOMMIT be set to ON. You can override these defaults by setting AUTOCOMMIT OFF or TRANSACT OFF in the OTERRO11.CFG file or with the SQLSetConnectOption; the SQLSetConnectOption will override settings in the OTERRO11.CFG file. (An OTERRO11.CFG file is installed with the Oterro Engine.) To simplify development and remove the need to recover databases with inconsistent transaction information caused by a program or hardware failure, set TRANSACT OFF during application development.

SQL_ACCESS_MODE, SQL_MICRORIM_MULTIUSER_MODE, SQL_MICRORIM_TRANSACTION_MODE, and SQL_MICRORIM_STATICDB_MODE require users to connect in the same mode. The first person who connects to a database determines the required settings for all other users.

Related Functions

Function	Description
SQLAllocConnect	Allocates a connection handle.
SQLConnect	Opens a connection to a database.
SQLDriverConnect	Prompts for information to open a connection to a database.
SQLGetConnectOption	Queries the status of a connection option.

Errors

SQLSTATE	Description
S1000	An error has occurred that has no defined SQLSTATE—see the error message text.
S1001	A memory allocation failure.
S1092	An option type was out of range.
S1C00	The driver or data source does not support the specified type.

Visual Basic Example

SQLAPI.BAS:
Declare Function SQLSetConnectOption Lib "ODBC32.DLL" (ByVal hdbc&, ByVal usOption%, ByVal ulParam&) As Integer

CODE:
```
Global cbcol1 As Long
Global cbcol2 As Long

Private Sub ok1_Click()
     If iType = 0 Then
          cbcol1 = conn2.check1.Value
          retcode = SQLSetConnectOption(hdbc&, SQL_MICRORIM_MULTIUSER_MODE,
cbcol1)
               errorcheck retcode
          cbcol2 = conn2.Check2.Value
          retcode = SQLSetConnectOption(hdbc&,
SQL_MICRORIM_TRANSACTION_MODE, cbcol2)
               errorcheck retcode
     End If
     Unload conn2
End Sub
```

4.56 SQLSetCursorName

SQLSetCursorName associates a cursor name with the current statement handle.

Syntax

RETCODE = SQLSetCursorName (hstmt, szCursorName, cbCursor)

Arguments

Type	Argument	Use	Description
Long	hstmt	Input	The statement handle.
String	szCursorName	Input	The buffer containing the cursor name.
Integer	cbCursor	Input	The length of the cursor name.

Return Values

```
SQL_SUCCESS, SQL_ERROR, or SQL_INVALID_HANDLE
```

Comments

A new cursor name cannot be defined if a SELECT statement has been executed and the driver has already defined a cursor name for the statement.

Related Functions

Function	Description
SQLExecDirect	Executes an SQL statement.
SQLExecute	Executes a prepared SQL statement.
SQLGetCursorName	Gets the name of the cursor associated with a statement handle.
SQLPrepare	Prepares an SQL statement for execution.

Errors

SQLSTATE	Description
24000	An invalid cursor state: A cursor is currently open on the statement handle.
34000	An invalid cursor name: The cursor specified in the statement does not exist.
3C000	The cursor name already exists.
S1000	An error has occurred that has no defined SQLSTATE—see the error message text.
S1001	A memory allocation failure.
S1009	An invalid argument value—a null pointer was passed.
S1090	An invalid string or buffer length.

Visual Basic Example

SQLAPI.BAS:
Declare Function SQLSetCursorName Lib "ODBC32.DLL" (ByVal hstmt&, ByVal szCursorName$, ByVal cbCursor%) As Integer

CODE:

```
Global xarray1(50) As Variant
Global colresults As String * 5000
Global cbcolresults As Long
Global sqlstring As String

Private Sub ms2cursor_Click()
     Dim cblong1 As Long
     Dim cbint1 As Integer
     Dim szcur1 As String
     Dim cbcur1 As Integer
     Dim szcur2 As String * 50
     Dim cbcur2 As Integer
     Dim hstmtselect As Long
     Dim hstmtupd As Long
     'to perform a positioned update, you need to define 2 statement
handles
     retcode = SQLAllocStmt(hdbc&, hstmtselect)
     retcode = SQLAllocStmt(hdbc&, hstmtupd)
     'define a cursor name and set it
     szcur1 = "testCursor9x9" & vbNullChar
     cbcur1 = Len(Chop(szcur1))
     retcode = SQLSetStmtOption(hstmtselect, SQL_ROWSET_SIZE, 1)
          errorcheck retcode
     retcode = SQLSetCursorName(hstmtselect, szcur1, cbcur1)
          errorcheck retcode
     'perform a select to activate the cursor
     sqlstring = "select real1 from numbers" & vbNullChar
     retcode = SQLExecDirect(hstmtselect, sqlstring, SQL_NTS)
          errorcheck retcode
     'fetch the third row
     retcode = SQLExtendedFetch(hstmtselect, SQL_FETCH_NEXT, 1, cblong1,
cbint1)
          errorcheck retcode
     retcode = SQLExtendedFetch(hstmtselect, SQL_FETCH_NEXT, 1, cblong1,
cbint1)
          errorcheck retcode
     retcode = SQLExtendedFetch(hstmtselect, SQL_FETCH_NEXT, 1, cblong1,
cbint1)
          errorcheck retcode
     'get the cursor name to use with the update
     retcode = SQLGetCursorName(hstmtselect, szcur2, 18, cbcur2)
          errorcheck retcode
     'execute the update and close both statements
     sqlstring = "update numbers set real1 = 9.9 where current of "
Chop(szcur2) & vbNullChar
     retcode = SQLExecDirect(hstmtupd, sqlstring, SQL_NTS)
          errorcheck retcode
     retcode = SQLFreeStmt(hstmtselect, SQL_CLOSE)
     retcode = SQLFreeStmt(hstmtupd, SQL_CLOSE)
     'verify the update was performed
     sqlstring = "select real1 from numbers" & vbNullChar
     retcode = SQLExecDirect(hstmtselect, sqlstring, SQL_NTS)
          errorcheck retcode
     i = 1
     Do While SQLFetch(hstmtselect) = SQL_SUCCESS
          retcode = SQLGetData(hstmtselect, 1, SQL_C_CHAR, colresults,
5000, cbcolresults)
               errorcheck retcode
          xarray1(i) = Chop(colresults)
          i = i + 1
     Loop
     view2.List1.Clear
     i = 1
     n = 1
     Do While n <> 0
          view2.List1.AddItem xarray1(i)
          i = i + 1
     n = Len(xarray1(i))
     Loop
     retcode = SQLFreeStmt(hstmtselect, SQL_DROP)
     retcode = SQLFreeStmt(hstmtupd, SQL_DROP)
End Sub
```

4.57 SQLSetEnvAttr

SQLSetEnvAttr sets attributes that govern aspects of environments.

Syntax

RETCODE = SQLSetEnvAttr(EnvironmentHandle,Attribute,ValuePtr,StringLength)

Arguments

Type	Argument	Use	Description
Long	EnvironmentHandle	Input	Environment handle.
Integer	Attribute	Input	Attribute to set, listed in "Comments."
String	ValuePtr	Input	Pointer to the value to be associated with Attribute. Depending on the value of Attribute, ValuePtr will be a 32-bit integer value or point to a null-terminated character string.
Integer	StringLength	Input	If ValuePtr points to a character string or a binary buffer, this argument should be the length of *ValuePtr. For character string data, this argument should contain the number of bytes in the string. If ValuePtr is an integer, StringLength is ignored.

Return Values

SQL_SUCCESS, SQL_SUCCESS_WITH_INFO, SQL_ERROR, or SQL_INVALID_HANDLE

Errors

When SQLSetEnvAttr returns SQL_ERROR or SQL_SUCCESS_WITH_INFO, an associated SQLSTATE value can be obtained by calling SQLGetDiagRec with a HandleType of SQL_HANDLE_ENV and a Handle of EnvironmentHandle. The following table lists the SQLSTATE values typically returned by SQLSetEnvAttr and explains each one in the context of this function; the notation "(DM)" precedes the descriptions of SQLSTATEs returned by the Driver Manager. The return code associated with each SQLSTATE value is SQL_ERROR, unless noted otherwise. If a driver does not support an environment attribute, the error can be returned only during connect time.

SQLSTATE	Error	Description
01000	General warning	Driver-specific informational message. (Function returns SQL_SUCCESS_WITH_INFO.)
01S02	Option value changed	The driver did not support the value specified in ValuePtr and substituted a similar value. (Function returns SQL_SUCCESS_WITH_INFO.)

HY000	General error	An error occurred for which there was no specific SQLSTATE and for which no implementation-specific SQLSTATE was defined. The error message returned by SQLGetDiagRec in the *MessageText buffer describes the error and its cause.
HY001	Memory allocation error	The driver was unable to allocate memory required to support execution or completion of the function.
HY009	Invalid use of null pointer	The Attribute argument identified an environment attribute that required a string value, and the ValuePtr argument was a null pointer.
HY010	Function sequence error	(DM) A connection handle has been allocated on EnvironmentHandle.
HY013	Memory management error	The function call could not be processed because the underlying memory objects could not be accessed, possibly because of low memory conditions.
HY024	Invalid attribute value	Given the specified Attribute value, an invalid value was specified in ValuePtr.
HY090	Invalid string or buffer length	The StringLength argument was less than 0 but was not SQL_NTS.
HY092	Invalid attribute/option identifier	(DM) The value specified for the argument Attribute was not valid for the version of ODBC supported by the driver.
HYC00	Optional feature not implemented	The value specified for the argument Attribute was a valid ODBC environment attribute for the version of ODBC supported by the driver, but was not supported by the driver. (DM) The Attribute argument was SQL_ATTR_OUTPUT_NTS, and ValuePtr was SQL_FALSE.

Comments

An application can call SQLSetEnvAttr only if no connection handle is allocated on the environment. All environment attributes successfully set by the application for the environment persist until SQLFreeHandle is called on the environment. More than one environment handle can be allocated simultaneously in ODBC 3.x.

The format of information set through ValuePtr depends on the specified Attribute. SQLSetEnvAttr will accept attribute information in one of two different formats: a null-terminated character string or a 32-bit integer value. The format of each is noted in the attribute's description.

There are no driver-specific environment attributes.

Connection attributes cannot be set by a call to SQLSetEnvAttr. Trying to do this will return SQLSTATE HY092 (Invalid attribute/option identifier).

Attribute	**ValuePtr contents**
SQL_ATTR_CONNECTION_POOLIN G (ODBC 3.0)	A 32-bit SQLUINTEGER value that enables or disables connection pooling at the environment level. The following values are used: SQL_CP_OFF = Connection pooling is turned off. This is the default. SQL_CP_ONE_PER_DRIVER = A single connection pool is supported for each driver. Every connection in a pool is associated with one driver. SQL_CP_ONE_PER_HENV = A single connection pool is supported for each environment. Every connection in a pool is associated with one environment. Connection pooling is enabled by calling SQLSetEnvAttr to set the SQL_ATTR_CONNECTION_POOLING attribute to SQL_CP_ONE_PER_DRIVER or SQL_CP_ONE_PER_HENV. This call must be made before the application allocates the shared environment for which connection pooling is to be enabled. The environment handle in the call to SQLSetEnvAttr is set to null, which makes SQL_ATTR_CONNECTION_POOLING a process-level attribute.

	After connection pooling is enabled, the application then allocates an implicit shared environment by calling SQLAllocHandle with the InputHandle argument set to SQL_HANDLE_ENV. After connection pooling has been enabled and a shared environment has been selected for an application, SQL_ATTR_CONNECTION_POOLING cannot be reset for that environment, because SQLSetEnvAttr is called with a null environment handle when setting this attribute. If this attribute is set while connection pooling is already enabled on a shared environment, the attribute affects only shared environments that are allocated subsequently.
SQL_ATTR_CP_MATCH (ODBC 3.0)	A 32-bit SQLUINTEGER value that determines how a connection is chosen from a connection pool. When SQLConnect or SQLDriverConnect is called, the Driver Manager determines which connection is reused from the pool. The Driver Manager tries to match the connection options in the call and the connection attributes set by the application to the keywords and connection attributes of the connections in the pool. The value of this attribute determines the level of precision of the matching criteria. The following values are used to set the value of this attribute: SQL_CP_STRICT_MATCH = Only connections that exactly match the connection options in the call and the connection attributes set by the application are reused. This is the default. SQL_CP_RELAXED_MATCH = Connections with matching connection string keywords can be used. Keywords must match, but not all connection attributes must match. For more information about how the Driver Manager performs the match in connecting to a pooled connection, see SQLConnect.
SQL_ATTR_ODBC_VERSION (ODBC 3.0)	A 32-bit integer that determines whether certain functionality exhibits ODBC 2.x behavior or ODBC 3.x behavior. The following values are used to set the value of this attribute: SQL_OV_ODBC3 = The Driver Manager and driver exhibit the following ODBC 3.x behavior: • The driver returns and expects ODBC 3.x codes for date, time, and timestamp. • The driver returns ODBC 3.x SQLSTATE codes when SQLError, SQLGetDiagField, or SQLGetDiagRec is called. • The CatalogName argument in a call to SQLTables accepts a search pattern. SQL_OV_ODBC2 = The Driver Manager and driver exhibit the following ODBC 2.x behavior. This is especially useful for an ODBC 2.x application working with an ODBC 3.x driver. • The driver returns and expects ODBC 2.x codes for date, time, and timestamp. • The driver returns ODBC 2.x SQLSTATE codes when SQLError, SQLGetDiagField, or SQLGetDiagRec is called. • The CatalogName argument in a call to SQLTables does not accept a search pattern. An application must set this environment attribute before it calls any function that has an SQLHENV argument, or the call will return SQLSTATE HY010 (Function sequence error). It is driver-specific whether additional behavior exists for these environmental flags.
SQL_ATTR_OUTPUT_NTS (ODBC 3.0)	A 32-bit integer that determines how the driver returns string data. If SQL_TRUE, the driver returns string data null-terminated. If SQL_FALSE, the driver does not return string data null-terminated.

	This attribute defaults to SQL_TRUE. A call to SQLSetEnvAttr to set it to SQL_TRUE returns SQL_SUCCESS. A call to SQLSetEnvAttr to set it to SQL_FALSE returns SQL_ERROR and SQLSTATE HYC00 (Optional feature not implemented).

Related Functions

Function	Description
SQLAllocHandle	Allocating a handle

4.58 SQLSetStmtAttr

SQLSetStmtAttr sets attributes related to a statement.

Syntax

```
RETCODE  = SQLSetStmtAttr(StatementHandle,Attribute,ValuePtr,StringLength)
```

Arguments

Type	Argument	Use	Description
Long	StatementHandle	Input	Statement handle.
Integer	Attribute	Input	Option to set, listed in "Comments."
Long	ValuePtr	Input	Pointer to the value to be associated with Attribute. Depending on the value of Attribute, ValuePtr will be a 32-bit unsigned integer value or a pointer to a null-terminated character string, a binary buffer, or a driver-defined value. If the Attribute argument is a driver-specific value, ValuePtr may be a signed integer.
Integer	StringLength	Input	If Attribute is an ODBC-defined attribute and ValuePtr points to a character string or a binary buffer, this argument should be the length of *ValuePtr. If Attribute is an ODBC-defined attribute and ValuePtr is an integer, StringLength is ignored. If Attribute is a driver-defined attribute, the application indicates the nature of the attribute to the Driver Manager by setting the StringLength argument. StringLength can have the following values: • If ValuePtr is a pointer to a character string, then StringLength is the length of the string or SQL_NTS. • If ValuePtr is a pointer to a binary buffer, then the application places the result of the SQL_LEN_BINARY_ATTR(length) macro in StringLength. This places a negative value in StringLength. • If ValuePtr is a pointer to a value other than a character string or a binary string, then StringLength should have the value SQL_IS_POINTER. • If ValuePtr contains a fixed-length value, then StringLength is either SQL_IS_INTEGER or SQL_IS_UINTEGER, as appropriate.

Return Values

```
SQL_SUCCESS, SQL_SUCCESS_WITH_INFO, SQL_ERROR, or SQL_INVALID_HANDLE
```

Errors

When SQLSetStmtAttr returns SQL_ERROR or SQL_SUCCESS_WITH_INFO, an associated SQLSTATE value may be obtained by calling SQLGetDiagRec with a HandleType of SQL_HANDLE_STMT and a Handle of StatementHandle. The following table lists the SQLSTATE values commonly returned by SQLSetStmtAttr and explains each one in the context of this function; the notation "(DM)" precedes the descriptions of SQLSTATEs returned by the Driver Manager. The return code associated with each SQLSTATE value is SQL_ERROR, unless noted otherwise.

SQLSTATE	Error	Description
01000	General warning	Driver-specific informational message. (Function returns SQL_SUCCESS_WITH_INFO.)
01S02	Option value changed	The driver did not support the value specified in ValuePtr, or the value specified in ValuePtr was invalid because of implementation working conditions, so the driver substituted a similar value. (SQLGetStmtAttr can be called to determine the temporarily substituted value.) The substitute value is valid for the StatementHandle until the cursor is closed, at which point the statement attribute reverts to its previous value. The statement attributes that can be changed are: SQL_ ATTR_CONCURRENCY SQL_ ATTR_CURSOR_TYPE SQL_ ATTR_KEYSET_SIZE SQL_ ATTR_MAX_LENGTH SQL_ ATTR_MAX_ROWS SQL_ ATTR_QUERY_TIMEOUT SQL_ATTR_ROW_ARRAY_SIZE SQL_ ATTR_SIMULATE_CURSOR (Function returns SQL_SUCCESS_WITH_INFO.)
08S01	Communication link failure	The communication link between the driver and the data source to which the driver was connected failed before the function completed processing.
24000	Invalid cursor state	The Attribute was SQL_ATTR_CONCURRENCY, SQL_ATTR_CURSOR_TYPE, SQL_ATTR_SIMULATE_CURSOR, or SQL_ATTR_USE_BOOKMARKS, and the cursor was open.
HY000	General error	An error occurred for which there was no specific SQLSTATE and for which no implementation-specific SQLSTATE was defined. The error message returned by SQLGetDiagRec in the *MessageText buffer describes the error and its cause.
HY001	Memory allocation error	The driver was unable to allocate memory required to support execution or completion of the function.
HY009	Invalid use of null pointer	The Attribute argument identified a statement attribute that required a string attribute, and the ValuePtr argument was a null pointer.
HY010	Function sequence error	(DM) An asynchronously executing function was called for the StatementHandle and was still executing when this function was called. (DM) SQLExecute, SQLExecDirect, SQLBulkOperations, or SQLSetPos was called for the StatementHandle and returned SQL_NEED_DATA. This function was called before data was sent for all data-at-execution parameters or columns.
HY011	Attribute cannot be set now	The Attribute was SQL_ATTR_CONCURRENCY, SQL_ ATTR_CURSOR_TYPE, SQL_ ATTR_SIMULATE_CURSOR, or SQL_ ATTR_USE_BOOKMARKS, and the statement was prepared.
HY013	Memory management error	The function call could not be processed because the underlying memory objects could not be accessed, possibly because of low memory conditions.
HY017	Invalid use of an automatically allocated descriptor handle	(DM) The Attribute argument was SQL_ATTR_IMP_ROW_DESC or SQL_ATTR_IMP_PARAM_DESC.

		(DM) The Attribute argument was SQL_ATTR_APP_ROW_DESC or SQL_ATTR_APP_PARAM_DESC, and the value in ValuePtr was an implicitly allocated descriptor handle other than the handle originally allocated for the ARD or APD.
HY024	Invalid attribute value	Given the specified Attribute value, an invalid value was specified in ValuePtr. (The Driver Manager returns this SQLSTATE only for connection and statement attributes that accept a discrete set of values, such as SQL_ATTR_ACCESS_MODE or SQL_ ATTR_ASYNC_ENABLE. For all other connection and statement attributes, the driver must verify the value specified in ValuePtr.) The Attribute argument was SQL_ATTR_APP_ROW_DESC or SQL_ATTR_APP_PARAM_DESC, and ValuePtr was an explicitly allocated descriptor handle that is not on the same connection as the StatementHandle argument.
HY090	Invalid string or buffer length	(DM) *ValuePtr is a character string, and the StringLength argument was less than 0 but was not SQL_NTS.
HY092	Invalid attribute/option identifier	(DM) The value specified for the argument Attribute was not valid for the version of ODBC supported by the driver. (DM) The value specified for the argument Attribute was a read-only attribute.
HYC00	Optional feature not implemented	The value specified for the argument Attribute was a valid ODBC statement attribute for the version of ODBC supported by the driver but was not supported by the driver. The Attribute argument was SQL_ATTR_ASYNC_ENABLE, and a call to SQLGetInfo with an InfoType of SQL_ASYNC_MODE returns SQL_AM_CONNECTION. The Attribute argument was SQL_ATTR_ENABLE_AUTO_IPD, and the value of the connection attribute SQL_ATTR_AUTO_IPD was SQL_FALSE.
HYT01	Connection timeout expired	The connection timeout period expired before the data source responded to the request. The connection timeout period is set through SQLSetConnectAttr, SQL_ATTR_CONNECTION_TIMEOUT.
IM001	Driver does not support this function	(DM) The driver associated with the StatementHandle does not support the function.

Comments

Statement attributes for a statement remain in effect until they are changed by another call to SQLSetStmtAttr or until the statement is dropped by calling SQLFreeHandle. Calling SQLFreeStmt with the SQL_CLOSE, SQL_UNBIND, or SQL_RESET_PARAMS option does not reset statement attributes.

Some statement attributes support substitution of a similar value if the data source does not support the value specified in ValuePtr. In such cases, the driver returns SQL_SUCCESS_WITH_INFO and SQLSTATE 01S02 (Option value changed). For example, if Attribute is SQL_ATTR_CONCURRENCY and ValuePtr is SQL_CONCUR_ROWVER, and if the data source does not support this, the driver substitutes SQL_CONCUR_VALUES and returns SQL_SUCCESS_WITH_INFO. To determine the substituted value, an application calls SQLGetStmtAttr.

The format of information set with ValuePtr depends on the specified Attribute. SQLSetStmtAttr accepts attribute information in one of two different formats: a character string or a 32-bit integer value. The format of each is noted in the attribute's description. This format applies to the information returned for each attribute in SQLGetStmtAttr. Character strings pointed to by the ValuePtr argument of SQLSetStmtAttr have a length of StringLength.

Note: The ability to set statement attributes at the connection level by calling SQLSetConnectAttr has been deprecated in ODBC 3.x. ODBC 3.x applications should never set statement attributes at the connection level. ODBC 3.x statement attributes cannot be set at the connection level, with the exception of the SQL_ATTR_METADATA_ID and SQL_ATTR_ASYNC_ENABLE attributes, which are both connection attributes and statement attributes, and can be set at either the connection level or the statement level.

Related Functions

Function	Description
SQLCancel	Canceling statement processing
SQLGetConnectAttr	Returning the setting of a connection attribute
SQLGetStmtAttr	Returning the setting of a statement attribute
SQLSetConnectAttr	Setting a connection attribute

4.59 SQLSetPos

Note: This function requires SQLBindColumns in order to exercise all of its options. Since Visual Basic does not support SQLBindColumns, this function will not work. It is included here with the syntax for using the C or C++ programming language.

SQLSetPos sets the cursor position in a rowset and allows an application to refresh, update, delete, or add data to the rowset.

Syntax

```
RETCODE PASCAL SQLSetPos (hstmt, irow, fOption, fLock)
```

Arguments

Type	Argument	Use	Description
HSTMT	hstmt	Input	The statement handle.
UWORD	irow	Input	The position of the row in the rowset on which the operation is to be performed. If irow is 0, the operation applies to every row.
UWORD	fOption	Input	The operation to perform.
UWORD	fLock	Input	Specifies how to lock the row after performing the operation: SQL_LOCK_NO_CHANGE SQL_LOCK_EXCLUSIVE SQL_LOCK_UNLOCK

Return Values

```
SQL_SUCCESS, SQL_SUCCESS_WITH_INFO, SQL_NEED_DATA, SQL_STILL_EXECUTING,
SQL_ERROR, or SQL_INVALID_HANDLE
```

fOption Argument

fOption Argument	Operation
SQL_POSITION	The driver puts the cursor on the row specified by irow.
SQL_REFRESH	The driver puts the cursor on the row specified by irow and refreshes data in the rowset buffers for that row.
SQL_UPDATE	The driver puts the cursor on the row specified by irow and updates the row of data with the values in the rowset buffers.
SQL_DELETE	The driver positions the cursor on the row specified by irow and deletes the row of data.

4.60 SQLSetScrollOptions

SQLSetScrollOptions sets the scrolling functionality of cursors; it is limited to setting scrolling type and has no effect on the row locking methods.

Syntax

```
RETCODE = SQLSetScrollOptions (hstmt, fConcurrency, crowKeyset, crowRowset)
```

Arguments

Type	Argument	Use	Description
Long	hstmt	Input	The statement handle.
Integer	fConcurrency	Input	Accepts only SQL_CONCUR_LOCK.
Long	crowKeyset	Input	Accepts only SQL_SCROLL_FORWARD_ONLY or SQL_SCROLL_KEYSET_DRIVEN.
Integer	crowRowset	Input	The number of rows in a rowset.

Return Values

```
SQL_SUCCESS, SQL_ERROR, or SQL_INVALID_HANDLE
```

Comments

This function has been replaced by the SQLSetStmtOption function; it is included only for compatibility with previous versions of ODBC.
This function calls SQLSetStmtOption to set the cursor scroll options.

4.61 SQLSetStmtOption

SQLSetStmtOption sets an option for a particular statement and sets cursor scrolling functionality.

Syntax

```
RETCODE = SQLSetStmtOption (hstmt, fOption, vParam)
```

Arguments

Type	Argument	Use	Description
Long	hstmt	Input	The statement handle.
Integer	fOption	Input	The information option to set.
Long	vParam	Input	A 32-bit integer value for fOption.

Return Values

```
SQL_SUCCESS, SQL_ERROR, or SQL_INVALID_HANDLE
```

Information Types

Statement options that are supported by the Oterro Engine:

fOption Constant	**Value**	**Description**
SQL_ASYNC_ENABLE	4	Specifies whether a statement is executed asynchronously.
SQL_BIND_TYPE	5	Specifies the binding orientation.
SQL_CONCURRENCY	7	May be one of the following: SQL_CONCUR_LOCK SQL_CONCUR_READ_ONLY SQL_CONCUR_ROWVER SQL_CONCUR_VALUES
SQL_CURSOR_TYPE	6	May be one of the following: SQL_CURSOR_FORWARD_ONLY SQL_CURSOR_KEYSET_DRIVEN SQL_CURSOR_STATIC SQL_CURSOR_DYNAMIC
SQL_KEYSET_SIZE	8	Number of rows in a KEYSET.
SQL_MAX_LENGTH	3	The maximum length of data from a text column.
SQL_MAX_ROWS	1	The maximum number of rows to be returned.
SQL_NOSCAN	2	Specifies whether SQL strings are scanned for escape clauses.
SQL_QUERY_TIMEOUT	0	Number of seconds before timing out on an SQL statement.
SQL_RETRIEVE_DATA	15	Specifies whether an SQLExtendedFetch retrieves data after positioning.
SQL_ROWSET_SIZE	9	Number of rows returned.
SQL_SIMULATE_CURSOR	10	Specifies whether positioned update and delete affect only one row.
SQL_USE_BOOKMARK	23	Specifies whether bookmarks will be used. SQL_UB_OFF or SQL_UB_ON.

Related Functions

Function	Description
SQLCancel	Ends processing on a statement.
SQLExtendedFetch	Fetches one row of a result set; allows scrolling.
SQLGetStmtOption	Queries the status of a statement option.
SQLSetScrollOptions	Sets the scrolling functionality of cursors.

Errors

SQLSTATE	Description
01000	A driver-specific informational message. (The function returns SQL_SUCCESS_WITH_INFO.)
08S01	The data source connection failed before the function completed processing.
24000	An invalid cursor state: A cursor is currently open on the statement handle.
IM001	The driver associated with the hstmt does not support the function.
S1000	An error has occurred that has no defined SQLSTATE—see the error message text.
S1001	A memory allocation failure.
S1009	An invalid argument value—a null pointer was passed.
S1092	An option type was out of range.
S1C00	The driver or data source does not support the specified type.

Visual Basic Example

SQLAPI.BAS:
Declare Function SQLSetStmtOption Lib "ODBC32.DLL" (ByVal hstmt&, ByVal fOption%, ByVal vParam&) As Integer

```
Global sqlstring As String

Private Sub ms2extfetch_Click()
     Dim cblong1 As Long
     Dim cbint1 As Integer
     Dim i As Integer
     Dim n As Integer
     sqlstring = "select int1,real1,doub1 from numbers" & vbNullChar
     retcode = SQLPrepare(hStmt&, sqlstring, SQL_NTS)
          errorcheck retcode
     retcode = SQLExecute(hStmt&)
          errorcheck retcode
     retcode = SQLSetStmtOption(hStmt&, SQL_CURSOR_TYPE,
SQL_CURSOR_DYNAMIC)
     retcode = SQLGetStmtOption(hStmt&, SQL_CURSOR_TYPE, cblong1)
     xarray1(1) = "cursor type = " & cblong1
     retcode = SQLExtendedFetch(hStmt&, SQL_FETCH_FIRST, 1, cblong1,
cbint1)
     retcode = SQLGetData(hStmt&, 1, SQL_C_CHAR, colresults, 5000,
cbcolresults)
     xarray1(2) = Chop(colresults)
     retcode = SQLExtendedFetch(hStmt&, SQL_FETCH_LAST, 1, cblong1, cbint1)
     retcode = SQLGetData(hStmt&, 1, SQL_C_CHAR, colresults, 5000,
cbcolresults)
     xarray1(3) = Chop(colresults)
     retcode = SQLExtendedFetch(hStmt&, SQL_FETCH_PRIOR, 1, cblong1,
cbint1)
     retcode = SQLGetData(hStmt&, 1, SQL_C_CHAR, colresults, 5000,
cbcolresults)
     xarray1(4) = Chop(colresults)
     view2.List1.Clear
     i = 1
     n = 1
     Do While n <> 0
          view2.List1.AddItem xarray1(i)
          i = i + 1
          n = Len(xarray1(i))
     Loop
     retcode = SQLFreeStmt(hStmt&, SQL_CLOSE)
End Sub
```

4.62 SQLSpecialColumns

SQLSpecialColumns returns information about the set of columns that uniquely identifies a row associated with the set of columns (such as a special rowid column, or a column with a primary key or unique constraint), and a set of columns that identifies when a row is modified.

Syntax

RETCODE=SQLSpecialColumns(hstmt, fColType, szTableQualifier, cbTableQualifier, szTableOwner, cbTableOwner, szTableName, cbTableName, fScope, fNullable)

Arguments

Type	Argument	Use	Description
Long	hstmt	Input	The statement handle.
Integer	fColType	Input	SQL_BEST_ROWID or SQL_ROWVER.
String	szTableQualifier	Input	The buffer containing the table qualifier.
Integer	cbTableQualifier	Input	The length of the table qualifier.
String	szTableOwner	Input	The buffer containing the table-owner name.
Integer	cbTableOwner	Input	The length of the table-owner name.
String	szTableName	Input	The buffer containing the table name.
Integer	cbTableName	Input	The length of the table name.
Integer	fScope	Input	Accepts one of the following: SQL_SCOPE_CURROW SQL_SCOPE_TRANSACTION SQL_SCOPE_SESSION
Integer	fNullable	Input	Returns either SQL_NO_NULLS (columns that are defined as primary keys or not NULL constraints) or SQL_NULLABLE.

Return Values

```
SQL_SUCCESS, SQL_ERROR, or SQL_INVALID_HANDLE
```

Result Set

Column Name	**Data Type**	**Comments**
SCOPE	INTEGER	SQL_SCOPE_SESSION (2) or NULL.
COLUMN_NAME	TEXT 18	The column name.
DATA_TYPE	INTEGER	Either the ODBC or Oterro data type. See SQLGetTypeInfo for a list of valid data types.
TYPE_NAME	TEXT 8	The textual name of the data type, for example, INTEGER.
PRECISION	INTEGER	The precision of the data type for the COLUMN_NAME.
LENGTH	INTEGER	The length of the data type for COLUMN_NAME.
SCALE	INTEGER	The scale of the column.
PSEUDO_COLUMN	INTEGER	Specifies if the column is a pseudo-column.

Comments

When the value of fColType is SQL_BEST_ROWID, the function returns the column name(s) that uniquely defines a row. The function returns a primary key or unique column when one is defined, or an empty set when a primary key or unique column is not defined.

When the value of fColType is SQL_ROWVER, the function returns the column name(s) to use to identify when a row is modified. This use is primarily for concurrency control.

Related Functions

Function	Description
SQLCancel	Ends processing on a statement.
SQLColumns	Returns the columns in a table(s).
SQLExtendedFetch	Fetches one row of a result set; allows scrolling.
SQLFetch	Fetches one row of a result set.
SQLPrimaryKeys	Returns the columns defined as primary keys.

Errors

SQLSTATE	Description
01000	A driver-specific informational message. (The function returns SQL_SUCCESS_WITH_INFO.)
08S01	The data source connection failed before the function completed processing.
24000	An invalid cursor state: A cursor is currently open on the statement handle.
IM001	The driver associated with the hstmt does not support the function.
S1000	An error has occurred that has no defined SQLSTATE—see the error message text.
S1001	A memory allocation failure.
S1090	An invalid string or buffer length.
S1097	An invalid fColType value was specified.
S1098	An invalid fScope value was specified.
S1099	An invalid fNullable value was specified.
S1C00	The driver or data source does not support the specified type.

Visual Basic Example

SQLAPI.BAS:
Declare Function SQLSpecialColumns Lib "ODBC32.DLL" (ByVal hstmt&, ByVal fColType%, ByVal szTableQualifier$, ByVal cbTableQualifier%, ByVal szTableOwner$, ByVal cbTableOwner%, ByVal szTableName$, ByVal cbTableName%, ByVal fScope%, ByVal fNullable%) As Integer

CODE:
```
Global colresults As String * 5000
Global cbcolresutles As Long
Global colnum As Integer

Private Sub mdb1spcol_Click()
     retcode = SQLSpecialColumns(hStmt&, SQL_BEST_ROWID, "", 0, "", 0,
"numbers", 7, SQL_SCOPE_SESSION, SQL_NULLABLE)
          errorcheck retcode
     loadtest
End Sub

Sub loadtest()
     Dim i As Integer
     Dim cblong1 As Long
     Dim cbint1 As Integer
     i = 0
     retcode = SQLNumResultCols(hStmt&, colnum)
          errorcheck retcode
     i = 1
     Do While SQLFetch(hStmt&) = SQL_SUCCESS
          Do While i <= colnum
               retcode = SQLGetData(hStmt&, i, SQL_C_CHAR, colresults,
5000, cbcolresults)
               view1.text1.Text = view1.text1.Text & vbCrLf & "Col" & i &
": " Chop(colresults)
               i = i + 1
          Loop
          i = 1
     Loop
     retcode = SQLFreeStmt(hStmt&, SQL_CLOSE)
```

```
End Sub
```

4.63 SQLStatistics

SQLStatistics returns statistics for a table and its indexes.

Syntax

RETCODE=SQLStatistics(hstmt, szTableQualifier, cbTableQualifier, szTableOwner, cbTableOwner, szTableName, cbTableName, fUnique, fAccuracy)

Arguments

Type	**Argument**	**Use**	**Description**
Long	hstmt	Input	The statement handle.
String	szTableQualifier	Input	The buffer containing the table qualifier.
Integer	cbTableQualifier	Input	The length of the table qualifier.
String	szTableOwner	Input	The buffer containing the table-owner name.
Integer	cbTableOwner	Input	The length of the table-owner name.
String	szTableName	Input	The buffer containing the table name.
Integer	cbTableName	Input	The length of the table name.
Integer	fUnique	Input	The type of index; accepts either SQL_INDEX_UNIQUE or SQL_INDEX_ALL
Integer	fAccuracy	Input	The accuracy of the statistics returned. Accepts either: SQL_ENSURE—statistics are always retrieved SQL_QUICK—statistics are retrieved if they are readily available

Return Values

SQL_SUCCESS, SQL_ERROR, or SQL_INVALID_HANDLE

Result Set

Column Name	Data Type	Comments
TABLE_QUALIFIER	TEXT 18	The table qualifier. Always NULL for the Oterro Engine.
TABLE_OWNER	TEXT 18	The table owner name. Always NULL for the Oterro Engine.
TABLE_NAME	TEXT 18	The table name.
NON_UNIQUE	INTEGER	Indicates if the index allows duplicate values. 0 = UNIQUE index 1 = duplicate values allowed
INDEX_QUALIFIER	TEXT 18	The index qualifier. Always NULL for the Oterro Engine.
INDEX_NAME	TEXT 18	The index name.
TYPE	INTEGER	The information type returned. The Oterro Engine always returns SQL_INDEX_OTHER (3) indicating "other index type".
SEQ_IN_INDEX	TEXT 18	The column sequence number in the index. Always 1 unless a multi-column index is defined. Multi-column indexes contain the sequence number identifying the order of the column in the multi-column index, for example, 1, 2, 3 etc. Multi-column indexes are sorted in the order the columns are specified.
COLUMN_NAME	TEXT 18	The column name the index is built on.

COLLATION	TEXT 1	The sort order for the index. A - Ascending D - Descending
CARDINALITY	INTEGER	The cardinality of the table or index. Always NULL for the Oterro Engine.
PAGES	INTEGER	The number of pages used to store the table or index. Always NULL for the Oterro Engine.
FILTER_CONDITION	TEXT 18	The check condition for the index. Always NULL for the Oterro Engine.

Comments

The value for fUnique can be either SQL_INDEX_UNIQUE or SQL_INDEX_ALL. The option SQL_INDEX_UNIQUE only returns indexes with unique constraints. The option SQL_INDEX_ALL returns all indexes in random order.

Related Functions

Function	Description
SQLCancel	Ends processing on a statement.
SQLExtendedFetch	Fetches one row of a result set; allows scrolling.
SQLFetch	Fetches one row of a result set.
SQLForeignKeys	Returns the columns defined as foreign keys.
SQLPrimaryKeys	Returns the columns defined as primary keys.
SQLTablePrivileges	Returns the privileges assigned to the table.

Errors

SQLSTATE	Description
01000	A driver-specific informational message. (The function returns SQL_SUCCESS_WITH_INFO.)
08S01	The data source connection failed before the function completed processing.
24000	An invalid cursor state: A cursor is currently open on the statement handle.
IM001	The driver associated with the hstmt does not support the function.
S1000	An error has occurred that has no defined SQLSTATE—see the error message text.
S1001	A memory allocation failure.
S1090	An invalid string or buffer length.
S1100	An invalid fUnique value was specified.
S1101	An invalid fAccuracy value was specified.
S1C00	The driver or data source does not support the specified type.

Visual Basic Example

SQLAPI.BAS:
Declare Function SQLStatistics Lib "ODBC32.DLL" (ByVal hstmt&, ByVal szTableQualifier$, ByVal cbTableQualifier%, ByVal szTableOwner$, ByVal cbTableOwner%, ByVal szTableName$, ByVal cbTableName%, ByVal fUnique%, ByVal fAccuracy%) As Integer

CODE:
```
Global colnum As Integer
Global szTableName As String * 20
Global cbTableName As Integer
Global szFirst As String * 1500
Global cbFirst As Long

Sub statindex()
     Dim i As Integer
     retcode = SQLStatistics(hstmt&, "", 0, "", 0, szTableName,
cbTableName, SQL_INDEX_ALL, SQL_ENSURE)
          errorcheck retcode
     load2grid
End Sub


Sub load2grid()
     Dim i As Integer
     Dim n As Integer
     n = 1
     i = 1
     retcode = SQLNumResultCols(hstmt&, colnum)
          errorcheck retcode
     Do While SQLFetch(hstmt&) = SQL_SUCCESS
          dbstr1.Grid2.Row = n
          Do While i <= colnum
               retcode = SQLGetData(hstmt&, i, SQL_C_CHAR, szFirst, 255,
cbFirst)
               If dbstr1.Grid2.ColWidth(i) < Abs(cbFirst) * 120 Then
                    dbstr1.Grid2.ColWidth(i) = Abs(cbFirst) * 120
               End If
               dbstr1.Grid2.Col = i
               dbstr1.Grid2.Text = Chop(szFirst)
               i = i + 1
          Loop
          n = n + 1
          i = 1
     Loop
     retcode = SQLFreeStmt(hstmt&, SQL_CLOSE)
End Sub
```

4.64 SQLTablePrivileges

SQLTablePrivileges returns a list of tables and the privileges associated with each table.

Syntax

RETCODE = SQLTablePrivileges (hStmt, szTableQualifier, cbTableQualifier, szTableOwner, cbTableOwner, szTableName, cbTableName)

Arguments

Type	Argument	Use	Description
Long	hStmt	Input	The statement handle.
String	szTableQualifier	Input	The table qualifier.
Integer	cbTableQualifier	Input	The length of the table qualifier.

String	szTableOwner	Input	The table owner name.
Integer	cbTableOwner	Input	The length of the table owner name.
String	szTableName	Input	A table name.
Integer	cbTableName	Input	The length of the table name.

Return Values

`SQL_SUCCESS, SQL_SUCCESS_WITH_INFO, SQL_STILL_EXECUTING, SQL_ERROR, or SQL_INVALID_HANDLE`

Result Set

Column Name	Data Type	Comments
TABLE_QUALIFIER	TEXT 18	The table qualifier. Always NULL in the Oterro Engine.
TABLE_OWNER	TEXT 18	The table owner name. Always NULL in the Oterro Engine.
TABLE_NAME	TEXT 18	The table name.
GRANTOR	TEXT 18	The user who granted the privilege. Always NULL in the Oterro Engine.
GRANTEE	TEXT 18	The user the privilege was granted to.
PRIVILEGE	TEXT 128	The table privilege; one of the following: ALTER, SELECT, INSERT, UPDATE, DELETE, REFERENCES.
IS_GRANTABLE	TEXT 3	Indicates if the grantee is permitted to grant the privilege to other users. Always NULL for the Oterro Engine.

Related Functions

Function	Description
SQLColumnPrivileges	Returns the privileges assigned to the columns of a table.
SQLColumns	Returns the columns in a table.
SQLExtendedFetch	Fetches one row of a result set; allows scrolling.
SQLFetch	Fetches one row of a result set.
SQLStatistics	Returns statistics for tables and indexes.
SQLTables	Returns the tables in a database.

Errors

SQLSTATE	Description
01000	A driver-specific informational message. (Function returns SQL_SUCCESS_WTIH_INFO.)
08S01	The data source connection failed before the function completed processing.
24000	An invalid cursor state.
IM001	The driver does not support this function.
S1000	An error has occurred that has no defined SQLSTATE—see the error message text.
S1001	A memory allocation failure.
S1008	The operation was canceled.
S1010	A function sequence error occurred.
S1090	An invalid string or buffer length.
S1C00	The driver or data source does not support the specified type.
S1T00	The timeout period expired before the data source returned the result.

Visual Basic Example

SQLAPI.BAS:
Declare Function SQLTablePrivileges Lib "ODBC32.DLL" (ByVal hstmt&, ByVal szTableQualifier$, ByVal cbTableQualifier%, ByVal szTableOwner$, ByVal cbTableOwner%, ByVal szTableName$, ByVal cbTableName%) As Integer

CODE:
```
Global colresults As String * 5000
Global cbcolresutles As Long
Global colnum As Integer

Private Sub mdb1tpriv_Click()
     'the following will return privileges on all tables in the database
     retcode = SQLTablePrivileges(hStmt&, "", 0, "", 0, "", 0)
          errorcheck retcode
     loadtest
End Sub

Sub loadtest()
     Dim i As Integer
     Dim cblong1 As Long
     Dim cbint1 As Integer
     i = 0
     retcode = SQLNumResultCols(hStmt&, colnum)
          errorcheck retcode
     i = 1
     Do While SQLFetch(hStmt&) = SQL_SUCCESS
          Do While i <= colnum
               retcode = SQLGetData(hStmt&, i, SQL_C_CHAR, colresults,
5000, cbcolresults)
               view1.text1.Text = view1.text1.Text & vbCrLf & "Col" & i &
": " Chop(colresults)
               i = i + 1
          Loop
          i = 1
     Loop
     retcode = SQLFreeStmt(hStmt&, SQL_CLOSE)
End Sub
```

4.65 SQLTables

SQLTables retrieves the list of available tables in the database.

Syntax

RETCODE = SQLTables(hstmt, szTableQualifier, cbTableQualifier, szTableOwner, cbTableOwner, szTableName, cbTableName, szTableType, cbTableType)

Arguments

Type	Argument	Use	Description
Long	hstmt	Input	The statement handle.
String	szTableQualifier	Input	The buffer containing the table qualifier.
Integer	cbTableQualifier	Input	The length of the table qualifier.
String	szTableOwner	Input	The buffer containing the table-owner name.
Integer	cbTableOwner	Input	The length of the table-owner name.
String	szTableName	Input	The buffer containing the table name.
Integer	cbTableName	Input	The length of the table name.
String	szTableType	Input	The table types to match.
Integer	cbTableType	Input	The length of the table types.

Return Values

```
SQL_SUCCESS, SQL_ERROR, or SQL_INVALID_HANDLE
```

Result Set

Column Name	Data Type	Comments
TABLE_QUALIFIER	TEXT 18	The table qualifier. Always NULL for the Oterro Engine.
TABLE_OWNER	TEXT 18	The table owner name. Always NULL for the Oterro Engine.
TABLE_NAME	TEXT 18	The table name.
TABLE_TYPE	TEXT 18	The table type. The Oterro Engine returns one of the following: TABLE VIEW SYSTEM TABLE
REMARKS	NOTE	A description of the table, if one has been defined.

Comments

Only the statement handle and table-name parameter are allowed by the Oterro Engine.

Information about users' privileges is not necessarily checked, so that any table names returned by SQLTables are not guaranteed to be accessible by the user identifier specified during SQLDriverConnect.

The results are processed like any other query, such as a query using SQLFetch. SQLDescribeCol should be used to determine the data type and length of the column containing the table name.

The szTableName pattern-match string can contain the SQL wild card characters '%' (for matching many characters) and '_' (for matching a single character).

Related Functions

Function	**Description**
SQLColumnPrivileges	Returns the privileges assigned to the columns of a table.
SQLColumns	Returns the columns in a table(s).
SQLDescribeCol	Describes a column in a result set.
SQLExtendedFetch	Fetches one row of a result set; allows scrolling.
SQLFetch	Fetches one row of a result set.
SQLTablePrivileges	Returns the privileges assigned to the table.

Errors

SQLSTATE	Description

24000	An invalid cursor state: A cursor is currently open on the statement handle.
S1000	An error has occurred that has no defined SQLSTATE—see the error message text.
S1001	A memory allocation failure.
S1090	An invalid string or buffer length.

Visual Basic Example

SQLAPI.BAS:
Declare Function SQLTables Lib "ODBC32.DLL" (ByVal hstmt&, ByVal szTableQualifier$, ByVal cbTableQualifier%, ByVal szTableOwner$, ByVal cbTableOwner%, ByVal szTableName$, ByVal cbTableName%, ByVal szTableType$, ByVal cbTableType%) As Integer

CODE:

```
Global colnum As Integer
Global szTableName As String * 20
Global cbTableName As Integer
Global szFirst As String * 1500
Global cbFirst As Long
Global szLast As String * 1500
Global cbLast As Long

Private Sub Form_Load()
     dbstr1.MousePointer = 11
     'returns all tables in the database
     retcode = SQLTables(hstmt&, "", 0, "", 0, "", 0, "", 0)
          errorcheck retcode
     Do While SQLFetch(hstmt&) = SQL_SUCCESS
          retcode = SQLGetData(hstmt&, 3, SQL_C_CHAR, szFirst, 255,
cbFirst)
               errorcheck retcode
          retcode = SQLGetData(hstmt&, 4, SQL_C_CHAR, szLast, 255, cbLast)
               errorcheck retcode
          If Left(szFirst, 4) = "SYS_" Then
          Else
               If Chop(szLast) = "TABLE" Then
                    dbstr1.list1.AddItem "T " & szFirst
               Else
                    dbstr1.list1.AddItem "V " & szFirst
               End If
          End If
     Loop
     retcode = SQLFreeStmt(hstmt&, SQL_CLOSE)
     dbstr1.MousePointer = 1
End Sub
```

4.66 SQLTransact

SQLTransact commits or rolls back all uncommitted transactions for all statement handles for the connection specified by hdbc.

Syntax

```
RETCODE = SQLTransact (henv, hdbc, fType)
```

Arguments

Type	Argument	Use	Description
Long	henv	Input	The environment handle.
Long	hdbc	Input	The database connection handle.
Integer	fType	Input	Accepts either SQL_COMMIT (0) or SQL_ROLLBACK (1).

Return Values

SQL_SUCCESS, SQL_ERROR, or SQL_INVALID_HANDLE

Comments

You can think of a single connection as processing a sequence of statements. In other words, separate statement handles do not represent different statement processing threads. Only one thread is in a statement execution. Therefore, executing a commit or rollback affects all statement handles on the current connection. All cursors are closed upon processing a transaction command.

Passing an environment handle to the SQLTransact function causes all transactions pending in the specified environment to be either committed or rolled back, depending on the value of fType.

Errors

SQLSTATE	Description
08003	No database has been connected.
S1000	An error has occurred that has no defined SQLSTATE—see the error message text.
S1001	A memory allocation failure.
S1012	An invalid transaction option: fType was not SQL_COMMIT or SQL_ROLLBACK.

Visual Basic Example

SQLAPI.BAS:
Declare Function SQLTransact Lib "ODBC32.DLL" (ByVal henv&, ByVal hdbc&, ByVal fType%) As Integer

CODE:
```
Private Sub mt1transact_Click()
     Dim cblong1 As Long
     Dim sqlstring as String
     sqlstring = "update numbers set int1 = 45 where int1 = 99" &
vbNullChar
     retcode = SQLExecDirect(hStmt&, sqlstring, SQL_NTS)
          errorcheck retcode
     retcode = SQLRowCount(hStmt&, cblong1)
          errorcheck retcode
     If cblong1 = 1 Then
          retcode = SQLTransact(hEnv&, hdbc&, SQL_COMMIT)
```

```
                    errorcheck retcode
          Else
               retcode = SQLTransact(hEnv&, hdbc&, SQL_ROLLBACK)
                    errorcheck retcode
          End If
          retcode = SQLCancel(hStmt&)
     End Sub
```

Part V

5 ODBC Reference Topics

5.1 Connection Attributes

Connection attributes are characteristics of the connection. For example, because transactions occur at the connection level, the transaction isolation level is a connection attribute. Similarly, the login timeout, or number of seconds to wait while trying to connect before timing out, is a connection attribute.

Connection attributes are set with SQLSetConnectAttr and their current settings retrieved with SQLGetConnectAttr. If SQLSetConnectAttr is called before the driver is loaded, the Driver Manager stores the attributes in its connection structure and sets them in the driver as part of the connection process. There is no requirement that an application set any connection attributes; all connection attributes have defaults, some of which are driver-specific.

A connection attribute can be set before or after connection, or either, depending on the attribute and the driver. The login timeout (SQL_ATTR_LOGIN_TIMEOUT) applies to the connection process and is effective only if set before connecting. The attributes that specify whether to use the ODBC cursor library (SQL_ATTR_ODBC_CURSORS) and the network packet size (SQL_ATTR_PACKET_SIZE) must be set before connecting, because the ODBC cursor library resides between the Driver Manager and the driver and therefore must be loaded before the driver.

The attributes to specify whether a data source is read-only or read-write (SQL_ATTR_ACCESS_MODE) and the current catalog (SQL_ATTR_CURRENT_CATALOG) can be set before or after connecting, depending on the driver. However, interoperable applications set them before connecting because some drivers do not support changing these after connecting.

Some connection attributes have a default before the connection is made, while others do not. Those that do are SQL_ATTR_ACCESS_MODE, SQL_ATTR_AUTOCOMMIT, SQL_ATTR_LOGIN_TIMEOUT, SQL_ATTR_ODBC_CURSORS, SQL_ATTR_TRACE, and SQL_ATTR_TRACEFILE.

The translation connection attributes (SQL_ATTR_TRANSLATE_DLL and SQL_ATTR_TRANSLATE_OPTION) must be set after connecting.

All other connection attributes can be set at any time.

5.2 Descriptors

A descriptor handle refers to a data structure that holds information about either columns or dynamic parameters.

ODBC functions that operate on column and parameter data implicitly set and retrieve descriptor fields. For instance, when SQLBindCol is called to bind column data, it sets descriptor fields that completely describe the binding. When SQLColAttribute is called to describe column data, it returns data stored in descriptor fields.

An application calling ODBC functions need not concern itself with descriptors. No database operation requires that the application gain direct access to descriptors. However, for some applications, gaining direct access to descriptors streamlines many operations. For example, direct access to descriptors provides a way to rebind column data, which can be more efficient than calling SQLBindCol again.

Note: The physical representation of the descriptor is not defined. Applications gain direct access to a descriptor only by manipulating its fields by calling ODBC functions with the descriptor handle.

See also:

Types of Descriptors

5.2.1 Types of Descriptors

A descriptor is used to describe one of the following:

- A set of zero or more parameters. A parameter descriptor can be used to describe:
  - The application parameter buffer, which contains either the input dynamic arguments as set by the application or the output dynamic arguments following the execution of a CALL statement of SQL.
  - The implementation parameter buffer. For input dynamic arguments, this contains the same arguments as the application parameter buffer, after any data conversion the application may specify. For output dynamic arguments, this contains the returned arguments, before any data conversion that the application may specify.

  For input dynamic arguments, the application must operate on an application parameter descriptor before executing any SQL statement that contains dynamic parameter markers. For both input and output dynamic arguments, the application can specify different data types from those in the implementation parameter descriptor to achieve data conversion.

- A single row of database data. A row descriptor can be used to describe:
  - The implementation row buffer, which contains the row from the database. (These buffers conceptually contain data as written to or read from the database. However, the stored form of database data is not specified. A database could perform additional conversion on the data from its form in the implementation buffer.)
  - The application row buffer, which contains the row of data as presented to the application, following any data conversion that the application may specify.

  The application operates on the application row descriptor in any case where column data from the database must appear in application variables. To achieve data conversion of column data, the application can specify different data types from those in the implementation row descriptor.

The descriptor types are summarized in the following table.

Buffer type	**Rows**	**Dynamic parameters**
Application buffer	Application row descriptor (ARD)	Application parameter descriptor (APD)
Implementation buffer	Implementation row descriptor (IRD)	Implementation parameter descriptor (IPD)

For either the parameter or the row buffers, if the application specifies different data types in corresponding records of the implementation and application descriptors, the driver performs data conversion when it uses the descriptors.

A descriptor can perform different roles. Different statements can share any descriptor that the application explicitly allocates. A row descriptor in one statement can serve as a parameter descriptor in another statement.

It is always known whether a given descriptor is an application descriptor or an implementation descriptor, even if the descriptor has not yet been used in a database operation. For the descriptors that the implementation implicitly allocates, the implementation records the predefined row relative to the statement handle. Any descriptor that the application allocates by calling SQLAllocHandle is an application descriptor.

5.3 Handles

Handles are opaque, 32-bit values that identify a particular item; in ODBC, this item can be an environment, connection, statement, or descriptor. When the application calls SQLAllocHandle, the Driver Manager or driver creates a new item of the specified type and returns its handle to the application. The application later uses the handle to identify that item when calling ODBC functions. The Driver Manager and driver use the handle to locate information about the item.

For example, the following code uses two statement handles (hstmtOrder and hstmtLine) to identify the statements on which to create result sets of sales orders and sales order line numbers. It later uses these handles to identify which result set to fetch data from.

```
SQLHSTMT      hstmtOrder, hstmtLine; // Statement handles.
SQLUINTEGER   OrderID;
SQLINTEGER    OrderIDInd = 0;
SQLRETURN     rc;

// Prepare the statement that retrieves line number information.
SQLPrepare(hstmtLine, "SELECT * FROM Lines WHERE OrderID = ?", SQL_NTS);

// Bind OrderID to the parameter in the preceding statement.
SQLBindParameter(hstmtLine, 1, SQL_PARAM_INPUT, SQL_C_ULONG,
SQL_INTEGER, 5, 0,
               &OrderID, 0, &OrderIDInd);

// Bind the result sets for the Order table and the Lines table. Bind
// OrderID to the OrderID column in the Orders table. When each row is
// fetched, OrderID will contain the current order ID, which will then
be
// passed as a parameter to the statement tofetch line number
// information. Code not shown.

// Create a result set of sales orders.
SQLExecDirect(hstmtOrder, "SELECT * FROM Orders", SQL_NTS);

// Fetch and display the sales order data. Code to check if rc equals
// SQL_ERROR or SQL_SUCCESS_WITH_INFO not shown.
while ((rc = SQLFetch(hstmtOrder)) != SQL_NO_DATA) {
   // Display the sales order data. Code not shown.

   // Create a result set of line numbers for the current sales order.
   SQLExecute(hstmtLine);

   // Fetch and display the sales order line number data. Code to check
   // if rc equals SQL_ERROR or SQL_SUCCESS_WITH_INFO not shown.
   while ((rc = SQLFetch(hstmtLine)) != SQL_NO_DATA) {
      // Display the sales order line number data. Code not shown.
   }

   // Close the sales order line number result set.
   SQLCloseCursor(hstmtLine);
}

// Close the sales order result set.
SQLCloseCursor(hstmtOrder);
```

Handles are meaningful only to the ODBC component that created them; that is, only the Driver Manager can interpret Driver Manager handles and only a driver can interpret its own handles.

For example, suppose the driver in the preceding example allocates a structure to store information about a statement and returns the pointer to this structure as the statement handle. When the application calls SQLPrepare, it passes an SQL statement and the handle of the statement used for sales order line numbers. The driver sends the SQL statement to the data source, which prepares it and returns an access plan identifier. The driver uses the handle to find the structure in which to store this identifier.

Later, when the application calls SQLExecute to generate the result set of line numbers for a particular sales order, it passes the same handle. The driver uses the handle to retrieve the access plan identifier from the structure. It sends the identifier to the data source to tell it which plan to execute.

ODBC has two levels of handles: Driver Manager handles and driver handles. The application uses Driver Manager handles when calling ODBC functions because it calls those functions in the Driver Manager.

That there are two levels of handles is an artifact of the ODBC architecture; in most cases, it is not relevant to either the application or driver. Although there is usually no reason to do so, it is possible for the application to determine the driver handles by calling SQLGetInfo.

See also:

Environment Handles
Connection Handles
Statement Handles
Descriptor Handles
State Transitions

5.3.1 Environment Handles

An environment is a global context in which to access data; associated with an environment is any information that is global in nature, such as:

- The environment's state
- The current environment-level diagnostics
- The handles of connections currently allocated on the environment
- The current settings of each environment attribute

Within a piece of code that implements ODBC (the Driver Manager or a driver), an environment handle identifies a structure to contain this information.

Environment handles are not frequently used in ODBC applications. They are always used in calls to SQLDataSources and SQLDrivers and sometimes used in calls to SQLAllocHandle, SQLEndTran, SQLFreeHandle, and SQLGetDiagRec.

Each piece of code that implements ODBC (the Driver Manager or a driver) contains one or more environment handles. For example, the Driver Manager maintains a separate environment handle for each application that is connected to it. Environment handles are allocated with SQLAllocHandle and freed with SQLFreeHandle.

5.3.2 Connection Handles

A connection consists of a driver and a data source. A connection handle identifies each connection. The connection handle defines not only which driver to use but which data source to use with that driver. Within a segment of code that implements ODBC (the Driver Manager or a driver), the connection handle identifies a structure that contains connection information, such as the following:

- The state of the connection
- The current connection-level diagnostics
- The handles of statements and descriptors currently allocated on the connection
- The current settings of each connection attribute

ODBC does not prevent multiple simultaneous connections, if the driver supports them. Therefore, in a particular ODBC environment, multiple connection handles might point to a variety of drivers and data sources, to the same driver and a variety of data sources, or even to multiple connections to the same driver and data source. Some drivers limit the number of active connections they support; the SQL_MAX_DRIVER_CONNECTIONS option in SQLGetInfo specifies how many active connections a particular driver supports.

Connection handles are primarily used when connecting to the data source (SQLConnect, SQLDriverConnect, or SQLBrowseConnect), disconnecting from the data source (SQLDisconnect), getting information about the driver and data source (SQLGetInfo), retrieving diagnostics (SQLGetDiagRec), and performing transactions (SQLEndTran). They are also used when setting and getting connection attributes (SQLSetConnectAttr and SQLGetConnectAttr) and when getting the native format of an SQL statement (SQLNativeSql).

Connection handles are allocated with SQLAllocHandle and freed with SQLFreeHandle.

5.3.3 Statement Handles

A statement is most easily thought of as an SQL statement, such as SELECT * FROM Employee. However, a statement is more than just an SQL statement — it consists of all of the information associated with that SQL statement, such as any result sets created by the statement and parameters used in the execution of the statement. A statement does not even need to have an application-defined SQL statement. For example, when a catalog function such as SQLTables is executed on a statement, it executes a predefined SQL statement that returns a list of table names.

Each statement is identified by a statement handle. A statement is associated with a single connection, and there can be multiple statements on that connection. Some drivers limit the number of active statements they support. A statement is defined to be active if it has results pending, where results are either a result set or the count of rows affected by an INSERT, UPDATE, or DELETE statement, or data is being sent with multiple calls to SQLPutData.

Within a piece of code that implements ODBC (the Driver Manager or a driver), the statement handle identifies a structure that contains statement information, such as:

- The statement's state
- The current statement-level diagnostics
- The addresses of the application variables bound to the statement's parameters and result set columns
- The current settings of each statement attribute

Statement handles are used in most ODBC functions. Notably, they are used in the functions to bind parameters and result set columns (SQLBindParameter and SQLBindCol), prepare and execute statements (SQLPrepare, SQLExecute, and SQLExecDirect), retrieve metadata (SQLColAttribute and SQLDescribeCol), fetch results (SQLFetch), and retrieve diagnostics (SQLGetDiagRec). They are also used in catalog functions (SQLColumns, SQLTables, and so on) and a number of other functions.

Statement handles are allocated with SQLAllocHandle and freed with SQLFreeHandle.

5.3.4 Descriptor Handles

A descriptor is a collection of metadata that describes the parameters of an SQL statement or the columns of a result set, as seen by the application or driver (also known as the implementation). Thus, a descriptor can fill any of four roles:

- Application Parameter Descriptor (APD). Contains information about the application buffers bound to the parameters in an SQL statement, such as their addresses, lengths, and C data types.
- Implementation Parameter Descriptor (IPD). Contains information about the parameters in an SQL statement, such as their SQL data types, lengths, and nullability.
- Application Row Descriptor (ARD). Contains information about the application buffers bound to the columns in a result set, such as their addresses, lengths, and C data types.
- Implementation Row Descriptor (IRD). Contains information about the columns in a result set, such as their SQL data types, lengths, and nullability.

Four descriptors (one filling each role) are allocated automatically when a statement is allocated. These are known as automatically allocated descriptors and are always associated with that statement. Applications can also allocate descriptors with SQLAllocHandle. These are known as explicitly allocated descriptors. They are allocated on a connection and can be associated with one or more statements on that connection to fulfill the role of an APD or ARD on those statements.

Most operations in ODBC can be performed without explicit use of descriptors by the application. However, descriptors provide a convenient shortcut for some operations. For example, suppose an application wants to insert data from two different sets of buffers. To use the first set of buffers, it would repeatedly call SQLBindParameter to bind them to the parameters in an INSERT statement and then execute the statement. To use the second set of buffers, it would repeat this process. Alternatively, it could set up bindings to the first set of buffers in one descriptor and to the second set of buffers in another descriptor. To switch between the sets of bindings, the application would simply call SQLSetStmtAttr and associate the correct descriptor with the statement as the APD.

5.3.5 State Transitions

ODBC defines discrete states for each environment, each connection, and each statement. For example, the environment has three possible states: Unallocated (in which no environment is allocated), Allocated (in which an environment is allocated but no connections are allocated), and Connection (in which an environment and one or more connections are allocated). Connections have seven possible states; statements have 13 possible states.

A particular item, as identified by its handle, moves from one state to another when the application calls a certain function or functions and passes the handle to that item. Such movement is called a state transition. For example, allocating an environment handle with SQLAllocHandle moves the environment from Unallocated to Allocated, and freeing that handle with SQLFreeHandle returns it from Allocated to Unallocated. ODBC defines a limited number of legal state transitions, which is another way of saying that functions must be called in a certain order.

Some functions, such as SQLGetConnectAttr, do not affect state at all. Other functions affect the state of a single item. For example, SQLDisconnect moves a connection from a Connection state to an Allocated state. Finally, some functions affect the state of more than one item. For example, allocating a connection handle with SQLAllocHandle moves a connection from an Unallocated to an Allocated state and moves the environment from an Allocated to a Connection state.

If an application calls a function out of order, the function returns a state transition error. For example, if an environment is in a Connection state and the application calls SQLFreeHandle with that environment handle, SQLFreeHandle returns SQLSTATE HY010 (Function sequence error), because it can be called only when the environment is in an Allocated state. By defining this as an invalid state transition, ODBC prevents the application from freeing the environment while there are active connections.

Some state transitions are inherent in the design of ODBC. For example, it is not possible to allocate a connection handle without first allocating an environment handle, because the function that allocates a connection handle requires an environment handle. Other state transitions are enforced by the Driver Manager and the drivers. For example, SQLExecute executes a prepared statement. If the statement handle passed to it is not in a Prepared state, SQLExecute returns SQLSTATE HY010 (Function sequence error).

From the application's point of view, state transitions are usually straightforward: Legal state transitions tend to go hand-in-hand with the flow of a well-written application. State transitions are more complex for the Driver Manager and the drivers because they must track the state of the environment, each connection, and each statement. Most of this work is done by the Driver Manager; most of the work that must be done by drivers occurs with statements with pending results.

5.4 Multithreading

On multithread operating systems, drivers must be thread-safe. That is, it must be possible for applications to use the same handle on more than one thread. How this is achieved is driver-specific, and it is likely that drivers will serialize any attempts to concurrently use the same handle on two different threads.

Applications commonly use multiple threads instead of asynchronous processing. The application creates a separate thread, calls an ODBC function on it, and then continues processing on the main thread. Rather than having to continually poll the asynchronous function, as is the case when the SQL_ATTR_ASYNC_ENABLE statement attribute is used, the application can simply let the newly created thread finish.

Functions that accept a statement handle and are running on one thread can be canceled by calling SQLCancel with the same statement handle from another thread. Although drivers should not serialize the use of SQLCancel in this manner, there is no guarantee that calling SQLCancel will actually cancel the function running on the other thread.

Part VI

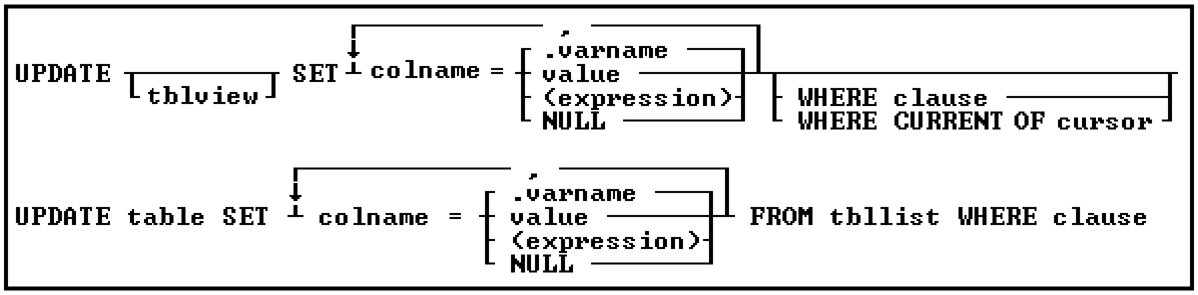
6 R:BASE Database Commands

This chapter provides general reference information about Oterro Database commands, syntax diagrams, and examples of their usage. Entries are listed in alphabetical order.

The commands and the examples provided are described as stand-alone commands. All must be executed as SQL statements through the function calls SQLExecDirect, SQLExecute, or the DBA utility. When using SQLExecDirect or SQLExecute, the commands must be passed through the functions as character strings. The discussions and examples provided are simplified by not specifying the exact API function syntax. Since these commands are executed through the SQLExecDirect, or SQLExecute functions, an Oterro database must be connected before the command is sent

6.1 Reading Command Syntax

A command syntax is read from left to right, and the required parts of the command syntax form a main line. When a required part of a command can vary, brackets are drawn around the available choices and the main line of the command connects with the information in brackets. The optional portion of a command appears below the main line. The following is an example of a command syntax:

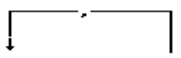


The required parts of the above command are the command keywords UPDATE and SET, the column to be modified, and the new value, expression (expression), or constant. The table name is usually specified, but when left out, the command updates the column in all tables where the column appears. A WHERE clause is optional and allows you to limit rows of data for the command.

Text Objects in the Command Syntax Diagram

- Keywords, which are uppercase, tell R:BASE what to do.
- Arguments, which are lowercase, represent specific information that you provide, such as a table name, column name, variable name, or expression.

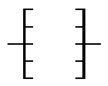
Graphical Parts of the Command Syntax Diagram



An arrow in a command syntax indicates what portion of the command can be repeated. Each part of the command that is repeated must be separated with a comma, or the current delimiter character.

...

Ellipses indicate that the syntax continues to the next line.



If you have a choice of keyword or argument to use, the choices are enclosed in brackets.

This is the main line of the syntax. Any keywords or arguments on the main line are required.

[ ]

This part of the syntax is below the main line and is therefore optional.

The Oterro database reserves some keywords for its use—do not use those keywords for column, table, or view names. For a list of reserved words, see Reserved Words.

6.2 Command Categories

Configuration

AUTONUM	SET		

Control Structures

BREAK	CONTINUE	GOTO	IF/ENDIF
LABEL	SWITCH/ENDSW	WHILE/ENDWHILE	

File Access

ATTACH	DETACH	SATTACH	SCONNECT
SDETACH	SDISCONNECT		

SQL / Query Language

ALTER TABLE	APPEND	CLOSE	COMMENT ON
CREATE INDEX	CREATE SCHEMA	CREATE TABLE	CREATE VIEW
DELETE	DELETE DUPLICATES	DECLARE CURSOR	DROP
FETCH	GRANT	INSERT	LOAD
OPEN	ORDER BY	PROJECT	RENAME
REVOKE	RULES	SELECT	UPDATE
WHERE	GROUP BY	HAVING	

Output Devices

OUTPUT			

Stored Procedures

CALL	GET	PUT	

Utilities

AUTOCHK	CONVERT	LAUNCH	MIGRATE
PACK	RELOAD	TURBO	UNLOAD

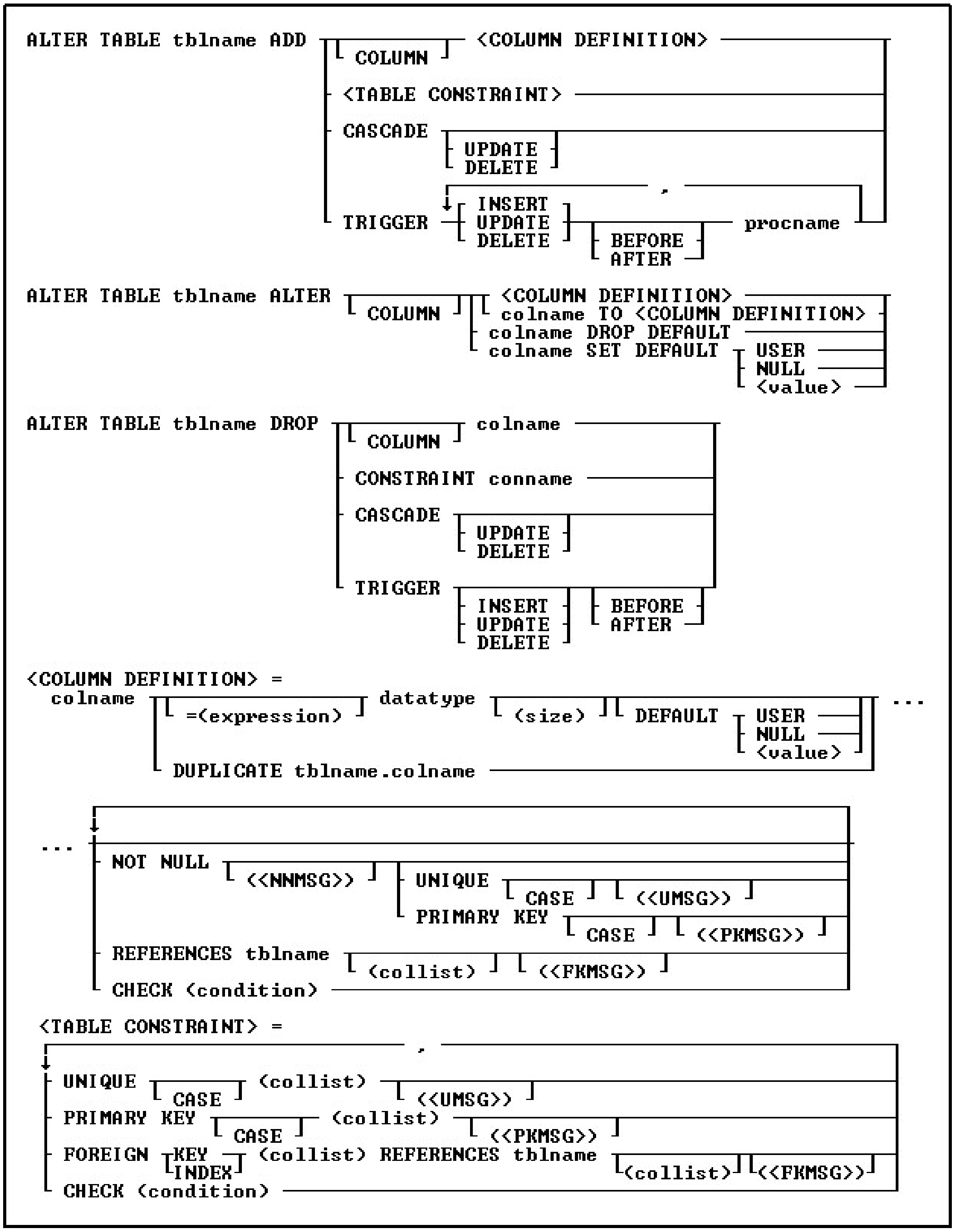
Variable Handling

CLEAR	SET STATICVAR	SET VARIABLE	

6.3 A

6.3.1 ALTER TABLE

Use the ALTER TABLE command to modify an existing table.



Options

,
Indicates that this part of the command is repeatable.

ADD
Specifies the column and its definition, or a table constraint to add.

ADD CASCADE
Maintains primary/foreign key relationships automatically. For example, if you either UPDATE or DELETE a primary key value from a table, the corresponding foreign key values are updated or deleted automatically. A CASCADE can be applied to UPDATE, DELETE or BOTH to specific primary keys. By not specifying either UPDATE or DELETE, both CASCADE restrictions will be enforced upon the primary/foreign key tables. Separate UPDATE and DELETE data restrictions can allow a CASCADE to be enforced for records that are updated, but not enforced when records are deleted, in order to avoid an accidental or undesired record delete. CASCADE can only be added to tables with primary keys.

ADD TRIGGER
Adds the specified triggers to the table. Triggers run a Stored Procedure when an UPDATE, DELETE, or INSERT is executed. If you are using BEFORE and AFTER triggers, BOTH must be ADDed at the same time.

AFTER
Specifies the AFTER trigger event to activate or drop the INSERT, UPDATE or DELETE action.

ALTER
Modifies a column definition.

BEFORE
Specifies the BEFORE trigger event to activate or drop the INSERT, UPDATE or DELETE action. This is the default setting when creating a trigger, if the BEFORE/AFTER parameter is unused.

CASE
Specifies that the data values will be case sensitive.

CHECK (condition)
Sets a condition to be satisfied before an update or insertion of a row can occur, which creates an R:BASE rule.

(collist)
Specifies a list of one or more column names, separated by a comma (or the current delimiter), used in the unique key specification. This option is only used when referencing a unique key.

colname
Specifies a column name. The column name is limited to 128 characters.

COLUMN
Specifies the column to add, drop, or alter.

conname
Specifies a constraint name.

datatype
Specifies an R:BASE data type.

DEFAULT
Specifies a default value for the column if no value is provided by the user.

DROP
Removes a column or a constraint. A column, including both its structure and data, is removed from the table. Dropping a constraint removes a primary key, foreign key, unique key, or a not-null constraint.

DROP CASCADE
Disables the CASCADE feature so that primary/foreign key relationships are not maintained automatically.

DROP CONSTRAINT
Removes a constraint.

Copyright © 1982-2025 R:BASE Technologies, Inc.

DROP DEFAULT
Removes a column's default value.

DROP TRIGGER
Drops triggers from a table. If the INSERT, UPDATE, or DELETE actions is not specified, all triggers are dropped from the table. If the BEFORE or AFTER events are not specified for an INSERT, UPDATE, or DELETE action, both BEFORE and AFTER triggers for the specified action are dropped from the table.

= (expression)
Determines a value using a text or arithmetic formula. The expression can include other columns from the table, constant values, functions, or system variables such as #date, #time, and #pi.

(<FKMSG>)
Creates a constraint violation message to appear whenever a foreign-key data integrity violation occurs. The message can suit the meaning of your data, such as "You must enter a valid number" for a foreign-key constraint violation. You can define two messages: one for inserting and one for updating. A constraint must be dropped, then recreated in order to modify the violation message.

FOREIGN INDEX
With the FASTFK setting on, creates a foreign key that has an index using row pointers for data retrieval on selected columns.

FOREIGN KEY
Specifies a column or set of columns required to match values in a particular primary key or unique key constraint defined in a table.

FOREIGN KEY (collist)
If (collist) comprises one column, this option is equivalent to FOREIGN KEY. If two or more columns are included in (collist), the values in the listed columns must be unique as a group in each row. Each column must be separated by a comma (or the current delimiter).

(<NNMSG>)
Creates a constraint violation message to appear whenever a not-null data integrity violation occurs. The message can suit the meaning of your data, such as "You must enter a valid number" for a not-null constraint violation. A constraint must be dropped, then recreated in order to modify the violation message.

NOCHECK
Optional NOCHECK parameter does not update references to views, tables, and columns in forms, reports, labels, access rights, and rules. In this case, user assumes the responsibilities to update any references to views, tables, and columns in forms, reports, labels, access rights, and rules. This condition is ONLY available for the **ALTER COLUMN** command.

NOT NULL
Prevents a column from accepting null values, but permits it to accept duplicate values. If this option is specified without a setting for a default value, you cannot insert rows without specifying values for the given column.

(<PKMSG>)
Creates a constraint violation message to appear whenever a primary-key data integrity violation occurs. The message can suit the meaning of your data, such as "You must enter a valid number" for a primary-key constraint violation. You can define three messages: one for uniqueness, one for deleting, and one for updating. A constraint must be dropped, then recreated in order to modify the violation message.

PRIMARY KEY
Specifies the column(s) to designate as a primary key constraint.

PRIMARY KEY (collist)
If (collist) comprises one column, this option is equivalent to PRIMARY KEY. If two or more columns are included in (collist), the values in the listed columns must be unique as a group in each row. Only columns defined as not null can be included in (collist). Each column must be separated by a comma (or the current delimiter).

procname
The procedure name. If a procedure by this name already exists in the database, an error is generated.

REFERENCES
Identifies the primary key or unique key table to which the foreign key refers.

SET DEFAULT
Changes a column's default value.

(size)
Defines the length of a column of the TEXT data type (if not the default 8). Defines the precision and scale of a column of the DECIMAL or NUMERIC data type, if not the default of precision 9 and scale 0 (9,0). VARBIT, VARCHAR, and BIT either require or can have a size.

tblname
Specifies a table name. The table name is limited to 128 characters.

(<UMSG>)
Creates a constraint violation message to appear whenever a unique-key data integrity violation occurs. The message can suit the meaning of your data, such as "You must enter a valid number" for a unique-key constraint violation. You can define three messages: one for uniqueness, one for deleting, and one for updating. A constraint must be dropped, then recreated in order to modify the violation message.

UNIQUE
Requires the values in a column to be unique by creating a unique key constraint.

UNIQUE (collist)
If (collist) is one column, this option is equivalent to UNIQUE. If two or more columns are included in (collist), the values in the listed columns must be unique as a group in each row. Only columns defined as not null can be included in (collist). Each column must be separated by a comma (or the current delimiter).

USER
NULL
(value)
Default USER: Specifies the default value to be the user identifier.
Default NULL: Specifies the default value to be null.
Default (value): Specifies the default to be the indicated value.

About the ALTER TABLE Command

ALTER TABLE creates a temporary internal table by copying a table's structure and data. You must have enough disk space to hold another copy of a table, and your database should not exceed the number of tables and columns R:BASE allows, which includes user-defined tables and system tables.

After the ALTER TABLE command has been executed, the temporary table goes away; however, the disk space the temporary table occupied is not available. To recover this space, pack or reload the database using the PACK or RELOAD commands.

Adding Columns

When you add a new column to a database, specify the name, data type, and length when the data type for the column is TEXT, or precision and scale when the data type for the column is DECIMAL or NUMERIC. When the column is computed, specify the name and expression-a data type is optional. When the column already exists in the database, specify only the name-R:BASE uses the existing data type, and length, if applicable.

Database Access Rights with ALTER TABLE

When access rights for a table or view have been assigned using the GRANT command, ALTER TABLE requires the database owner's user-identifier or permission from the owner to alter specific tables.

Limitations of the ALTER TABLE Command

You cannot assign an index to a new column or transfer the index of an existing column with ALTER TABLE. If the added column should be indexed, use the CREATE INDEX command.

You also cannot add or transfer rules with ALTER TABLE. If you want a rule to apply to a column in the table, you must add it with the RULES command or use the Database Designer.

You cannot use ALTER TABLE to modify a view.

You cannot add a foreign key to a temporary table.

Examples

The following command adds mailadrs, a TEXT column 40 characters wide, at the end (or far right) of the customer table.

```
ALTER TABLE customer ADD mailadrs TEXT (40)
```

The following command adds the profit column at the end of product table. The value of profit is computed from the current row values for listprice multiplied by 1.05. The data type specified is REAL.

```
ALTER TABLE product ADD profit=(listprice * 1.05) REAL
```

The following command adds an "update only" cascade to the employee table.

```
ALTER TABLE employee ADD CASCADE UPDATE
```

The following command defines columns one through three as a case-sensitive primary key. Before you use this command, you must add a not-null constraint to each of the columns.

```
ALTER TABLE tablename ADD PRIMARY KEY CASE (column1, column2, column3) ('This is a
message from the primary key')
```

The following command line adds a foreign index to the custidcolumn and references the primary key in the customer table.

```
ALTER TABLE transmaster ADD FOREIGN INDEX (custid) REFERENCES customer
```

The following command drops the before insert trigger in the InvoiceHeader table.

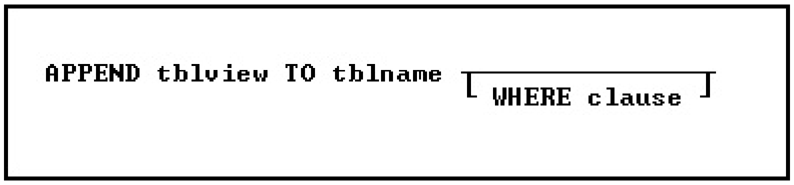
```
ALTER TABLE InvoiceHeader DROP TRIGGER INSERT BEFORE
```

The following command line adds the test trigger for the SampleTriggers Table.

```
DROP PROCEDURE MySampleTrigger
PUT AFTER.PRC AS MySampleTrigger
ALTER TABLE SampleTriggers ADD TRIGGER INSERT AFTER MySampleTrigger
RETURN
```

6.3.2 APPEND

Use the APPEND command to copy rows from a table or view to the end of a table.



Options

tblview
Names the table or view from which you want to copy rows-the source.

TO tblname
The name of the table to which you want to copy rows-the destination.

WHERE clause
Limits rows of data. For more information, see WHERE.

About the APPEND Command

R:BASE only copies values from the source table or view that have matching column names in the destination table. Columns in the destination table that are not in the source table or view are filled with null values.

Rows are copied, not removed, from the source.

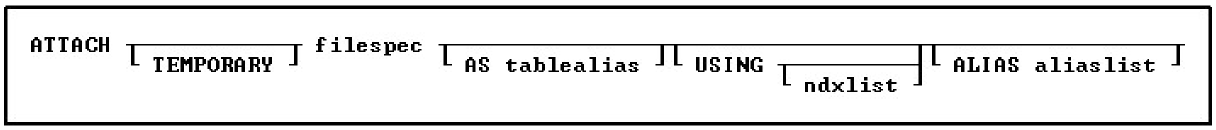
Example

The following command adds the rows containing new employee information from the newemp table to the end of emptable, a table containing information about previous employees. A WHERE clause is not specified, so all rows are copied to emptable.

```
APPEND newemp TO emptable
```

6.3.3 ATTACH

Use the ATTACH command to attach a dBASE file to an open R:BASE database.



Options

ALIAS AliasList
To specify alias names for columns.

AS tablealias
Specifies an alias, or temporary name, for the dBASE table. A table alias is sometimes required when attaching files that do not follow the same table name restrictions as R:BASE.

filespec
A dBASE database name with a drive and path specification in the form D:\PATHNAME\FILENAME.

ndxlist
Specifies a list of index files to associate with the specified dBASE file. You do not have to include the extension for each index file. Separate index file names with a comma (or the current delimiter). Index files must be located with the specified dBASE file.

TEMPORARY
Allows you to create a temporary table with the ATTACH command. The temporary tables will disappear when the database is disconnected.

USING
Removes dBASE index files that were previously associated with the attached dBASE file when this option is used without a list of index files.

About the ATTACH Command

Before you can attach a dBASE file, an R:BASE database must be open. You can open an existing database or use the CREATE SCHEMA command to create a database.

Include the file specification when the file is located on a different drive or directory. You do not have to include the .DBF extension for the dBASE file.

R:BASE directly reads and writes dBASE III and dBASE III PLUS data and index files. R:BASE can also read and write dBASE IV data files and index files that have the .NDX extension, just as dBASE III and dBASE III PLUS can share files with dBASE IV. R:BASE cannot read encrypted files nor read and write to dBASE IV index files, which have .MDX extensions.

Attaching to dBASE Files from a Network

From a network, R:BASE, dBASE III, and dBASE III PLUS users can access the same file at the same time. R:BASE can lock a dBASE file just as dBASE III and dBASE III PLUS can lock a dBASE file. When R:BASE is in multi-user mode, it does not support dBASE IV use. When a dBASE IV file is open by dBASE, R:BASE cannot access that file; when R:BASE attaches to a dBASE IV file, that file cannot be accessed by dBASE IV.

Listing dBASE Files

Use the LIST command to list the dBASE files in an R:BASE database. R:BASE displays DBF in the Rows column to indicate a dBASE file.

Reattaching dBASE Files to R:BASE

A dBASE file stays attached unless you use the DETACH command, which removes a dBASE file and its associated index file from the R:BASE database. The dBASE files stay attached because R:BASE remembers the attached dBASE files and index files when you open a database. At that time, R:BASE searches the current directory and path to find the attached dBASE file; therefore, the location of the dBASE files must be included in your path.

Associating and Modifying dBASE Index Files

You can associate a maximum of seven dBASE index files, which have .NDX extensions, with a dBASE data file by using the ndxlist option. R:BASE remembers each index file you associate with the dBASE data file. Also, use the ndxlist option to modify or preserve a set of indexes. If you issue another ATTACH command with a list of index files, R:BASE removes the current index files from the dBASE data file and associates the new list with the dBASE data file.

R:BASE updates the information stored in the dBASE data and index files each time you add or edit information in a dBASE file.

R:BASE Commands that Work with dBASE

The following R:BASE commands work with dBASE files. Limitations are noted following the table.

Commands that Work with dBASE			
ATTACH	DROP LABEL	LIST CURSORS	RENAME FORM
BROWSE	DROP REPORT	LIST DATABASES	RENAME OWNER
CHOOSE	DROP RULE	LIST FORMS	RENAME REPORT
COMMENT ON	DROP TABLE	LIST LABELS	RENAME VIEW
COMMIT *(4)*	DROP VIEW	LIST REPORTS	REPORTS
COMPUTE	EDIT	LIST RULES	REVOKE
CONNECT	EDIT DISTINCT	LIST TABLES	ROLLBACK *(4)*
CONTINUE	EDIT USING	LIST VIEWS	RULES
CREATE VIEW	ENTER USING	LOAD	SELECT
CROSSTAB	FETCH	OPEN CURSOR	SET
DECLARE CURSOR	FUNCTIONS	PACK *(1)*	SHOW
DELETE	GRANT	PRINT	TALLY
DELETE DUPLICATES	INSERT INTO	PROJECT *(2)*	UNLOAD *(3)*
DETACH	LBLPRINT	QUERY	UPDATE
DISCONNECT	LIST ACCESS	RBLABELS	WHENEVER
DROP CURSOR	LIST ALL	RELOAD	ZIP
DROP FORM	LIST COLUMNS	RENAME	COLUMN

Notes:

- dBASE files are not affected when you use a PACK command.
- Using the PROJECT command, you can create a new table from an existing table from dBASE to R:BASE, but not from R:BASE to dBASE.
- You can unload dBASE tables as ASCII only.
- You cannot modify dBASE tables when transaction processing is on.
- dBASE memo fields can be 64K in size. If the dBASE memo field is larger than 4K (the maximum size of an R:BASE note column), R:BASE reads as much as will fit. If you make changes and then write the record back to dBASE, the existing dBASE memo field is overwritten. The Carriage Return and Line Feed characters in dBASE are mapped to [Alt] + [0255].

R:BASE Commands that Do Not Work with dBASE

The following commands do not work with dBASE files in R:BASE.

Commands that Do Not Work with dBASE		
ALTER TABLE *	CREATE INDEX ON	JOIN
APPEND	CREATE SCHEMA AUTHOR	RENAME TABLE
AUTONUM	CREATE TABLE	RESTORE
BACKUP ALL	DROP COLUMN	SUBTRACT
BACKUP DATA	DROP INDEX	UNION
BACKUP STRUCTURE	INTERSECT	

* Column names for dBASE files can be changed with ALTER TABLE.

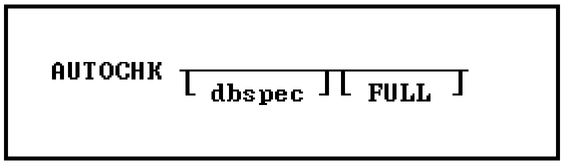
Example

In the following example, the first command opens the concomp database. The second command attaches the dBASE file SAMPGATE to the concomp database and associates the dBASE index files COMPID and PRODDESC with the R:BASE file table sampgate.

```
CONNECT concomp
ATTACH sampgate USING compid, proddesc
```

6.3.4 AUTOCHK

Use the AUTOCHK command to check the integrity of a database. AUTOCHK can be used when connected or disconnected from the database.



Options

dbspec
Specifies a database other than the open database to check; otherwise, the open database is checked.

FULL
Provides detailed information about the processing being performed, and when AUTOCHK encounters an error, it continues processing.

About the AUTOCHK Command

Use the AUTOCHK command to ensure that the connected database is intact before using the PACK or RELOAD commands, or before making a backup of the database with either the BACKUP or COPY commands.

Please Note: If any user connected to the database has temporary tables or views created you may receive an abnormal amount of errors. This is expected and is a side effect of having temporary tables active during the check. For completely accurate results, have all users disconnect from the database to be checked.

AUTOCHK checks the following:

- The structure-file block sizes and locations.
- The timestamps for all database files.
- The database-file lengths.
- The number of tables and columns.
- The starting and ending pointers for tables.
- The location of columns.
- The File 4 data pointers.
- The data types of columns.
- The size and number of rows in each table.
- The row pointers in the data file.

AUTOCHK does not check indexes.

When you run AUTOCHK, it systematically checks the structure file of the open database, and the data files. AUTOCHK only checks the index file for the timestamp and length of the file. When opening a database, AUTOCHK ignores any user-identifier protection. AUTOCHK without the FULL option sets the R:BASE error variable to a non-zero value if errors are found.

The results of AUTOCHK with the FULL option are displayed on screen, or the current output device. First, AUTOCHK validates the timestamps in the database files, then systematically checks the structure of each table and view in the database, providing a list of columns, constraints, and indexes for each. Any structure errors are noted after each table listing.

Database Statistics

Next, AUTOCHK checks the data for each table, listing active rows and deleted rows. Any problems with data, such as broken pointers, are listed after the respective tables. Finally, AUTOCHK provides a summary of the database structure, including the number of tables, columns, and indexes, and the actual space that the data occupies in the data file (File 2). AUTOCHK shows the percent of space used for the items in each list to give an idea of how much space has been used, and to indicate the need to recover space in the database files. Any numbers less than 100 percent indicate the need to pack or reload the database using the PACK or RELOAD commands.

The following section contains information about using AUTOCHK in application files and capturing the error variables returned. This allows the application developer to prevent users from continuing to use a corrupted database.

```
SET ERROR VAR E1
WRITE 'Checking database for errors...'
AUTOCHK dbname
IF E1 > 40 THEN
   WRITE 'AUTOCHK has found errors in the database!'
   BEEP
ENDIF
If E1 > 0 and E1 < 50 THEN
   WRITE 'AUTOCHK will not run - User Abort or Out of Memory'
   BEEP
ENDIF
IF E1 = 0 THEN
```

```
   WRITE 'AUTOCHK successful - No errors found'
ENDIF
PAUSE 2
RETURN
```

If AUTOCHK with no option finds an error, it stops checking the database and displays one error message. If the error message (see list below) indicates that the database is damaged, you might want to start using a backup copy of the database. Alternatively, you might want to use R:SCOPE, a database repair tool available from R:BASE Technologies, Inc.

If AUTOCHK finds no errors, it displays the message "NO ERRORS FOUND." If you press any key while AUTOCHK is checking the database, the program stops and displays the message "USER ABORT." AUTOCHK automatically sets the error variable to the number corresponding to the message returned. For example, if the error "UNABLE TO OPEN DATABASE FILE 2" is returned, the error variable is set to 52.

Multi-User Databases

Use caution when running AUTOCHK in a multi-user environment. If the database being checked is currently open with MULTI set on, AUTOCHK places a database lock on the database. The database lock remains in effect until AUTOCHK stops checking the database. Database users are unable to make any changes to the data or structure of the database while this lock is in place.

If a user attempts to open a database being checked by AUTOCHK and the database does not have any other users, the user receives an error message indicating that the database is currently open in a mode that makes it unavailable. If other users have the database open with the MULTI set on and the database is being checked, the user attempting to open the database receives a message indicating the user is waiting in a lock queue. If AUTOCHK successfully completes checking the database and finds no errors, it reports that no errors were found and sets the error variable to 0.

Checking continues in multi-user mode (even if a database lock cannot be obtained) if a database is connected by another user; however, row errors in File 2 can occur because of database activity.

Error Messages

AUTOCHK displays one of the following messages when it is unable to start checking or complete checking the database or when it finds an error in the database files. AUTOCHK returns *0 No errors found* if the database is okay. Some of these messages indicate that the database is damaged. Either switch to a backup copy of the database, or attempt repair of the database using R:SCOPE, R:BASE Technologies's database repair tool. If AUTOCHK is unable to open File 1 of the database, check that the path you specified to the database is correct; or, if you are trying a multi-user database, check that no other user has the database connected with MULTI set off.

Checking continues in multi-user mode (even if a database lock cannot be obtained) if a database is connected by another user; however, row errors in File 2 might occur because of database activity.

Any of these messages, except the first (code 0), indicates that the database is damaged. Either switch to a backup copy of the database, or attempt repair of the database using R:SCOPE, R:BASE Technologies's database repair tool.

AUTOCHK Error Messages

Number	**Code Message**
0	No errors found
1	This database is not of the correct version
2	The database filenames must all match
20	Out of memory
40	User Abort
50	Unable to open database file number 1
51	Unable to lock this database
52	Unable to open database number 2
53	Unable to open database number 3
54	Unable to open database number 4
55	Error reading the database information block

56	Error reading the timestamp information
57	Timestamp in file number 2 does not match file 1; run RBSYNC
58	Timestamp in file number 3 does not match file 1; run RBSYNC
59	Timestamp in file number 4 does not match file 1; run RBSYNC
60	Invalid number of tables
61	Invalid number of columns
62	Invalid number of indexes
63	File 1 is too small
70	Error in database structure block
80	Error reading the table list
81	Error reading the column list
82	Error reading the index list
100	Incorrect version flag
101	Error reading Case Folding and Collating tables
110	Error in DBinfo block offset
111	Error in DBinfo block length
120	Error in length of database file 2
121	Error in length of database file 3

Example

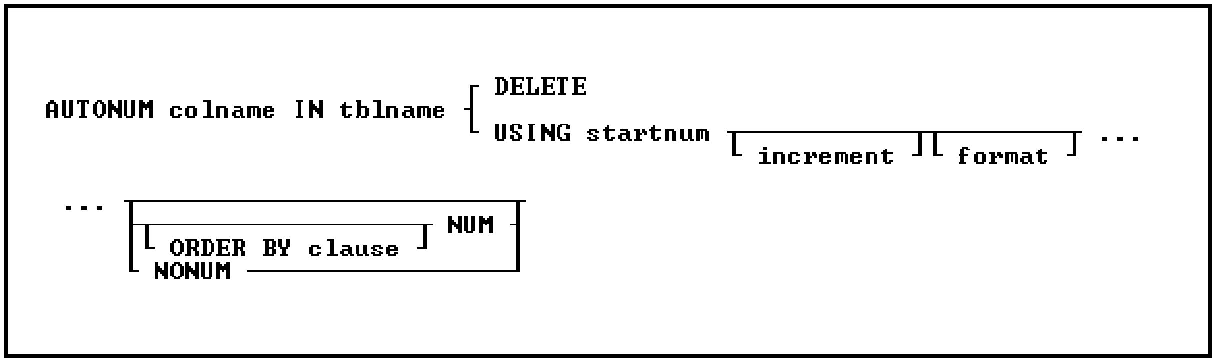
The following is an example of how to put AUTOCHK results in a file for viewing:

```
DISC
OUTPUT dbname.chk
AUTOCHK dbname FULL
OUTPUT SCREEN
```

You can view DBNAME.CHK in the Text Editor to view the results.

6.3.5 AUTONUM

Use the AUTONUM command to define, modify, or remove an autonumber formula from a column.



Options

colname
Specifies a column name. The column name is limited to 128 characters.

In a command, you can enter *#c*, where *#c* is the column number shown when the columns are listed with the LIST TABLES command.

DELETE
Removes a column's autonumber formula.

format

Defines the format in which values are displayed. This option is used only for columns with the TEXT data type.
You can use the following formatting characters:

Formatting Character	Result
9	Specify a numeric digit; leading zeros are suppressed.
0	Specify a numeric digit; leading zeros are displayed.
. (period)	Aligns digits along a decimal point.
[ ] (square brackets)	Encloses literal text.

For example, if the format is [MX]9999 and the numeric value is 123, the value entered will be MX123.

IN tblname
Specifies the table in which to autonumber the column.

increment
Specifies the value of the increment as each new row is added to the table. The default increment is 1.

NONUM
Leaves existing values unchanged and assigns autonumbered values to new rows as they are added to the table. NONUM is the default option.

NUM
Renumbers all the existing values in the column defined as an autonumbered column.

ORDER BY clause
Sorts rows of data. The ORDER BY clause is only used with the NUM option.

USING startnum
Defines or redefines the formula for an autonumber column. You must specify a starting value. Optionally, you can specify an increment, and for columns with the TEXT data type, a display format. For a column in a table that contains values, you can either renumber existing values or leave them as they are.

About the AUTONUM Command

An autonumbered column ensures that each row in that column has an incremental value. For example, use an autonumbered column to assign identification numbers, model numbers, or invoice numbers:

The following types of columns can be autonumbered:

- Columns that are not computed.
- Columns with DOUBLE, INTEGER, NUMERIC, REAL, or TEXT data types.

When you use the LIST command to list information about a column or table, autonumbered columns are described as AUTONUMBER in the *attributes* column.

Automatic Numbering

R:BASE automatically enters values in an autonumbered column when you add rows to a table using a form, the Data Editor, INSERT command, or LOAD command with the NUM option. When you import rows to a table that contains an autonumber column, you can either set autonumbering off and load imported values, or set autonumbering on and let R:BASE autonumber the values.

Capturing the Autonumbered Value

The next value for an autonumbered column can be captured for extended calculation or for display in a form. To capture the value, use the NEXT Function.

Changing Values

You can change the values in an autonumbered column by using a form, or the UPDATE or EDIT command. However, if you change a value in an autonumbered column, you could assign a duplicate number or disrupt the sequence of numbers. For more information about changing values in an autonumbered column, see the below guidelines in Renumbering Columns Containing Data.

Renumbering Columns Containing Data

If you renumber a column that contains data, use the following guidelines to decide whether to change the column's existing values.

Autonumber?	Option to Use	Conditions
Yes	NUM	A column exists in only one table in the database. You can use the ORDER BY NUM clause to sort the rows in the order in which you want them renumbered. When you add new rows, values are numbered in the order in which the rows are added to the table.
No	NONUM (or do not specify)	A linking (or common) column exists in more than one table. You will destroy the common column values that link your tables if you renumber the values in a linking column. R:BASE adds autonumbered values to new rows as you add them to the table.

Redefining Formulas

You can redefine the formula for an autonumbered column. For example, use the AUTONUM command with the NUM option to change a column's display format from suppressing leading zeros to displaying them. For more information about redefining formulas of an autonumbered column, see "Renumbering Columns" earlier in this entry.

Removing Formulas

To remove an autonumber formula for a column, use the DELETE option. R:BASE removes only the formula, not the existing values in the autonumbered column. After you remove an autonumber formula, the user must enter values in the column as rows are added.

Autonumbering Tables Created with Relational Commands

When you create a table with the PROJECT command, R:BASE transfers an autonumbered column as a regular column. You must define an autonumber formula for the column in the new table.

Database Access Rights with AUTONUM

When access rights for a table have been assigned using the GRANT command, AUTONUM requires either the database owner's user identifier, or the rights to alter a table.

Examples

The following command defines an autonumber formula for the *custid* column in the *customer* table. Existing values are renumbered starting at 100; assigned values increase by one for each row. Only use this command for a column that meets the renumbering guidelines in the section "Renumbering Columns Containing Data."

```
AUTONUM custid IN customer USING 100 1 NUM
```

The following command defines an autonumber formula for the *model* column in the *product* table. Existing values are not renumbered. Values in new rows are numbered starting with 100. Assigned values increase by one each time a row is added. The numbering format specifies that the letters MX always precede the numeric value. The 0000 provides space for a numeric value of up to four digits. When the value is less than four digits, R:BASE enters leading zeros.

```
AUTONUM model IN product USING 100 1 [MX]0000 NONUM
```

The following command assigns autonumbering to the *empid* column in the *employee* table. Existing values are renumbered starting at 100; assigned values increase by one for each row. The rows are renumbered by the employees' last and first names. Only use this command for a column that meets the renumbering guidelines in the section "Renumbering Columns Containing Data."

```
AUTONUM empid IN employee USING 100 1 ORDER BY emplname, empfname NUM
```

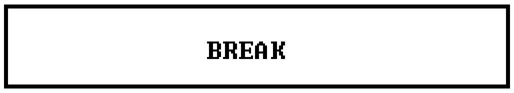
The command below deletes the autonumber formula from the *empid*column in the *employee* table.

```
AUTONUM empid IN employee DELETE
```

6.4 B

6.4.1 BREAK

Use the BREAK command to force an early exit from a WHILE...ENDWHILE loop or a SWITCH...ENDSW structure.



About the BREAK Command

The BREAK command is usually run in an IF...ENDIF structure contained within a WHILE...ENDWHILE loop or a CASE block within a SWITCH...ENDSW structure. The IF conditions indicate when to run the BREAK.

R:BASE exits the currently processing WHILE...ENDWHILE loop or SWITCH...ENDSW structure when a BREAK is encountered, and does not run any further commands in the WHILE loop or SWITCH structure. BREAK decreases the nesting level by one. BREAK passes control to the next line of the command file following the WHILE loop or SWITCH structure.

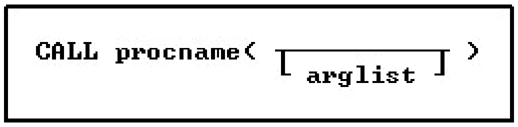
Examples

For an example of using BREAK with WHILE...ENDWHILE, see WHILE...ENDWHILE. For an example of using BREAK with SWITCH...ENDSW, see SWITCH...ENDSW.

6.5 C

6.5.1 CALL

Runs a Stored Procedure.



Syntax

A) As function: `SET VAR vVariable TYPE = (CALL procname(arglist))`

B) As command: `CALL procname(arglist)`

About the CALL Command

The call command is used to invoke a Stored Procedure that was created by using the SET PROCEDURE command. It can be referenced by either a function notation or as a stand-alone command. In either case the argument list must be included. If you wish to include a blank argument list then use an empty pair of parenthesis.

Both methods of using CALL have their advantages. For example, using the function notation allows you to use a Stored Procedure in a computed view or to invoke a Stored Procedure via an SQL statement. On the other hand, used as a command, you will be able to reference the Stored Procedure by itself.

When using the Function notation the return value of the Stored Procedure is stored in the variable itself or displayed in the column (in the case of a computed column in a table or view). When using the Command notation the return value of the Stored Procedure will be placed into the system variable STP_RETURN. STP being an abbreiviation of Stored Procedure.

Examples

In the example below, a view is using a Stored Procedure to calculate values from another table.

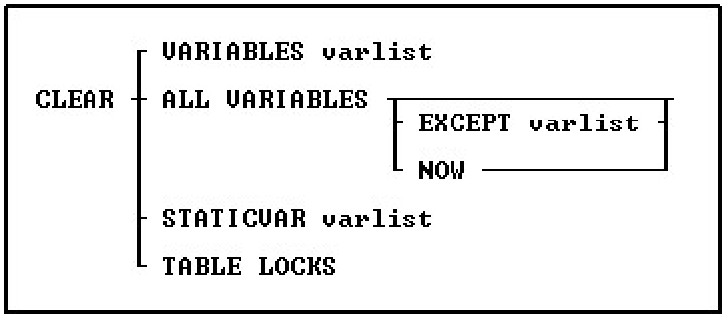
```
CREATE VIEW MonthSum (CustomerID,CustomerSummary) +
AS SELECT T1.CustomerID,(CALL SumUpCust(T1.CustomerID)) FROM Customers T1
```

In the following example, an SQL select statement is used to invoke a maintenance routine from another application using Visual Basic or Oterro. The use of WHERE LIMIT=1 causes the procedure to run once and only once. Without this clause the Stored Procedure would execute once for every matching row in the table. The *AnyTable* can be any table in the database in this case. The only requirement is that we must use a table in order to have a "healthy" SELECT clause.

```
SELECT (CALL DBCheck()) FROM AnyTable WHERE LIMIT=1
```

6.5.2 CLEAR

Use the CLEAR command to remove global variables from memory or clear table locks.



Options

ALL VARIABLES
Removes all global variables from memory.

EXCEPT varlist
Specifies variables that the CLEAR command will not remove. You can use wildcards in variable names.

NOW
Clears all the variable storage blocks (VSBs), including all printer control code variables and other non-permanent system variables (NPSVs). It re-allocates the blocks with just the permanent system variables (PSVs), #DATE, #TIME, #PI and SQLCODE. Without the fourth parameter, printer control code variables and other # variables are NOT cleared. Also, it does not completely re-initialize the memory blocks as does the CLEAR ALL VAR NOW. **The NOW argument is specific to DOS versions of R:BASE.**

STATICVAR varlist
Clears a list of one or more static variables. You can use wildcards in variable names. Static variables are created with the SET STATICVAR command.

TABLE LOCKS
Removes all locks on tables. This command parameter must be used with MULTI set to OFF and while connected to the database.

VARIABLES varlist
Removes a list of one or more variables. Use this option at the end of a complete set of procedures to clear variables that are no longer needed. You can use wildcards in variable names.

About the CLEAR Command

When R:BASE is first loaded into memory, only system variables are defined; they are not affected by the CLEAR command. Other variables you define remain in memory until you exit from R:BASE or use the CLEAR command.

Examples

The following command removes the global variables *vcounter* and *vname* from memory.

```
CLEAR VARIABLES vcounter, vname
```

The following command removes all global variables from memory.

```
CLEAR ALL VARIABLES
```

The following command removes all global variables except *var1*.

```
CLEAR ALL VARIABLES EXCEPT var1
```

The following command clears all variables beginning with the letter *v*.

```
CLEAR VARIABLES v%
```

The following command clears the vStartDate static variable.

```
CLEAR STATICVAR vStartDate
```

6.5.3 CLOSE

Use the CLOSE command to close an open cursor that was defined with the DECLARE CURSOR command.

```
CLOSE cursorname
```

Options

cursorname
Specifies a 1 to 18 character cursor name that has been previously specified by the DECLARE CURSOR command and opened with the OPEN command.

About the CLOSE Command

Cursors are pointers to rows in a table, and are defined using the DECLARE CURSOR command. When you no longer want to use the cursor but want to retain it for later use, use the CLOSE command to close the cursor. When you open the cursor again, it is positioned at the beginning of the set of rows defined by the DECLARE CURSOR command.

Use the LIST CURSOR command to list all of the currently defined cursors and whether the cursor is opened or closed.

When you close a cursor, most of the memory taken by the cursor definition is returned. If the DECLARE CURSOR command used any file handles, they are released.

Example

The following command makes the rows defined by the DECLARE CURSOR command as *cursor1* unavailable. To use the information defined by *cursor1* again, you need to reopen the cursor with the OPEN command.

```
CLOSE cursor1
```

6.5.4 COMMENT

Use comments in command or application files to provide internal program documentation.

There are a few comment designators: "--", "{ }", and "*( )".

1. A **"--"** comment can be used only on a single line either by itself or following a command.

   To comment an individual line, add two hyphen characters **"-"** to the beginning of the line. A carriage return at the end of the line indicates the end of the text for a comment that begins with two hyphens. In R:BASE Editor, the syntax highlighting will alter the display and change the font color to pink and the style to italicized. In the following example;

   ```
   CLEAR VAR vResult
   ```

   the command will become:

   ```
   --CLEAR VAR vResult
   ```

2. A **"{ }"** comment may share a command line with a command, occupy a line itself, or extend over multiple command lines.

   This designator is the recommended option with the latest releases of R:BASE, as it helps in avoiding any confusion when using parentheses with your R:BASE expressions. The set of squiggly brackets **"{}"**, with the desired commented text or commands enclosed within the squiggly brackets will comment the text. In the following example;

   ```
   PLUGIN RPDFMerge 'vResult +
   |ACTION MERGE +
   |DOC_LIST_FILE PDFFilesToMerge.LST +
   |SHOW_SETUP_DIALOG ON +
   |OUTPUT_FILE OneBigMergedFile.PDF '
   ```

   the command(s) will become:

   ```
   {
   PLUGIN RPDFMerge 'vResult +
   |ACTION MERGE +
   |DOC_LIST_FILE PDFFilesToMerge.LST +
   |SHOW_SETUP_DIALOG ON +
   |OUTPUT_FILE OneBigMergedFile.PDF '
   }
   ```

   Keep in mind that any command(s) that are embedded within a multiple-line comment will not be executed.

3. A **"*( )"** comment may share a command line with a command, occupy a line itself, or extend over multiple command lines.

   Another use of characters that will comment your code is the asterisk character preceding a set of parentheses **"*()"**, with the desired commented text or commands enclosed within the parentheses. R:BASE interprets text following an asterisk and left parenthesis as a comment until a closing right parenthesis is reached. If the right parenthesis is not entered, R:BASE responds with a continuation prompt (+>). Enter a closing parentheses until you are returned to the R> Prompt, or other processing. In the following example;

```
PLUGIN RPDFMerge 'vResult +
|ACTION MERGE +
|DOC_LIST_FILE PDFFilesToMerge.LST +
|SHOW_SETUP_DIALOG ON +
|OUTPUT_FILE OneBigMergedFile.PDF '
```

the commands will become:

```
*(
PLUGIN RPDFMerge 'vResult +
|ACTION MERGE +
|DOC_LIST_FILE PDFFilesToMerge.LST +
|SHOW_SETUP_DIALOG ON +
|OUTPUT_FILE OneBigMergedFile.PDF '
)
```

Although this option is still supported in R:BASE, it is now recommended that you use the squiggly bracket **"{}"** method above. Keep in mind that any command(s) that is embedded within a multiple-line comment will not be executed.

Using Comment Designators

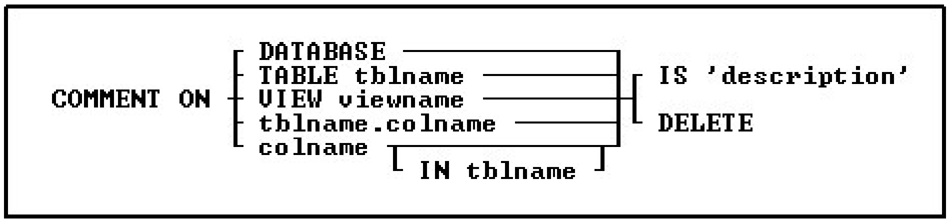
- Although used primarily in command files, you can enter a comment at the R> Prompt.
- If you place a comment on the same line as a command, leave at least one space between the comment and the command so the comment is not interpreted as part of the command.

Restrictions on Using the Comments

- Do not include comments within the text of an ASCII menu file or a menu block because the comment will be read as part of the file.
- Do not embed comments within multi-line commands between continuation characters.
- Once starting a comment with a { bracket, everything except other left and right brackets is ignored until the matching right bracket.
- Once starting a comment with *(, everything except other left and right parentheses is ignored until the matching right parenthesis.

6.5.5 COMMENT ON

Use the COMMENT ON command to add a description to a database, table, view, or column.



Options

colname

Adds a description for a column in all tables in which it appears.

DATABASE
Adds a description for the connected database.

DELETE
Removes a description for a table or for a column in either the specified table or in all tables.

IN tblname
Adds a description for a column only in the specified table.

IS 'description'
Defines a description for a table or for a column in either the specified table or in all tables. The description is limited to 128 characters. The text must be enclosed in quotes using the current QUOTES setting.

TABLE tblname
Adds a description for the specified table.

VIEW viewname
Adds a description for the specified view.

tblname.colname
Specifies a column name. In a command, you can enter *#c*, where *#c* is the column number. In an SQL command, a column name can be preceded by a table or correlation name and a period (*tblname.colname*).

About the COMMENT ON Command

Descriptions must be enclosed in single quotation marks ('), or the current delimiter character for QUOTES.

When you add a description to a column that appears in multiple tables, R:BASE adds the description to the column in every table. If you add a new table containing the column, you must add the description for that column to the new table.

The (CVAL('DBCOMMENT')) function can be used to retrieve the database comment.

Comments are stored in the *SYS_COMMENTS* system table. When a column or table is renamed or removed, R:BASE automatically updates the *SYS_COMMENTS* table to reflect the change.

When access rights for a table have been assigned using the GRANT command, COMMENT ON requires the database owner's user identifier to describe tables and columns.

Examples

The following command adds a description to the *Employee* table.

```
COMMENT ON TABLE Employee IS 'Employee Information'
```

The following command adds a description to the *EmpID* column in all tables in the database.

```
COMMENT ON EmpID IS 'Employee Identification Number'
```

The following commands show two ways to add a description to the *EmpID* column in only the *Employee* table.

```
COMMENT ON Employee.EmpID IS 'Employee Identification Number'
COMMENT ON EmpID IN Employee IS 'Employee Identification Number'
```

The following command removes the description from the *Employee* table.

```
COMMENT ON TABLE Employee DELETE
```

The following command removes the description from the *EmpID* column in every table in which the column occurs.

```
COMMENT ON EmpID DELETE
```

The following command removes the description from the *EmpID* column in the *Employee* table.

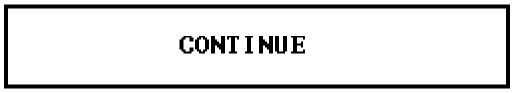
```
COMMENT ON EmpID IN Employee DELETE
```

The following command adds a description to the *UserManagement* database.

```
COMMENT ON DATABASE IS 'User and Contact Management System'
```

6.5.6 CONTINUE

Use the CONTINUE command to move to the next occurrence of the WHILE loop and run the code.



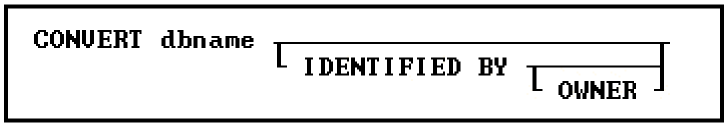
Example

In the following example, when the code is run, processing returns to line 3 after it completes the CONTINUE command on line 6. The while-block commands in line 8 are not run.

```
SET VARIABLE v1=0
SET VARIABLE V2=1
WHILE v1 = 0 THEN
    *(while-block commands)
  IF v2 <> 0 THEN
    CONTINUE
  ENDIF
    *(while-block commands)
ENDWHILE
```

6.5.7 CONVERT

The CONVERT command is used to convert 4.5 and higher databases to R:BASE 11.



Options

dbname
Specifies the database to be converted.

IDENTIFIED BY
Specifies the user identifier. If left blank, R:BASE prompts you for the user identifier. R:BASE does not display it as you enter the text.

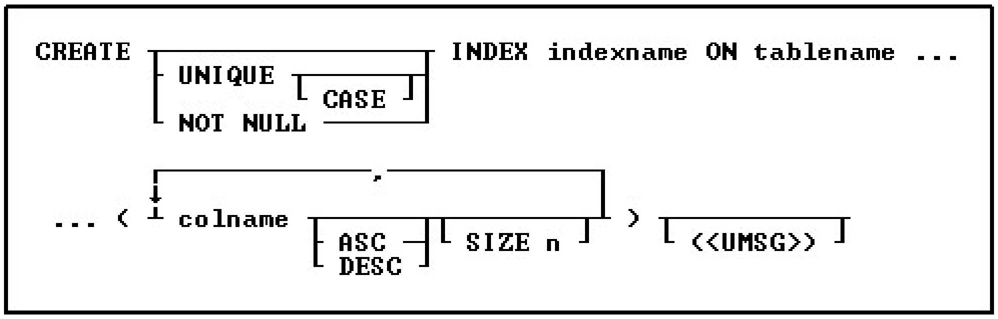
OWNER
Optional; specifies the database owner name. If omitted and an OWNER name exists, you will be prompted.

About the CONVERT Command

R:BASE requires the conversion of your existing 5.5 or lower R:BASE database. Once the database is converted, it CANNOT be accessed by any previous version of R:BASE. Be sure and backup your database before you convert it.

6.5.8 CREATE INDEX

Use the CREATE INDEX command to speed up data retrieval by creating pointers that locate rows in a table easily.



Options

,
Indicates that this part of the command is repeatable.

ASC
DESC
Specifies whether to sort a column in ascending or descending order.

CASE
Specifies that the data values will be case sensitive.

colname
Specifies a column name. The column name is limited to 128 characters. In a command, you can enter *#c*, where *#c* is the column number shown when the columns are listed with the LIST TABLES command.

INDEX indexname
Specifies an index, which is displayed with the LIST INDEX command. An *indexname* is required.

ON tblname
Specifies the table in which to create an index for a column.

SIZE n
Sets the minimum number of characters to preserve to determine uniqueness during hashing. This number can be a maximum of 196 characters. The index is created with the first *n* characters preserved and the rest of the value stored as a 4-byte hashed representation.

UNIQUE
Requires the values in a column to be unique.

(<UMSG>)
Creates a constraint violation message to appear whenever a unique index data integrity violation occurs. The message can suit the meaning of your data, such as "You must enter a unique number" for a unique index constraint violation. The index must be dropped, then recreated in order to modify the violation message.

About the CREATE INDEX Command

An index creates pointers to rows in columns, which allows R:BASE to find the rows using pointers much faster than searching the rows of data sequentially. You can index a column of any data type. An indexed column improves the performance of the following commands, clauses, and operations.

Commands, Clauses, and Operations to Use with Indexes	
DELETE DUPLICATES	RULES
ORDER BY	SELECT (when it includes a WHERE or an ORDER BY clause)
PROJECT	WHERE

Although indexes speed up processing, they might slow down data entry because building an index for each value as it is entered takes time. Creating indexes for columns that contain many duplicate values does not always speed up processing. Indexes also occupy space on a disk.

Null Values

An indexed column can contain null values, but R:BASE uses an index most efficiently if each row in the indexed column contains a value. Primary keys, unique keys, or unique indexes explicitly restrict the insertion of null values. For other indexes, you can define a rule to ensure that a column always contains a value.

UPDATE Permission

When access rights for a table have been assigned using the GRANT command, you must have UPDATE permission for the column you want to index.

Indexing Criteria

Some columns are better candidates than others for indexing. To receive the greatest benefit from indexes, use the following criteria to help you decide which type of column is the best choice for indexing your table(s):

Primary Key

R:BASE automatically indexes the column(s) that is defined as the table's primary key.

Foreign Key

R:BASE automatically indexes the column(s) that is defined as the table's foreign key.

Columns Used in Queries

Columns that are not primary or foreign keys but are frequently used in queries should be indexed. Create a unique key constraint for columns that are not primary or foreign keys, but which uniquely identify a row in the table.

Columns Frequently Using ORDER BY or GROUP BY

Include a column in an ascending-order index when the column is not a primary or foreign key but is frequently referenced in an ascending-column ORDER BY or GROUP BY clause. Similarly, include a column in a descending-order index when the column is frequently referenced in a descending-column ORDER BY clause.

Full- and Partial-Text Indexes

Text columns can make effective indexed columns. If the size of the column that has a TEXT data type is 200 bytes or less, R:BASE creates a full-text index. A full-text index is an index that stores the entire contents of a column as an index in File 3, which is the file that contains indexes to columns. If the size of the column is greater than 200 bytes, R:BASE creates a partial-text index.

If you specify the SIZE option to be less than the defined length of a column, R:BASE creates a partial-text index, and any text column that has a defined length over 200 bytes must be a partial-text index. For columns that have a TEXT data type and exceed 200 bytes, you can specify the SIZE option to be between 0 and 196 to create a partial-text index. Specifying the size allows you to base your index on a specified number of characters at the beginning of the columns and to hash the remaining characters. For example, you can index a 225-character column with a TEXT data type by specifying the SIZE option to be any number less than 197 bytes. R:BASE will create an index with the first *n* characters and the rest of the value will be stored as a four-byte hashed representation of the text.

Partial-text indexes minimize storage space. However, partial-text indexes might not be as efficient as a full-text index, for example:

```
CREATE TABLE cities (cityname TEXT(40), state TEXT(2), country +
TEXT(20))
CREATE INDEX cityindex ON cities (cityname, state)
INSERT INTO cities VALUES('Bellevue','WA','USA')
INSERT INTO cities VALUES('Belltown','PA','USA')
SELECT cityname, state from cities WHERE cityname = 'Bellevue'
```

In the above example, because the query reads data only from the index named *cityindex*, there is no need to read the actual data stored in File 2-which is the data file-so the query is done quickly. The query is an index-only retrieval and produces fast results.

If a partial-text index was used in the same query as above, the partial-text index could also only use the index named *cityindex*. Because the partial-text index only preserves the first four characters, it is impossible to return the correct answer to the query from the index. The query, as shown below, would slow processing because R:BASE must read data from the R:BASE data file.

```
CREATE INDEX cityindex ON cities (cityname SIZE 4, state)
SELECT cityname, state from cities WHERE cityname = 'Bellevue'
```

When creating text indexes, be aware of the following:

- If you omit the SIZE option and the text field in the column is greater than 200 bytes, R:BASE creates a partial text index by storing the first 32 bytes of each field and hashing the remaining bytes in each field into a four-byte numeric representation of the text. For example, if the text is 280 bytes and you do not specify a size, R:BASE stores the first 32 bytes of each field and hashes the remaining 248 bytes into a four-byte integer.
- If you specify the SIZE option to be 16 bytes for a 60-byte column with a TEXT data type, R:BASE stores the first 16 bytes of each 60-byte text field and hashes the remaining bytes in each field into a four-byte numeric representation of the text. The total length of each index entry will be 20 bytes (16 + 4).
- If you specify the SIZE option to be 30-bytes for a 250-byte column with a TEXT data type, R:BASE stores the first 30 bytes of each 250-byte field and hashes the remaining bytes in each field into a four-byte numeric representation of the text. The total length of each index entry will be 34 bytes.
- If you specify the SIZE option to be 250 bytes for a column with a TEXT data type, you have made an illegal request because the maximum value for the SIZE option is 196 bytes when the length of the text field is greater than 200 bytes. If you specified the SIZE option to be 196 bytes for a 250-byte column, R:BASE would hash the remaining 54 bytes into a four-byte numeric representation of the text.
- If you omit the SIZE option and the text field in the column is 200 bytes or less, R:BASE creates a full-text index. For example, if the text is 80 bytes and you do not specify a size, R:BASE builds a full-text index of 80 bytes.

MICRORIM_INDEXLOCK

The system variable, MICRORIM_INDEXLOCK, is available to control concurrency locks for the CREATE INDEX command.
This variable prevents CREATE INDEX from holding a permanant database lock. It locks only as necessary, allowing users access to the database. This results in longer index creation time but greater concurrency. MICRORIM_INDEXLOCK is set to any integer value.

Examples

The following command creates an index for the *custid* column in the *transmaster* table.

```
CREATE INDEX trancust ON transmaster (custid)
```

The following example creates a multi-column index for the *company*, *custaddress*, and *custstate* columns in the *customer* table.

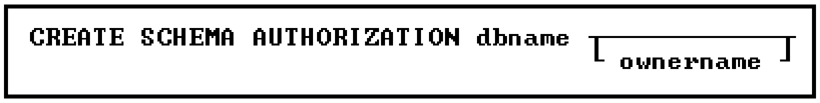
```
CREATE INDEX custaddr ON customer (company ASC, custaddress ASC, custstate ASC)
```

The following example creates a unique index for the *TRStockID* column in the *TStockHeader* table.

```
CREATE UNIQUE INDEX TRSID ON `TStockHeader` (`TRStockID` ASC ) +
('Values for rows in TurnRoundID must be unique!')
```

6.5.9 CREATE SCHEMA

Use the CREATE SCHEMA command to name a database and assign a user identifier for the database owner.



Options

AUTHORIZATION dbname
Specifies the name of the database.

ownername
Allows you to assign a unique identifier for the owner of the database.

About the CREATE SCHEMA Command
A database name is limited to 128 characters. The database name must begin with a letter, and can contain letters, numbers, and the following symbols: number or pound sign (#), dollar sign ($), underscore (_),or percent sign (%). A database name cannot contain blanks or have a file extension, and must follow the naming conventions for R:BASE and the operating system.

R:BASE creates four database files with extensions: .RX1, .RX2, .RX3, and .RX4. After you name a database, you need to use other commands to define the tables, views, rules, and access rights for the database.

Assigning A Database Owner's User Identifier
A database owner's user identifier must begin with a letter and can contain letters, numbers, and the and the following symbols: number or pound sign (#), dollar sign ($), underscore (_),or percent sign (%). If user identifiers are assigned to users, the database owner's user identifier must be unique among all user identifiers in the database. A database owner's user identifier can be a maximum of 128 characters.

If you do not specify a user identifier, R:BASE assigns the default user identifier, PUBLIC. Until a user identifier is assigned, anyone can modify the database structure, read, enter, change, or delete data. When an owner's user identifier is assigned to a database, the database is accessible only by the owner. To give other users access rights to the database, use the GRANT command.

You do not have to assign a user identifier when the database is created. To assign a user identifier after the database has been created, use the RENAME OWNER command, the **Utilities: Access Rights...** menu option in R:BASE for Windows or the **Info: Create: Access Rights: Change Owner** in R:BASE for DOS.

CREATE SCHEMA stores the owner's case folding and collating tables from the configuration file in the database. Be sure to keep a record of the owner's user identifier in a safe place away from your computer. If you lose the owner's identifier, you cannot search the database to find it.

Building a Database
You can use CREATE SCHEMA and other CREATE commands as an alternative to creating a new database using the menu options in R:BASE for Windows.

When you run the CREATE SCHEMA command, R:BASE closes the currently open database (if one exists), then defines and opens a new database.

Transaction Processing and the CREATE SCHEMA Command

If transaction processing is on when you enter a CREATE SCHEMA command, R:BASE first commits your current transaction (if any), then creates and connects you to the database. Transaction processing is on in the database, but you cannot reverse the CREATE SCHEMA command.

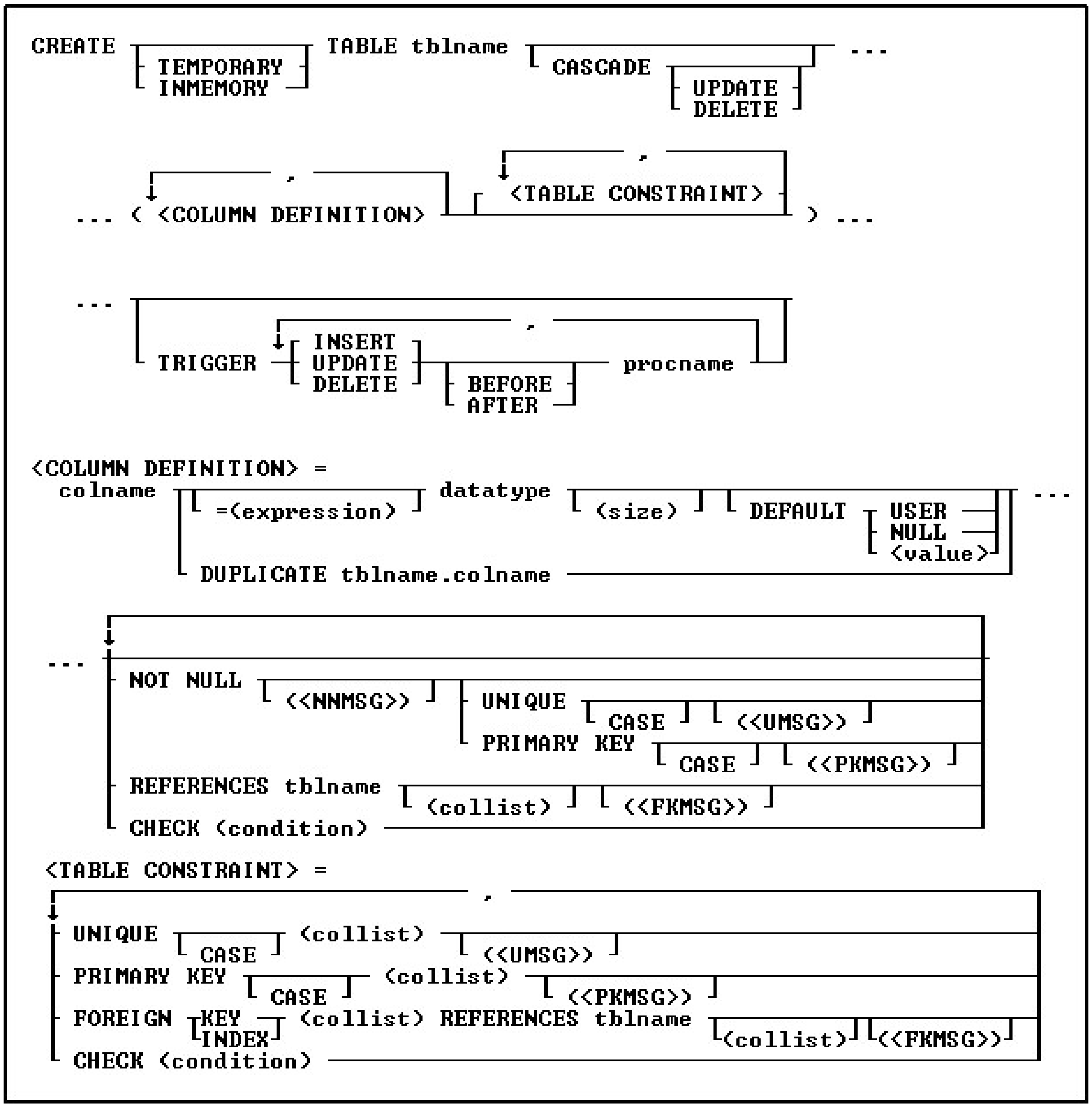
Example

The following command names the *finance* database and assigns *jane* as the database owner's user identifier.

```
CREATE SCHEMA AUTHORIZATION finance jane
```

6.5.10 CREATE TABLE

Use the CREATE TABLE command to define a new table in an existing database.



Options

,
Indicates that this part of the command is repeatable.

AFTER
Sets the trigger to activate after the INSERT, UPDATE or DELETE action.

BEFORE
Sets the trigger to activate before the INSERT, UPDATE or DELETE action. This is the default setting if the BEFORE/AFTER parameter is unused.

CASCADE
Maintains primary/foreign key relationships automatically. For example, if you either UPDATE or DELETE a primary key value from a table, the corresponding foreign key values are updated or deleted automatically. A CASCADE can be applied to UPDATE, DELETE or BOTH to specific primary keys. By not specifying either UPDATE or DELETE, both CASCADE restrictions will be enforced upon the primary/foreign key tables. Separate UPDATE and DELETE data restrictions can allow a CASCADE to be enforced for records that are updated, but not enforced when records are deleted, in order to avoid an accidental or undesired record delete. CASCADE can only be added to tables with primary keys.

CASE
Specifies that the data values will be case sensitive.

CHECK (condition)
Sets a condition to be satisfied before an update or insertion of a row can occur, which creates an R:BASE rule.

(collist)
Specifies a list of one or more column names, separated by a comma (or the current delimiter), used in the unique key specification. This option is only used when referencing a unique key.

colname
Specifies a column name. The column name is limited to 128 characters.

datatype
Specifies an R:BASE data type.

DEFAULT
Specifies a default value for the column if no value is provided by the user.

= (expression)
Determines a value using a text or arithmetic formula. The expression can include other columns from the table, constant values, functions, or system variables such as *#date*, *#time*, and *#pi*.

(<FKMSG>)
Creates a constraint violation message to appear whenever a foreign-key data integrity violation occurs. The message can suit the meaning of your data, such as "You must enter a valid number" for a foreign-key constraint violation. You can define two messages: one for inserting and one for updating. A constraint must be dropped, then recreated in order to modify the violation message.

FOREIGN INDEX
With the FASTFK setting on, creates a foreign key that has an index using row pointers for data retrieval on selected columns.

FOREIGN KEY
Specifies a column or set of columns required to match values in a particular primary key or unique key defined in a table.

(<NNMSG>)
Creates a constraint violation message to appear whenever a not-null data integrity violation occurs. The message can suit the meaning of your data, such as "You must enter a valid number" for a not-null constraint violation. A constraint must be dropped, then recreated in order to modify the violation message.

NOT NULL
Prevents a column from accepting null values, but permits it to accept duplicate values.

If this option is specified without a setting for a default value, you cannot insert rows without specifying values for the given column.

(<PKMSG>)
Creates a constraint violation message to appear whenever a primary-key data integrity violation occurs. The message can suit the meaning of your data, such as "You must enter a valid number" for a primary-key constraint violation. You can define three messages: one for uniqueness, one for deleting, and one for updating. A constraint must be dropped, then recreated in order to modify the violation message.

PRIMARY KEY
Specifies the column(s) to designate as a primary key.

procname
The procedure name. If a procedure by this name already exists in the database, an error is generated.

REFERENCES tablename
Identifies the primary key or unique key table to which the foreign key refers.

(size)
Defines the length of a column of the TEXT data type (if not the default 8). Defines the precision and scale of a column of the DECIMAL or NUMERIC data type, if not the default of precision 9 and scale 0 (9,0). VARBIT, VARCHAR, and BIT either require or can have a size.

tblname
Specifies a table name. The table name is limited to 128 characters.

TEMPORARY
Creates a temporary table that disappears when the database is disconnected.

TRIGGER
Runs a Stored Procedure when an UPDATE, DELETE, or INSERT command is run on the table.

(<UMSG>)
Creates a constraint violation message to appear whenever a unique-key data integrity violation occurs. The message can suit the meaning of your data, such as "You must enter a valid number" for a unique-key constraint violation. You can define three messages: one for uniqueness, one for deleting, and one for updating. A constraint must be dropped, then recreated in order to modify the violation message.

UNIQUE
Requires the values in a column to be unique by defining a unique key constraint.

USER
NULL
(value)
Default USER: Specifies the default value to be the user identifier.
Default NULL: Specifies the default value to be null.
Default (*value*): Specifies the default to be the indicated value.

About the CREATE TABLE Command

To define a table, you need to specify column definitions. Table and column names must begin with an upper- or lowercase letter. Names can contain letters, numbers, and the following special characters: #, $, _, and %. R:BASE verifies that a table or column name is unique by reading all characters.

When you define a table, you can also add table constraints. However, you cannot add a foreign key to a temporary table.

To define more than one column in a command, use commas (or the current delimiter character) to separate the column definitions.

Computed Columns

A computed column is a column containing a value that R:BASE calculates from an expression defined for the column. In the expression, you can use other columns from the table, constant values, functions, and the system variables *#date*, *#time*, and *#pi*. Global variables are not allowed in an expression.

You must assign a data type that is compatible with the result of the computation. The columns used for calculating the computed column must precede the computed column in the table.

Indexing Columns

If you want to assign an index to a column, use the CREATE INDEX command.

Database Access Rights with CREATE TABLE

CREATE TABLE requires either the CREATE access right or the owner's user identifier when access rights have been assigned with the GRANT command.

Examples

The command below defines a table named *employee* with the following columns and data types: *empid* (INTEGER), *emptitle*(TEXT 30), *empfname* (TEXT 10), *emplname* (TEXT 16), *empaddress* (TEXT 30), *empcity* (TEXT 20), *empstate* (TEXT 2), *empzip* (TEXT 10), *empphone* (TEXT 12), *empext*(INTEGER), *hiredate* (DATE), and *entrydate* (DATE). In addition, the NOT NULL option specifies that the columns *empfname*, *emplname*, and *hiredate* must contain a value. The NOT NULL UNIQUE option specifies that the *empid* and *empext* columns must contain unique values.

```
CREATE TABLE employee (empid INTEGER NOT NULL UNIQUE, +
emptitle TEXT (30), empfname TEXT (10) NOT NULL, emplname TEXT +
(16) NOT NULL, empaddress TEXT (30), empcity TEXT (20), empstate +
TEXT (2), empzip TEXT (10), empphone TEXT (12), empext INTEGER +
NOT NULL UNIQUE, hiredate DATE NOT NULL, entrydate DATE)
```

The following command creates a table using the column constraint CHECK on the *empid* column.

```
CREATE TABLE employee (empid INTEGER CHECK (empid > 0), +
empname TEXT (40), empage INTEGER)
```

The example below creates a table using the column constraint CHECK on the *empid* and *empage* columns.

```
CREATE TABLE employee (empid INTEGER CHECK (empid > 0), +
empname TEXT (40), empage INTEGER CHECK (empage >0 and empage < 100))
```

Table constraints are defined if it is necessary to reference multiple columns within the same expression. The UNIQUE (*collist*) option is entered at the end of the following command so that the values in the *empid*, *empfname*, *emplname* columns are unique as a group in a row. Because this option follows a column definition, precede the option with a comma.

```
CREATE TABLE employee (empid INTEGER NOT NULL, +
emptitle TEXT (30), empfname TEXT (10) NOT NULL, emplname TEXT +
(16) NOT NULL, empaddress TEXT (30), empcity TEXT (20), empstate +
TEXT (2), empzip TEXT (10), empphone TEXT (12), empext INTEGER +
NOT NULL UNIQUE, hiredate DATE NOT NULL, entrydate DATE, +
UNIQUE (empid, empfname, emplname))
```

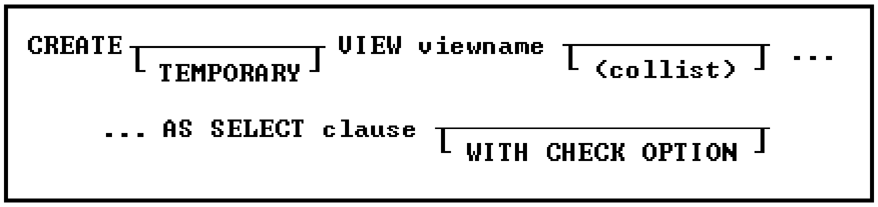
The following command creates a table that would contain an employee's total years of employment. The command places a column constraint on the *empid* and *empage* columns, and a table constraint on the *yrshere* and *yrsanywhere* columns. The value entered for *yrshere* must be less than or equal to the value entered for *yrsanywhere*.

```
CREATE TABLE employee (empid INTEGER CHECK (empid > 0), +
empname TEXT (40), empage INTEGER CHECK +
(empage > 0 and empage < 100), yrshere INTEGER, +
yrsanywhere INTEGER, CHECK (yrshere <= yrsanywhere))
```

Copyright © 1982-2025 R:BASE Technologies, Inc.

6.5.11 CREATE VIEW

Use the CREATE VIEW command to define a view that combines columns from existing tables or views.



Options

AS SELECT clause
Specifies the columns and rows to include in the view. As a rule anything that is acceptable in an ordinary select clause will work here. If you are planning on using expressions you should considering using the SELECT AS notation to give each column meaningful names.

(collist)
Specifies a list of one or more column names or aliases, separated by a comma (or the current delimiter). These names will be the column headings displayed in the result of a SELECT command or the Data Browser.

TEMPORARY
Creates a temporary view that disappears when the database is disconnected.

viewname
Specifies a view name.

WITH CHECK OPTION
Specifies that a row cannot be added or updated unless it meets the conditions included in the WHERE clause (which is part of the SELECT clause). R:BASE uses this option only on a view that can be updated.

About the CREATE VIEW Command

CREATE VIEW defines a view to store in the *sys_views* table. You can use a stored view whenever necessary. Unlike a table, stored views contain no data. R:BASE collects data for the view from the source tables or views when a command calling the view is run.

A view is the most efficient way to gather data from separate tables or views into one location. A view that can be updated allows you to enter, change, and delete data from the source table. The number of tables in a view is dependent on available memory.

You can define a view containing a maximum of 400 columns from as many tables or views as memory allows. However, a view is still limited to the 32,786 character row-size limit. You can use the SQL symbol asterisk (*) to include all columns from all tables or views, or you can specify the columns you want to include. You can combine these two methods to include all columns from one table or view and specify columns from another table or view. You must separate column, table, and view names with commas (or the current delimiter character).

Avoiding Multiple Occurrences of Columns

When you use only an asterisk (*) in the SELECT clause, the view will contain all columns from all tables or views. If the tables or views contain common columns, the view will contain multiple occurrences of those columns.

To avoid multiple occurrences of common columns, specify which columns to include in the view. For example, to include all columns from one table but only certain columns from another table, use an * for the first table, then list the column names to be included from the second table. You can specify the columns for a view as *t1*.*, *t2.col2*, *t2.col3*, where *t1*.* specifies all columns from table *t1*and *t2.col2*, *t2.col3* specifies two columns from table *t2*. Be sure that the list does not include the common columns contained in the second table. When you use a combination of * and column names, you must specify the table with which * is associated. However, you can omit the table or correlation name for the columns listed individually if those columns occur in only one table in the view.

Duplicate Rows

If the tables forming a view contain duplicate rows, either individually or in combination with other tables in the view, multiple duplicate rows will be displayed. Usually, the presence of duplicate rows in a view indicates a database design problem. Check your database structure for design flaws such as redundant data storage.

Linking Columns

When you build a view from two or more tables or views, define the relationship between the source tables and views by identifying linking columns in a WHERE clause. Linking columns are columns that contain the same values; their names can be the same or different. For example, the following WHERE clause specifies that a view displays only those rows where the values in *t1.col1* are equal to the values in the common column *t2.col1*.

WHERE t1.col1 = t2.col1

Updating Views

You can update the data for columns in a view when the view does not contain a UNION operator, and its SELECT clause meets the following requirements:

- The clause does not specify DISTINCT.
- The clause does not include a sub-SELECT command in the WHERE clause.
- The clause does not include a GROUP BY or HAVING clause.
- The clause does not include an ORDER BY clause.

When you add, change, or delete rows by updating a single table view, you also modify the data in the source table. In multi-table views you cannot add, edit, or delete rows. Any additions or changes to data made through a view are subject to all the user-defined rules specified for the table when it was constructed. In addition, if you specify the WITH CHECK OPTION for the view, you can only add or modify rows that meet the conditions defined in the WHERE clause.

You can only use the DELETE, INSERT, LOAD, and UPDATE commands with views that can be updated. If a view cannot be updated, you can use the view only to display data or as the basis for reports.

Views Compared with Look-up Tables

If the data used in a report is stored in more than one table, using a view is more efficient than a driving table and several look-up tables because it takes less time to print the report using a view. Using a view is more efficient because R:BASE gathers the data for a view before, rather than during, printing.

Database Access Rights with CREATE VIEW

The access rights that can be assigned with the GRANT command depend on whether or not the view can be updated. The ALL PRIVILEGES, DELETE, INSERT, SELECT, and UPDATE access rights can be granted on a view that can be updated. You must have ALL PRIVILEGES or SELECT access rights on a table or view to include it in a view.

If you have been assigned the SELECT access right and the WITH GRANT OPTION has been assigned on all the source tables or views used in a view, you can grant both SELECT and the WITH GRANT OPTION to other users.

If you are the database owner or you have the WITH GRANT OPTION on a view, you can also assign access rights on stored views.

Changing Views

You cannot change a view at the R> Prompt. To change a view from the R> Prompt, you must delete the view by using the DROP command, then define a new view. However, you can use the **View Designer** option from the **Tools** menu in R:BASE for Windows or **Views: Create/modify: Manage views** in R:BASE for DOS, to change a view that meets the following requirements:

- The view does not include a GROUP BY or HAVING clause.
- The view does not include a sub-SELECT command in the WHERE clause.

Examples

The following command defines a view that can be updated and specifies a subset of columns "*custid*, *company*, *custaddress*, *custcity*, *custstate*, and *custzip*" from one table, *customer*. The column list must match the number of columns in the SELECT clause; the names, however, can be different. The WHERE clause restricts the rows to those with zip codes ranging from 40001 through 49999. The WITH CHECK OPTION specifies that only rows that meet the condition included in the WHERE clause can be added or changed in the database.

```
CREATE VIEW cust_addr (custid, custcompany, custaddress, custcity, custstate, +
custzip) AS SELECT custid, company, custaddress, +
custcity, custstate, custzip FROM customer WHERE custzip +
BETWEEN 40001 AND 49999 WITH CHECK OPTION
```

The following command defines a view that will display only those rows from the *customer* and *transmaster* tables that have matching values in the common column *custid*. Therefore, only the rows that have customers who have had a transaction will be included in the view. The command line ORDER BY *custid* tells R:BASE to sort the rows and display them by the customer identification number.

```
CREATE VIEW cust_trans AS SELECT t1.custid, company, netamount +
FROM customer t1, transmaster t2 WHERE t1.custid = t2.custid +
ORDER BY custid
```

The following command defines a view that will display only those rows from the *customer* table where the values in the *custid*column do not exist in the *transmaster* table. Therefore, only the rows that have customers who have not had a transaction will be included in the view.

```
CREATE VIEW cust_notrans AS SELECT custid, company FROM +
customer WHERE custid NOT IN (SELECT custid FROM transmaster) +
ORDER BY custid
```

The following command combines the commands in the two preceding examples, creating a view that will display all rows from both the *customer* and the *transmaster* tables. The UNION operator joins the two SELECT clauses, allowing you to display rows for all customers whether or not they have had a transaction.

The first SELECT clause instructs R:BASE to include the rows from both tables where the values in *custid* match. The second SELECT clause instructs R:BASE to include rows from the *customer* table where there are no values for *custid* in the *transmaster* table.

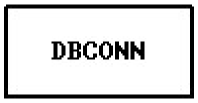
When you use the UNION operator, the number of columns specified in the SELECT clauses must be the same and the data types of the columns must be compatible. If there is no column in one table that matches a column listed in the other table's SELECT clause, you must substitute a value (or null value). Because the *netamount* column does not exist in the *customer* table used in the second SELECT statement, the value $0.00 was entered in place of *netamount*.

```
CREATE VIEW all_cust_trans AS SELECT t1.custid, company, netamount +
FROM customer t1, transmaster t2 WHERE t1.custid = t2.custid +
UNION SELECT custid, company, $0.00 FROM customer +
WHERE custid NOT IN (SELECT custid FROM transmaster) +
ORDER BY custid
```

6.6 D

6.6.1 DBCONN

Use the DBCONN command to view all applications with connections to the currently connected database.



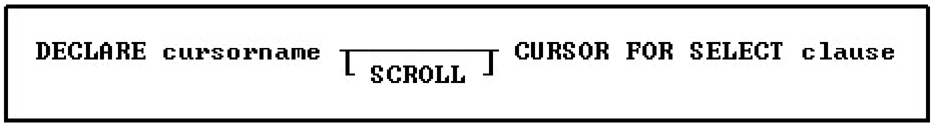
The list of applications may include R:BASE, Runtime for R:BASE, compiled applications, programs that have made ODBC connections to the databases, etc.

The output shows the application name, path, and file name.

```
R> DBCONN

Current connected database applications:
R:BASE 11 (C:\RBTI\RBG11\RBG11.exe)
```

6.6.2 DECLARE CURSOR

Use the DECLARE CURSOR command to create a cursor that points to a row in a table or view.



Options

cursorname
Specifies a 1 to 18 character cursor name.

CURSOR FOR SELECT clause
Specifies the columns and rows from the table whose values you want to use. You may include the DISTINCT modifier as well as WHERE clauses and ORDER BY clauses.

SCROLL
Defines a cursor that moves forwards and backwards through a table. If this option is omitted, the cursor can only move forward.

About the DECLARE CURSOR Command

In the SELECT clause, specify the columns that contain the values you want to use from the row. Specifying the columns makes the column values accessible to the FETCH and SET VARIABLE commands. Once a cursor is declared, use the OPEN command to initialize the cursor and position it before the first row specified by DECLARE CURSOR.

Use DECLARE CURSOR to define a path through a table or view. You can move through the defined rows using the FETCH command by using either multiple FETCH commands or embedded FETCH commands within a WHILE loop. You only need to point to specific columns with DECLARE CURSOR, then FETCH can retrieve those columns by placing their values into variables. You can define a scrollable cursor, which is a cursor that moves backwards and forwards through a table.

DECLARE CURSOR defines a temporary view in memory; R:BASE does not store the view definition in the *sys_views* table. The SELECT clause defines columns, tables, rows, sort order, and potential grouping for the rows. When DECLARE CURSOR executes, it validates the syntax and names of columns and tables. The OPEN command can evaluate variables, create a copy of the cursor based on those values, then position the cursor before the first row.

Listing Cursors

Use LIST CURSOR to list all currently defined cursors and their status, open or closed.

Using Cursor Names in Commands

You can use the cursor name instead of a table name in commands. The following table provides examples of using the cursor name instead of a table name in commands.

To do this...	**Use the cursor name like this...**
Set a variable to a column value	FETCH *cursorname* INTO *varlist* SET VARIABLE *varname* = *colname* WHERE CURRENT OF *cursorname*
Change a column value to a constant	UPDATE *tblname* SET *colname* + = *value* WHERE CURRENT OF *cursorname*
Change a column value to a variable value	UPDATE *tblname* SET *colname* + = .*varname* WHERE CURRENT OF *cursorname*
Change a column value to an expression	UPDATE *tblname* SET *colname* + = (expression) WHERE CURRENT OF *cursorname*
Delete the pointed-to row	DELETE FROM *tblname* + WHERE CURRENT OF *cursorname*

Modifying Data Using a Cursor

If you use a cursor in commands that modify data (the UPDATE and DELETE commands), only the current row is modified. To modify all referenced rows, include FETCH in a WHILE loop to move the cursor through the rows.

Closing Cursors

The following commands close cursors.

Command Name	**Description**
CLOSE	Closes the open cursor but does not remove the cursor definition. However using CLOSE frees most of the memory used when a cursor is opened. CLOSE also frees any file handles used by DECLARE CURSOR.
DROP CURSOR	Entirely removes the cursor definition. Dropping a cursor definition frees all memory used by the definition.

Examples

The following example uses the SCROLL option with DECLARE CURSOR.

```
DECLARE c1 SCROLL CURSOR FOR SELECT empid, transid, transdate, custid, netamount FROM
transmaster
```

Checking End-of-Data Conditions Using sqlcode

The following example uses *sqlcode* to check end-of-data conditions, which is the recommended program structure for DECLARE CURSOR. The sqlcode system variable holds values only for specific types of status.

Type of Error	**SQLCODE**
Data found	0
Data not found	100

In the following example, the WHILE statement checks the value of *sqlcode*.

```
1) DECLARE cursor1 CURSOR FOR SELECT custid, netamount +
     FROM transmaster ORDER BY netamount
2) OPEN cursor1
3) FETCH cursor1 INTO vcustid INDICATOR vi1, vnetamt +
     INDICATOR vi2
```

```
4) WHILE sqlcode <>100 THEN
     SHOW VARIABLE vcustid
     SHOW VARIABLE vnetamt
5)   FETCH cursor1 INTO vcustid INDICATOR vi1, vnetamt +
       INDICATOR vi2
   ENDWHILE
6) DROP CURSOR cursor1
```

1. DECLARE CURSOR defines the cursor path.
2. OPEN opens the cursor, evaluates variables, and positions the cursor before the first row.
3. The first FETCH command retrieves the first set of values. The indicator variables *vi1* and *vi2* capture the status values, -1 for null and 0 for a value. If you omit indicator variables in FETCH commands, R:BASE displays a message if it encounters a null value, but continues processing rows.
4. The WHILE loop processes the rows until there are no more rows. At that point, sqlcode is set to 100, and the WHILE loop ends. Control passes to the command after ENDWHILE. If the first FETCH retrieved no data, the WHILE loop is not entered.
5. FETCH retrieves all succeeding rows and sets *sqlcode*each time. When it does not find any more data, *sqlcode* is set to 100 and the WHILE loop ends.
6. DROP CURSOR removes the cursor definition from memory.

Using the WHENEVER Command with DECLARE CURSOR

The following example shows the use of the WHENEVER command, which checks the value of *sqlcode*. A single WHENEVER command can start a status-checking cycle that remains in operation until a command or procedure file finishes running. As in the first two examples, an indicator variable is included with each variable in FETCH. Without the indicator variables, R:BASE displays a message if it encounters a null value, but continues processing rows.

```
1) WHENEVER NOT FOUND GOTO skiploop
2) DECLARE cursor1 CURSOR FOR SELECT custid, netamount +
     FROM transmaster ORDER BY netamount
3) OPEN cursor1
4) FETCH cursor1 INTO vcustid INDICATOR vi1, vnetamt +
     INDICATOR vi2
5) WHILE #DATE IS NOT NULL THEN
     SHOW VARIABLE vcustid
     SHOW VARIABLE vnetamt
     FETCH cursor1 INTO vcustid INDICATOR vi1, vnetamt +
     INDICATOR vi2
   ENDWHILE
6) LABEL skiploop
7) DROP CURSOR cursor1
```

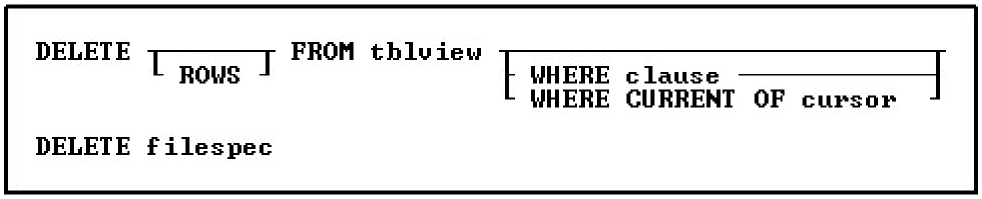
1. WHENEVER NOT FOUND tells R:BASE to execute GOTO if a command that searches for data, such as FETCH, cannot find more rows. If the first FETCH command does not find any rows, control passes to the command following LABEL *skiploop*. WHENEVER automatically checks any command that searches for data. If a data-not-found condition occurs, control passes to the command following the specified label.
2. DECLARE CURSOR defines the cursor path.
3. OPEN opens the cursor, evaluates the variables, and positions the cursor before the first row.
4. The first FETCH command retrieves the first set of values. If no rows match, control passes to LABEL *skiploop*. Indicator variables *vi1* and *vi2* capture the status values (-1 for null and 0 for a value). If you omit indicator variables in FETCH commands, R:BASE displays a message if it encounters a null value, but continues processing rows. (WHENEVER instructs R:BASE to exit the WHILE loop only when *sqlcode* is 100.)
5. The WHILE loop processes rows until WHENEVER stops execution.
6. This label defines where to pass control if a data-not-found condition occurs before the WHILE loop begins executing. WHENEVER includes this label name.
7. DROP CURSOR removes the cursor definition from memory.

Visit the From The Edge Web site to download the "R:BASE Cursors Explained" technical document.

Copyright © 1982-2025 R:BASE Technologies, Inc.

6.6.3 DELETE

Use the DELETE command to remove selected rows from a table.



```
DELETE [ROWS] FROM tblview [WHERE clause / WHERE CURRENT OF cursor]

DELETE filespec
```

Options

FROM tblview
Specifies the table or view.

ROWS
This word is optional.

WHERE clause
Limits rows of data. For more information, see WHERE.

WHERE CURRENT OF cursor
Specifies a cursor pointing to the row the DELETE command will remove. This option can replace a standard WHERE clause.

Use the DECLARE CURSOR command to define the cursor.

filespec
Specifies the file to be deleted. Optionally, include a drive and path specification in the form D:\PATHNAME\FILENAME.EXT.

On a workstation with multiple drives (local or mapped), especially when the files are on the different drive, it is always the best practice to define a drive letter when copying, deleting, renaming or running files, unless the specified files are located in the working directory. You will not need to specify the drive letter if all of the files are located in the default directory when using the copy, delete, rename or run commands.

About the DELETE Command

DELETE removes rows from a table or view. Without a WHERE or WHERE CURRENT OF clause, R:BASE deletes all rows from the specified table or view. R:BASE displays a confirmation message before deleting the rows. R:BASE does not display a confirmation message when you execute a DELETE command from a command file. Views must be updatable to delete rows from it; for more information about updatable views, see CREATE VIEW.

Before you use a WHERE clause with the DELETE command, test the clause by using it with a SELECT command, which allows you to view the rows before deleting them.

The WHERE CURRENT OF clause specifies a cursor pointing to a row that the DELETE command will remove. Once you define a cursor with DECLARE CURSOR and open a route with the OPEN command, you can use the cursor in a WHERE CURRENT OF clause to delete only the current row. Use the FETCH command to move the cursor to the next available row.

You must restore deleted rows from a backed up database or table. To recover disk space after deleting rows, use the PACK or RELOAD commands.

DELETE removes rows from a table or single-table view. If you have set transaction processing on, you can restore rows with ROLLBACK. If not, you must restore them from a backup database or table. If you prefer not to use transaction processing, you can first use a relational command, such as PROJECT, to make a backup copy of the table from which you are deleting rows. Then you can delete rows from the original table and remove the backup copy later.

Examples

The following command deletes all rows from the *transmaster*table. When you omit a WHERE clause, be sure that you want to delete all rows from the table.

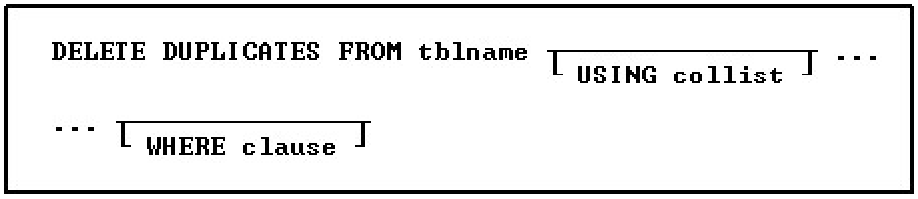
```
DELETE FROM transmaster
```

The following command deletes rows from the *transmaster* table where the *custid* value is 100.

```
DELETE FROM transmaster WHERE custid = 100
```

6.6.4 DELETE DUPLICATES

Use the DELETE DUPLICATES command to remove duplicate rows from a table.



Options

FROM tblname
Specifies the table name.

USING collist
Deletes rows based on duplicate values in the specified list of columns.

WHERE clause
Limits the rows of data to be deleted. For more information, see WHERE.

About the DELETE DUPLICATES Command

Use DELETE DUPLICATES to delete duplicate rows from a table. A duplicate row is a row where the values for each column are exactly the same as those in another row in the table. This command deletes all but the first row for each set of duplicate rows.

DELETE DUPLICATES processes faster when the table contains an indexed column and the USING *collist* option is used.

Rules for Column Deletion
You can specify which rows to delete in a list of columns. The following rules apply:

- The first row is retained in the table.
- Any row with duplicate values in a specified column list is deleted, regardless of the values in any of its other columns.

Case Sensitivity
DELETE DUPLICATES is case sensitive when CASE is set on. For example, if CASE is set on and one row included the name SMITH and another row included the name Smith, R:BASE would not delete either row. However, if CASE was set off, R:BASE would delete the second row. (The default setting for CASE is off.)

NULL Values
When NULL values exist in the table, the EQNULL setting must be set to ON to ensure duplicates are removed.

You must restore deleted rows from a backed up database or table. To recover the data's disk space after rows are deleted, use PACK or RELOAD.

If you have set transaction processing on, you can restore rows with ROLLBACK. If not, you must restore them from a backup database or table. If you prefer not to use transaction processing, you can first use a relational command, such as PROJECT to make a backup copy of the table from which you are deleting rows. Then you can delete rows from the original table and remove the backup copy later.

Example

The following command deletes duplicate rows from the *transmaster* table, but retains the first of the duplicate rows.

```
DELETE DUPLICATES FROM transmaster
```

The following example deletes duplicate rows based on the *transid*, *empid*, and *custid* columns in the *transmaster* table. Only the designated columns will be used to determine whether the rows are duplicates.

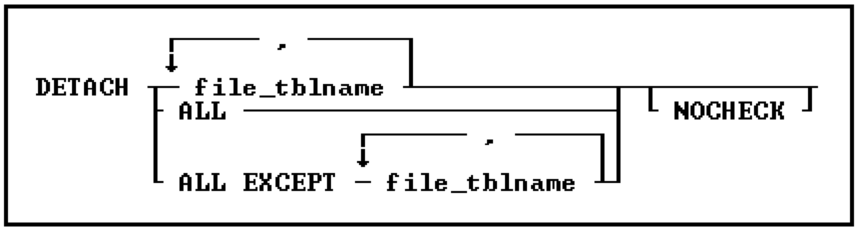
```
DELETE DUPLICATES FROM transmaster USING transid, empid, custid
```

The following deletes the duplicate rows based on the *transid and empid* columns in the *transmaster* table, where transaction dates are greater than January 1, 2020.

```
DELETE DUPLICATES FROM transmaster +
  USING transid, empid +
  WHERE transdate in (
  SELECT transdate FROM transmaster +
  WHERE transdate > 01/01/2020)
```

6.6.5 DETACH

Use the DETACH command to remove a dBASE file table and its associated dBASE index files from the open R:BASE database.



Options

,
Indicates that this part of the command is repeatable.

ALL
Removes all dBASE tables and associated dBASE index files from the open R:BASE database.

ALL EXCEPT file_tblname
Removes all dBASE tables and associated dBASE index files from the open R:BASE database, except the specified table.

file_tblname
Removes the specified dBASE table and associated dBASE index files from the open R:BASE database.

NOCHECK
Eliminates the confirmation message.

About the DETACH Command

Before you can detach a dBASE file table, an R:BASE database must be open. When you detach a dBASE file table, do not include the .DBF extension. DETACH requires the database owner's user identifier if one has been defined.

If you remove a dBASE file that is used in a form, report, label, or application, make the necessary changes to reflect detaching the dBASE file from the database. The rules, descriptions, and any access rights are deleted when you detach the dBASE file.

Use ATTACH to reattach a detached file to the same or a different database.

When STATICDB is set on-which actives a read-only schema mode-DETACH is unavailable.

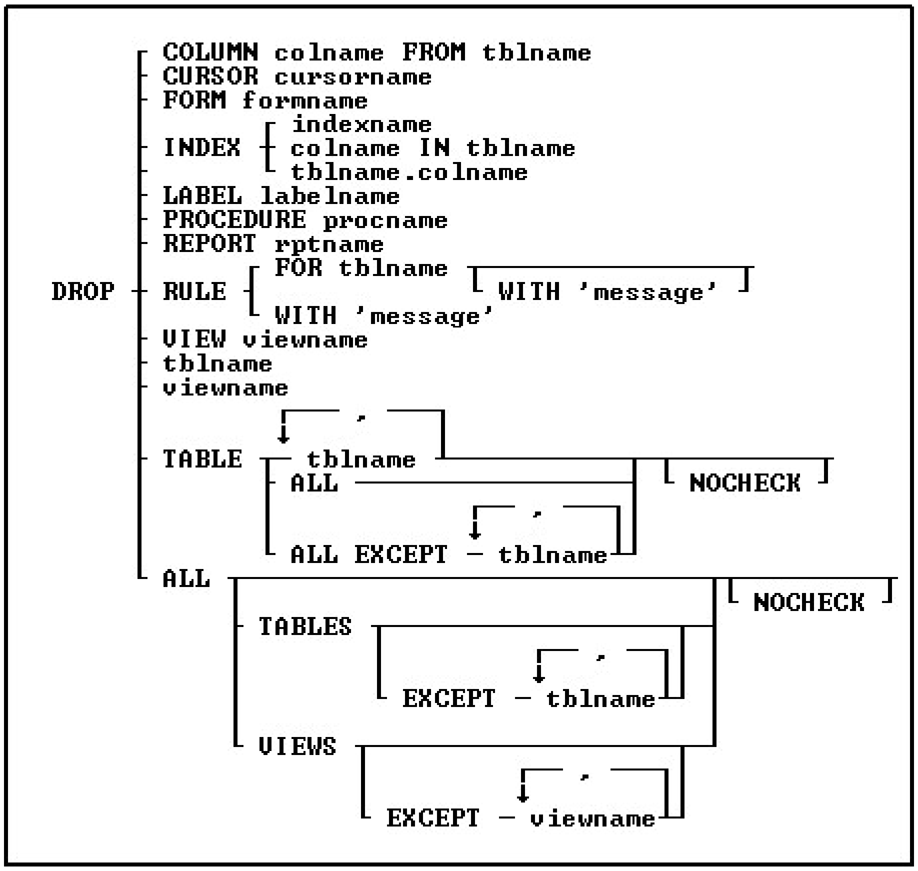
Example

In the following example, the first command line opens the *concomp* database. The second command line detaches the dBASE file table *sampgate* from the *concomp* database.

```
CONNECT concomp
DETACH sampgate
```

6.6.6 DROP

Use the DROP command to remove the specified item from the database.



Options

,
Indicates that this part of the command is repeatable.

ALL

Removes all tables from the database.

ALL EXCEPT tblname
Removes all tables from the database except the table(s) listed.

ALL EXCEPT viewname
Removes all views from the database except the view(s) listed.

colname IN tblname
Removes the index from the specified column in the specified table.

COLUMN colname FROM tblname
Removes a column's structure and data from the specified table.

CURSOR cursorname
Removes the specified cursor. The DROP CURSOR command removes a cursor definition from memory, therefore freeing memory.

Cursors are dropped when you use the CONNECT command to open another database, or the DISCONNECT command to close the database.

FOR tblname
Removes the rule from the specified table for the specified table.

FORM formname
Removes the specified form.

INDEX
Removes the index from the column in the specified table.

indexname
Removes a named index.

LABEL labelname
Removes the specified label.

NOCHECK
Eliminates the confirmation message.

PROCEDURE procname
Removes the specified Stored Procedure from the database.

REPORT rptname
Removes the specified report.

RULE
Removes the specified rule.

TABLE
Removes the specified table.

tblname
Specifies the table name to be removed.

tblname.colname
Removes the index from the specified column in the specified table.

VIEW viewname
Removes the specified view.

viewname
Specifies the view name to be removed.

WITH 'message'

Removes the specified rule from all tables to which it applies. Omitting the WITH *'message'* option removes all rules.

About the DROP Command

The table below lists guidelines for using the DROP command.

When you drop...	You must...
A column used in the definition of a computed column	Remove the computed column first.
A column used in a form, report, or label	Delete the column from the form, report, or label definition.
A column used in a rule	Delete the rule for that column in any table from which you removed the column.
A form, report, label, table, or view used in an application	Revise your application to reflect the changes in the database after dropping the form, report, or label.
A view or table used in a form report or label	Delete the form, report, or label from the database after dropping the form, report, or label.
A column or table used in a view	Delete the view from the database.
A table that has rules	Delete the rule with DROP if the table you are dropping is used in the WHERE clause of a rule definition. For example, a table name is used in a WHERE clause of a rule definition when a rule is used to verify a value in one table against values in another table.

R:BASE deletes the rules if the table is the table on which the rules are based.

After running the DROP command, the database item is gone, but the disk space the item occupied is not available. To recover that space, use the PACK or RELOAD commands.

You can rebuild a dropped index using CREATE INDEX.

When STATICDB is set on-which actives a read-only schema mode-DROP is unavailable.

Database Access Rights with DROP

The DROP command requires that you enter the database owner's user identifier if a user identifier has been assigned with the GRANT command. However, if a user has CREATE or ALTER access rights, that user can use the DROP command to drop tables or any columns in tables to which the user has rights.

Removing Rules with DROP RULE

Before you remove a rule with the DROP RULE command, enter a SELECT command to verify that you would be removing the correct rule from the correct table(s). Use the conditions in a WHERE clause to enter the exact message and any table names that you plan to use in the DROP command. Once you have verified that the message would remove the correct rules, proceed with the DROP RULES command. For example, to verify that a DROP command with the message "Model number must be unique" would remove only the rules you want to remove, enter the following SELECT command. R:BASE would display all the rules for all the tables in the database to which this message applies.

```
SELECT * FROM SYS_RULES WHERE SYS_MESSAGE = 'Model number must be unique'
```

Examples

The following command removes the *empext* column from the *employee* table.

```
DROP COLUMN empext FROM employee
```

The following two command lines show alternative ways to remove the index from the *custid* column in the *transmaster* table.

```
DROP INDEX custid IN transmaster
DROP INDEX transmaster.custid
```

The following command removes from the database all rules with the message 'Model number must be unique.'

```
DROP RULE WITH 'Model number must be unique'
```

The following command removes any rule from the *product* table that starts with the message 'Model number.' You can use the wildcard character for MANY (%) in a message.

```
DROP RULE FOR product WITH 'Model number%'
```

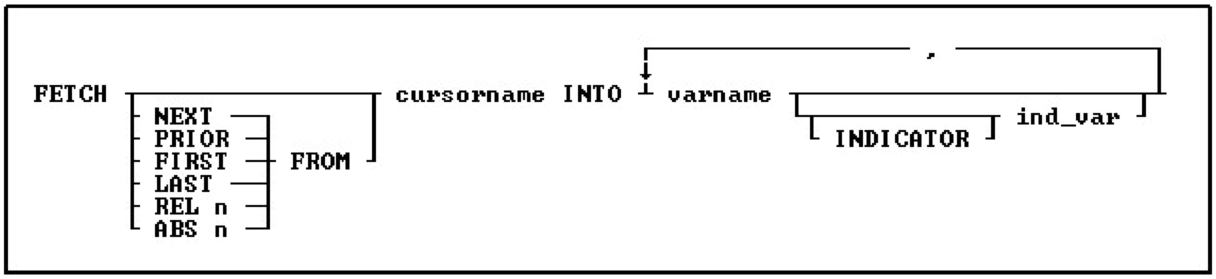
The following command removes the cursor named *cursor1* from memory.

```
DROP CURSOR cursor1
```

6.7 F

6.7.1 FETCH

Use the FETCH command to position the cursor on a row specified by the DECLARE CURSOR command, and place values from the columns into global variables.



Options

ABS n
The value *n* is the *n*th row in the cursor list. The current cursor location is not relevant. Positive numbers count from the first row in the list. Negative numbers force an end-of-data condition. This option applies only to scrolling cursors, which are defined with the DECLARE CURSOR command.

cursorname INTO
Names the cursor from which to fetch data into the specified variable list.

FIRST
Specifies the first row in the cursor list. This option applies only to scrolling cursors, which are defined with the DECLARE CURSOR command.

INDICATOR ind_var
Stores the status of the variable: non-null (0) or null (-1).

LAST
Specifies the last row in the cursor list. This option applies only to scrolling cursors, which are defined with the DECLARE CURSOR command.

NEXT
Specifies the next entry the cursor points to. This option applies only to scrolling cursors, which are defined with the DECLARE CURSOR command.

PRIOR
Specifies the prior entry the cursor points to. This option applies only to scrolling cursors, which are defined with the DECLARE CURSOR command.

REL n
Moves the cursor *n* rows. Positive integers move forward, and negative integers move backwards. For example, if *n* is 5, the cursor moves forward 5 rows. This option applies only to scrolling cursors, which are defined with the DECLARE CURSOR command.

varname
Specifies a variable name, which must be unique among the variable names within the database. The variable name is limited to 128 characters.

About the FETCH Command

FETCH moves the cursor to the next available row referred to by the DECLARE CURSOR command and also accommodates scrollable cursors specified by DECLARE CURSOR. FETCH retrieves the values of columns in the order in which the columns were specified by DECLARE CURSOR. The LIST CURSORS command lists all the defined cursors.

FETCH *cursorname* without any variable specification will retrieve the next row from the cursor. Use the SET VAR *varname* WHERE CURRENT OF *cursorname* to retrieve the columns you need.

Using the Sqlcode Variable

You must check the *sqlcode* variable with each use of FETCH to verify that all rows specified by DECLARE CURSOR have been found. Use the WHENEVER sqlcode command to check for SQL processing errors other than data-not-found errors.

Use WHENEVER NOT FOUND to check for a data-not-found errors. When you use the WHENEVER NOT FOUND command, data-not-found error checking is automatic; however, you must use the LABEL command. When a data-not-found error occurs, control passes to the command line specified by the LABEL command and the subsequent error-handling commands.

Using Indicator Variables

If the data contains null values, use indicator variables to capture the status of a value. If you do not use indicator variables, R:BASE displays an error message when it encounters a null value, but produces no rows.

Placing a Value into a Numeric Variable

If you use FETCH to place a value into a variable that has not been previously defined and has a NUMERIC data type, then that variable acquires the precision and scale of the column from which the value is fetched.

Using the FETCH Command Without Variable Specification

Using FETCH *cursorname* without any variable specification will retrieve the next row from the cursor. Use the SET VAR *varname* WHERE CURRENT OF *cursorname* to retrieve the columns you need.

Example

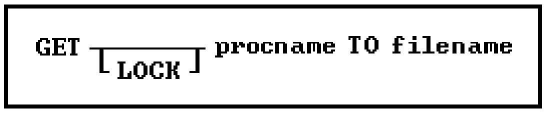
The following command lines fetch every other row from a table.

```
DROP CURSOR C1
DECLARE c1 SCROLL CURSOR FOR SELECT transid, transdate, +
   netamount FROM transmaster ORDER BY netamount DESC
OPEN c1
FETCH c1 INTO vtransid ind1, vtransdate ind2, vnetamount ind3
SELECT COUNT(*) INTO vtotcount i1 FROM transmaster
SET VAR vcount INT=0
WHILE sqlcode <> 100 THEN
   SET VAR vcount = (.vcount+1)
   WRITE 'Total count', .vtotcount, 'Cursor count', .vcount
   --fetch every other row
   FETCH REL 2 FROM c1 INTO vtransid int1, +
      vtransdate ind2, vnetamount ind3
ENDWHILE
CLOSE C1
DROP CURSOR C1
```

6.8 G

6.8.1 GET

Retrieves a Stored Procedure.



Options

filename
The name of the ASCII text format file the Stored Procedure is placed in.

LOCK
Locks the procedure so it cannot be locked or unlocked by another user. When a procedure is locked, only the user placing the lock can replace the procedure. The NAME setting is used for identification of the user.

procname
The name of the procedure to retrieve.

About the GET Command

The GET command is used to read a Stored Procedure from the database into an ASCII file. If the LOCK option is used with the GET command, the procedure cannot be replaced by using the PUT command.

Rows are copied, not removed, from the source.

Example

The following command retrieves the SetOrderID procedure and places it into a file name SetOrdID.STP.

```
GET SetOrderID TO SetOrdID.STP
```

The following series of commands will retrieve the CreateTempTabs procedure and place it into a file name TempTabs.STP, then Edit the file, and finally replace the Stored Procedure from the file with an updated version.

```
GET CreateTempTabs TO TempTabs.STP
RBE TempTabs.STP
PUT TempTabs.STP AS CreateTempTabs
```

Note: The STP file extension is not required by R:BASE it is merely a suggestion for a meaningful naming convention.

6.8.2 GOTO

Use the GOTO command in a program to pass control to the commands following the LABEL command.

```
GOTO lblname
```

Option

lblname
Specifies a 1 to 18 character name that labels a line to skip to when a GOTO command is executed in a command or procedure file.

About the GOTO Command

You should limit the use of the GOTO command because GOTO runs more slowly than other R:BASE control structures. Instead, when possible, use a WHILE loop, SWITCH structure, or IF structure to build the command-file logic. Never use GOTO to exit from a WHILE loop or SWITCH structure.

Using the LABEL Command with GOTO

GOTO must have a corresponding LABEL command within the same command block or file. The LABEL command may precede or follow GOTO in the same command file or, in a procedure file, within the same command block.

You can use a variable containing the name of the label instead of using the specific label name in the GOTO command. To do this, you must use a dot or ampersand variable to tell R:BASE to use the contents of a variable, rather than the variable name as the label name.

Examples

The following example uses a dotted variable containing the name of the label instead of using the specific label name in the GOTO command. If the variable was not a dotted variable, R:BASE would search for a label named *vlabel*. Because it is a dotted variable, R:BASE looks for the correct label name *label1*.

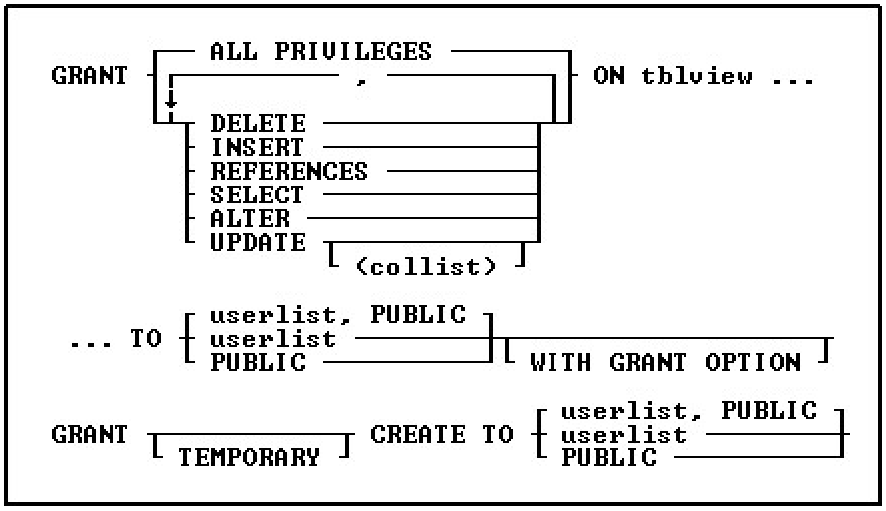
```
SET VARIABLE vlabel = 'label1'
GOTO .vlabel
```

The GOTO *lexit* command in the following example causes the commands following the ENDIF command to be skipped and the QUIT TO command to be run. The only way the commands between the IF structure and LABEL *lexit* command would be executed would be if the value of *v1* is not greater than the value of *.v2*.

```
IF v1 > .v2 THEN
 GOTO lexit
ENDIF
 .
 .
 .
LABEL lexit
QUIT TO caller
```

6.8.3 GRANT

Use the GRANT command to assign privileges to users of a table or view.



Options

,
Indicates that this part of the command is repeatable.

ALL PRIVILEGES
Grants all user privileges on the specified table, or on a view that can be updated.

ALTER
Grants permission to alter specific tables.

CREATE TO
Grants permission to users to create tables using the CREATE TABLE command. Users who have been granted permission to use this command have all privileges on the tables they create, including the WITH GRANT OPTION. However, users do not have privileges on any other tables in the database unless they are specifically granted permission by the owner.

DELETE
Grants permission to remove rows from the specified table or from a view that can be updated.

INSERT
Grants permission to add rows to the specified table or to a view that can be updated.

ON tblview
Specifies a table or view.

PUBLIC
Grants specified user privileges to all users.

REFERENCES
Grants permission to create a table with a foreign key that references a table with a primary key.

SELECT
Grants permission to display or print data for the specified table or view.

TEMPORARY
Grants permission to users to create temporary tables. Users who have been granted permission to use this command have all privileges on the temporary tables they create, including the WITH GRANT OPTION. However, users do not have privileges on any other tables in the database unless they are specifically granted permission by the owner.

UPDATE (collist)
Grants permission to change the values of columns in the specified table or a view that cannot be updated. If you do not include the optional (*collist*), the user can update all columns in the table. If you list columns, the user can update only the specified columns.

userlist
Grants specified user privileges to listed users. You must separate user identifiers with a comma (or the current delimiter). For a value with spaces, the userid must be enclosed in quotes.

userlist, PUBLIC
Grants specified user privileges to listed users and PUBLIC. Users in *userlist* can retain their user privileges if user privileges granted to PUBLIC are revoked. If, for example, *Ralph*, *Sam*, *Jane*, and PUBLIC have been granted certain user privileges, revoking those privileges from PUBLIC would not affect the three listed users. You must separate the user identifier with a comma (or the current delimiter). For a value with spaces, the userid must be enclosed in quotes.

WITH GRANT OPTION
Allows the specified users to pass the granted user privileges to other users. When you use the LIST ACCESS command, an asterisk is displayed in front of the user privilege to show a user can grant the assigned user privilege to others; for example, *SELECT means a user has permission to display or print data for specified tables or views, and can grant SELECT rights to other users.

About the GRANT Command

As the database owner, you must first set your own user identifier. After setting your user identifier, you can assign privileges to other users for the tables or views in your database. You must specifically grant privileges to other users. You can assign privileges for a table to individual users, to PUBLIC, or to both. Each user can have a different set of user privileges for the same table, and you can grant a user the right to grant user privileges to others. You can set your user identifier with the RENAME OWNER command and assign user privileges to other users by using the GRANT command.

In R:BASE for Windows you can also set your user identifier by choosing **Utilities: Set User ID and Password**. To assign user privileges to other users, choose the **User Privileges** option from the **Utilities** menu.

In R:BASE for DOS, you can also set a user identifier and assign access rights in RBDefine; enter the RBDEFINE command at the R> Prompt.

Granting User Privileges

You grant user privileges or access rights on tables or views, however, UPDATE rights must be granted at the column level and CREATE rights must be granted at the database level. If you assign more than one user privilege in a single GRANT command, separate the user privileges with a comma (or the current delimiter).

You can grant the following user privileges: ALL PRIVILEGES, ALTER, CREATE, DELETE, INSERT, REFERENCES, SELECT, and UPDATE; however, you can grant only the SELECT user privilege on views that cannot be updated.

Using User Identifiers and Passwords

A user identifier can be any unique string that uniquely identifies a user to the system. A user identifier can be of 128 characters (or less). To maximize security, create user identifiers that are difficult to guess-such as a random string of letters and numbers. Users can assign passwords to their user identifiers for an added level of security. For information about users assigning passwords see SET USER.

In a database where users have been assigned rights, printing reports requires one of these conditions:

- A user has been granted SELECT privileges on the driving table or view and any look-up tables.
- A user has been granted SELECT privileges or ALL PRIVILEGES on all tables used for the report.
- PUBLIC has been granted SELECT privileges on the driving table or view.
- PUBLIC has been granted SELECT privileges or ALL PRIVILEGES on all tables used for the report.

The only exception to this system of assigning rights is password-protected forms. Passwords assigned to forms, override user privileges assigned with the GRANT command. If a form has not been assigned a password, the user privileges you granted to the tables associated with the form are in effect.

Once R:BASE determines that a user can have access to a password-protected form, R:BASE does not verify user privileges on the underlying tables. Therefore, access to a password-protected form overrides table-level user privileges, making it possible for a user who does not have user privileges on a table to modify the information in that table.

Creating New Tables

To create new tables in a database, a user must be assigned the CREATE user privilege. R:BASE assigns all user privileges to the user for all tables created, including the GRANT user privilege.

A user must be assigned the SELECT user privilege to create a new table from existing tables using the PROJECT command. R:BASE assigns users who use these commands all user privileges on the new table. These user privileges do not include the GRANT user privilege.

Creating Views

CREATE VIEW also requires the SELECT user privilege on the existing tables. R:BASE assigns users who create views the same user privileges they have on the source table. For views that cannot be updated, R:BASE only assigns users the SELECT user privilege.

Command Authorization Requirements
The following three tables list R:BASE commands and the user privileges they require.

R:BASE Commands that Require the SELECT Access Right

Command	SELECT Access Right on...
CREATE VIEW	Component tables
DECLARE CURSOR	Table
FETCH	Table
OPEN CURSOR	Table
PROJECT	Table 1
SELECT	Table
SET VARIABLE *	Table
UNLOAD DATA	Table

* SET VARIABLE requires the SELECT user privilege only when the value of the variable is derived from a column.

R:BASE Commands that Require the UPDATE User Privilege

Command	UPDATE User Privilege on...
CREATE INDEX	Column
UPDATE	Column list

R:BASE Commands that Require Other User Privileges

Command	User Privilege	Access on...
DELETE	DELETE	Table or single-table view.
INSERT	INSERT	Table or single-table view, without calculations.

The following table lists the user privileges and the commands that use them. Some commands appear under more than one user privilege.

User Privileges for R:BASE Commands

Access Right	R:BASE Commands that Require The Access Right		
ALTER	ALTER TABLE	AUTONUM	DROP COLUMN
CREATE	ALTER TABLE	DROP	REVOKE *1*
	AUTONUM	GRANT *1*	RULES

	COMMENT ON	PACK	UNLOAD ALL
	CREATE TABLE	RELOAD	UNLOAD STRUCTURE
	RENAME		
Database owner's user identifier	ALTER TABLE	DROP	REVOKE *1*
	AUTONUM	GRANT *1*	
	COMMENT ON	PACK	RULES
	CREATE TABLE	RELOAD	UNLOAD ALL
		RENAME	UNLOAD STRUCTURE
DELETE	DELETE		
INSERT	INSERT	LOAD	
REFERENCES	LOAD	INSERT	UPDATE
SELECT	SET VARIABLE *4*	FETCH	SELECT
	CREATE VIEW	PROJECT	UNLOAD DATA
	DECLARE CURSOR		
UPDATE	CREATE INDEX	UPDATE	

1. GRANT and REVOKE do not require the database owner's user identifier for an user privilege that includes GRANT permission.

2. Form passwords override user privileges assigned with the GRANT command. If a form does not have a password, the INSERT, DELETE , SELECT, or UPDATE user privileges are required for the underlying tables.

3. Any user privilege granted allows users to list all tables for which they have user privileges.

4. SET VARIABLE requires the SELECT user privilege only when the value of the variable is derived from a column.

Revoking User Privileges

The database owner can remove user privileges with the REVOKE command. The syntax for the REVOKE command is the same as the syntax for the GRANT command. If you issue the REVOKE ALL PRIVILEGES command without specifying a table, R:BASE revokes all user privileges including ALTER and CREATE.

Examples

The following command grants user privileges to display the view named *SLSView* to a specific user-*Jane*, and to all users-PUBLIC.

```
GRANT SELECT ON SLSView TO Jane, PUBLIC
```

The following command grants user privileges to add or remove information to or from the *TransMaster* table to any user entering the user identifier *Sam* or *Ralph*.

```
GRANT INSERT, DELETE ON TransMaster TO Sam, Ralph
```

The following command grants user privileges to display and enter information in the *TransMaster* table. Also, the command allows any user entering the user identifier *Jane* to pass the SELECT and INSERT user privileges on to other users.

```
GRANT SELECT, INSERT ON TransMaster TO Jane WITH GRANT OPTION
```

The following command grants the user *Abe*, who is not the database owner, permission to alter the *Customer* table.

```
GRANT ALTER ON Customer TO Abe
```

The following command line grants the user *Abe*, who is not the database owner, permission to create tables.

```
GRANT CREATE TO Abe
```

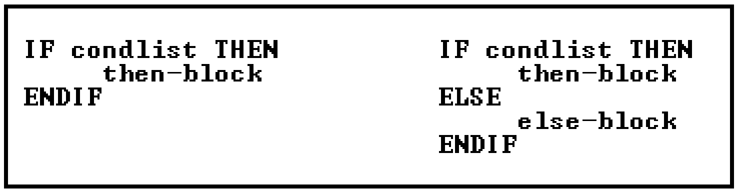
The following command line grants the user *Noah* permission to create temporary tables.

```
GRANT TEMPORARY CREATE TO Noah
```

6.9 I

6.9.1 IF/ENDIF

Use an IF...ENDIF structure in a command file to cause a block of commands to be run when the specified conditions are met.



```
IF condlist THEN              IF condlist THEN
     then-block                    then-block
ENDIF                         ELSE
                                   else-block
                              ENDIF
```

Options

condlist
Lists a set of conditions that combine to form a statement that is either true or false. Conditions are combined with the connecting operators AND, OR, AND NOT, and OR NOT.

else-block
Contains one or more R:BASE commands to execute when the conditions specified in *condlist* are false.

then-block
Contains one or more R:BASE commands to execute when the conditions specified in *condlist* are true.

About the IF...ENDIF Command

When the conditions in an IF...ENDIF structure are true, R:BASE runs all the commands between THEN and ELSE, or if the ELSE option is not included, between the THEN and ENDIF.

If you use the ELSE option and the conditions are false, R:BASE runs the block of commands between the ELSE and the ENDIF. If you do not use the ELSE option and the conditions are false, R:BASE runs the command line immediately after ENDIF.

IF...ENDIF structures can be nested with other IF...ENDIF structures.

IF...ENDIF structures can be on a single line in a command file. You cannot put an IF...ENDIF structure on a single line when any of the following occur in a command file:

- The last command in the then-block is QUIT.
- The structure contains an else-block.

Using Conditions in an IF...ENDIF Structure

The conditions for an IF...ENDIF structure are listed in the table below.

Condition	**Description**
varname IS NULL	The value of the variable is null.
varname IS NOT NULL	The value of the variable is not null.
varname CONTAINS *'string'*	The variable has a TEXT data type and contains a '*string*' as a substring in the variable value.
varname NOT CONTAINS *'string'*	The variable has a TEXT data type and a *'string'* is not contained as a substring in the variable value.
varname LIKE *'string'*	The variable equals a '*string*.' A *'string'* can contain wildcards.

varname NOT LIKE *'string'*	The variable does not equal the *'string*'. A '*string'* can contain wildcards.
varname BETWEEN *value1* AND *value2*	The value of the variable is greater than or equal to *value1* and less than or equal to *value2*. The variable and the values must be the same data type.
varname NOT BETWEEN *value1* AND *value2*	The value of the variable is less than *value1* or greater than *value2*. The variable and the values must be the same data type.
varname IN *(valuelist)*	The value of the variable is in the value list.
varname NOT IN *(valuelist)*	The value of the variable is not in the value list.
item1 op *item2*	*Item1* has the specified relationship to *item2. Item1* can be a column name, value, or expression; *item2* can be a column name, value, or expression.

The valid operators (*op*) for the conditions in an IF...ENDIF structure are listed in the table below. Do not use wildcard characters with these operators.

Operator	Description
=	Equals
<=	Less than or equal to
>=	Greater than or equal to
<	Less than
>	Greater than
<>	Not equal

An expression can be substituted for the first variable in each of the conditions. The second variable in the comparison must be dotted so that the value of the variable is used, not the variable name.

Wildcards can be used with LIKE or NOT LIKE (for example, *varname* LIKE 'string%').

If you use multiple AND and OR operators, you must enter them in the correct order, or use parentheses to obtain the desired result. If SET AND is on (the default setting) conditions connected by AND are evaluated first; then conditions connected by OR are evaluated.

Examples

The following command lines show three nested IF...ENDIF structures.

```
IF vqtyord > .vlastqty THEN
 IF vqtyord <> 0 THEN
 IF vchng > 0 AND vbackord IS NOT NULL THEN
 .
 .
 .
 ELSE
 .
 .
 .
 ENDIF
 ENDIF
 ENDIF
```

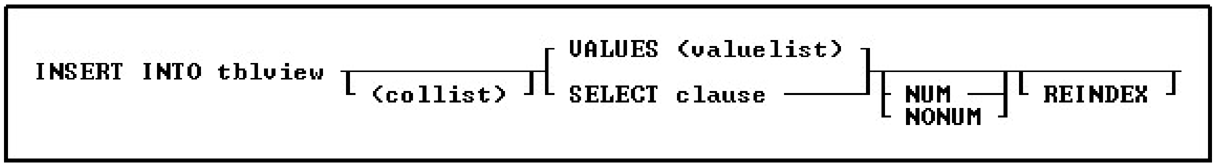
The following example shows an IF...ENDIF structure on a single line.

```
IF v2 = 0 THEN ; SET VARIABLE v1 = (.v1 + 1) ; ENDIF
```

Copyright © 1982-2025 R:BASE Technologies, Inc.

6.9.2 INSERT

Use the INSERT command to add data to a table or view without using a data-entry form.



Options

(collist)
Specifies a list of one or more column names, separated by a comma (or the current delimiter). In an SQL command, any column name in the list can be preceded by a table or correlation name and a period (*tblname.colname*).

INTO tblview
Specifies the table or view name (views must be updatable).

NUM
NONUM
NUM specifies that autonumbering columns will be numbered as they are inserted. NONUM turns off autonumbering while inserting, thereby allowing inserting of a specific value for autonumber columns. The default is NUM.

SELECT clause
Finds values in a table, tables, or view to insert into the table or view specified by the INTO *tblview* option and the columns specified by the *collist* option.

VALUES (vallist)
Specifies a list of values to insert into the table specified by the INTO *tblview* option and the columns specified by the *collist* option. Separate values with a comma or the current delimiter.

For these data types...	**Use this format for *vallist***
All data types except BIT, BITNOTE, LONG VARBIT, and VARBIT	'string' or value
BIT, BITNOTE, LONG VARBIT, LONG VARCHAR, VARBIT, and VARCHAR	['filename.ext'] or ['filename.ext', filetype, offset, length] **Note:** When you use VARCHAR, the filetype is always TXT. When you use VARBIT, BIT, and BITNOTE, filetype refers to the standard graphical file types.

About the INSERT Command

The INSERT command assigns a default value of null to any column not named in the *collist* unless a default value has been assigned to a column with the CREATE TABLE or ALTER TABLE command.

To ensure that rules are checked while adding data with the INSERT command, SET RULES ON before running the INSERT command.

The setting of the SET ZERO command affects the calculation of numeric computed columns. To have null values treated as zeros in expressions, set ZERO on. When ZERO is set off, if the value of a column used in a expression is null, the computed value will be null.

You cannot insert values into the table used in the SELECT clause.

To ensure that data is placed in the intended column, use the following guidelines:

- Do not embed commas within entries for CURRENCY, DOUBLE, INTEGER, NUMERIC, or REAL data types. R:BASE automatically inserts commas and the current currency symbol.
- When values for CURRENCY, DOUBLE, NUMERIC, or REAL or data types are decimal fractions, you must enter the decimal point. When values are whole numbers, R:BASE adds a decimal point for you at the end of the number. R:BASE adds zeros for subunits in whole currency

values; For example, using the default currency format, R:BASE loads an entry of 1000 as $1,000.00.
- When values for NOTE or TEXT data types contain commas, you can either enclose the entries within quotes, or use SET DELIMIT to change the default delimiter (comma) to another character.
- When values for NOTE or TEXT data types contain single quotes ('), and you are using the default QUOTES character ('), use two single quotes ('') in the text string. For example, 'Walter Finnegan''s order.'
- When a value you specify for a column is not the same data type as the column's data type, R:BASE displays an error message and you need to re-enter the entire row.
- When values for NOTE or TEXT data types exceed the maximum length of a column, R:BASE truncates the value and adds it to the table. A message is displayed that tells you which row has been truncated.

Inserting an Autonumbered Column

When you use INSERT to add a row, INSERT assigns the next available number to autonumbered columns in the table. Therefore, omit autonumbered columns and their values from a *collist.* Also, if you use the SELECT option, omit an autonumbered column from the *collist*. If a value is included for an autonumbered column that was omitted from the column list, R:BASE does not run the command because it cannot identify which column to load.

Inserting a Computed Column

Because a computed column's value is calculated, you cannot insert a new value. Omit computed columns from a *collist* or, if you are adding data to all columns, do not use a *collist* and do not specify a value for the computed column. R:BASE will skip the computed column when the row is inserted.

Examples

In the following example, the *sales* table has three columns, *col1*, *col2*, and *col3*; and *col2* is a computed column. To insert a row, you would only specify values for *col1* and *col3*. In this example, the value for *col1* is 100, and the value for *col3*is 200.

If the expression for *col2* was (*col1 + 200*), then *col2* would have the value 300 when the row is inserted.

```
INSERT INTO sales VALUES (100, 200)
```

In the following example, a *vallist* adds a new row to the *product* table, filling the *model*, *prodname*, *proddesc*, and *listprice* columns.

```
INSERT INTO product (model, prodname, proddesc, listprice) +
   VALUES ('PB3060', 'Portable Advanced PC', 'System-Single +
   Drive w/Hard Disk-Portable', 3795)
```

The following command uses a *vallist* with global variables to insert the values from variables *v1*, *v2*, and *v3* into the *bonusrate* table.

```
SET VARIABLE v1 CURRENCY = 50000, v2 CURRENCY = 75000, +
   v3 REAL = .10
INSERT INTO bonusrate VALUES (.v1, .v2, .v3)
```

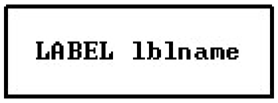
The following example adds rows to *customer* table selected from *temp* table. It adds data into the *company* and *custphone* columns. The columns taken from the *temp* table can have different column names, the data types must be the same, and the order and number of columns in the column list of the source table (designated by SELECT) must match the column list of the destination table (designated by INTO).

```
INSERT INTO customer (company, custphone) +
   SELECT cname, phone FROM temp
```

6.10 L

6.10.1 LABEL

In a GOTO or WHENEVER statement, use the LABEL command to identify the command line to which control should be passed.



```
LABEL lblname
```

Option

lblname
Specifies a 1 to 18 character name that labels a line to skip to when a GOTO command is executed in a command or procedure file.

MODAL
Allows an EEP command block to launch a Form, Label or Report in a designer. After making appropriate changes, you will have to save the changes and close the designer in order to continue the next command in your EEP.

About the LABEL Command

After a command file is run once, R:BASE stores the labels in memory. When the command file is run again, R:BASE retrieves the location of a label when the GOTO or WHENEVER is run. However, if the GOTO or WHENEVER command is going to be run only once, place the LABEL command below the GOTO or WHENEVER command because it is more efficient for R:BASE to search downward in the command file for the matching label.

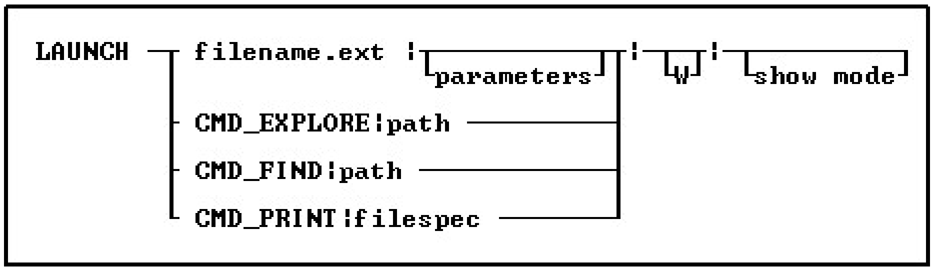
Example

In the following example, if the *vctrvar* variable in the IF...THEN statement is equal to 999, control passes to the command lines following the LABEL command, which defines the *endproc* label. If *vctrvar* does not equal 999, the command lines following the ELSE statement are performed.

```
IF vctrvar = 999 THEN
   GOTO endproc
ELSE
   *( commands to execute )
ENDIF

.
.
.

LABEL endproc
   *( commands to execute )
```

6.10.2 LAUNCH

The LAUNCH command invokes or runs executable programs, as well as files associated with programs.



Options

|
The pipe character is needed when a file parameter, W (wait), or show mode is specified. The number of pipes needed depend on the last part of the LAUNCH command being used. If only a file is being launched and no addition portions of the LAUNCH command are being used, the pipe characters are not needed. However, if a file is being launched and only a show mode is used, then all three pipes are required.

CMD_EXPLORE
Displays the Windows Explorer from the specified folder.

CMD_FIND
Initiates a search from the specified folder.

CMD_PRINT
Prints a file to the default printer, or displays a print dialog, based upon file type.

filename.ext
The name of file which has a corresponding association registered with Windows. If the file does not exist, an error will be displayed.

filespec
The name of the file to print. Based upon the file type, the file will be sent directly to the default printer, or will display in a print dialog, If the file does not exist, or if the file is not printable, an error will be displayed.

path
Specifies the directory name.

<parameters>
To specify additional parameters associated to the launched program.

show mode
Specifies how the launched program or operation is displayed. The show modes may not be supported by some programs.

- NORMAL - launches the program in a normal window
- MINIMIZED - launches the program in a minimized window
- MAXIMIZED - launches the program in a maximized window
- HIDE - launches the program, but it is hidden from display

W
To specify the "Wait Until Finished" option.

About the LAUNCH Command

LAUNCH command is used to execute (or LAUNCH) another application or process, from within R:BASE for Windows. Based upon the extension for the FileName, a corresponding program associated to it within

Windows will open when launching the file. For example, setting FileName to 'README.TXT' will invoke the Windows NotePad when launched. If a full path is not specified, the current search path is used.

The LAUNCH command allows for command line parameters to be specified that will be recognized by the program being launched. The launched application will start in the same directory as the executable file or in the directory referenced, if specified.

The LAUNCH command also allows you to use a "Wait Until Finished" option to specify whether or not you want R:BASE to sleep while the launched program executes or continue running. If the "Wait Until Finished" parameter "|W", (e.g. pipe and W with no space) is added to the LAUNCH command, R:BASE will sleep until the launched process completes. If the "Wait Until Finished" parameter "|W" is not specified, the launched process is executed in its own thread, and the R:BASE will continue to execute.

The number of pipes needed depend on the last part of the LAUNCH command being used. If only a file is being launched and no addition portions of the LAUNCH command are being used, the pipe characters are not needed. However, if a file is being launched and only a show mode is used, then all three pipes are required.

Launch Command Features:

- LAUNCH command is supported in EEPs
- If a folder path, file name, or parameter contains spaces, double quotes (") must be added.
- The LAUNCH command will execute ANY windows program as long as it is installed and the file extension is associated within the operating system.
- No flashing screens (in and out of DOS)
- No need to use the ZIP OUT command or run batch files, etc.

Examples

Example 01:

```
LAUNCH charts.VBS
```

This will invoke VBScript:Windows Scripting Host Sample and will demonstrate how to access Excel using the Windows Scripting Host.

Example 02:

```
LAUNCH MyDoc.PDF
```

This will invoke Adobe Reader and display the file.

Example 03:

```
LAUNCH c:\mydocs\ReadMe.txt |||MAXIMIZED
```

This will invoke the Notepad with README.TXT in a maximized window. All three pipes are required as the "show mode" parameter is specified.

Example 04: (LAUNCHing NotePad with the "Wait Until Finished" option)

```
LAUNCH ReadMe.txt||W
```

This will invoke the Windows Notepad while R:BASE sleeps in the background. You will have to Exit Notepad in order to get control back to R:BASE the application. Notice the two pipe characters with no spaces right after the file name. The first pipe with no parameters and no space while the second pipe with the "Wait Until Finished" |W option.

Example 05:

```
LAUNCH CMD_EXPLORE|C:\RBTI
```

This will launch Windows Explorer and display the contents of the C:\RBTI folder.

Example 06:

```
LAUNCH CMD_FIND|F:\DATA
```

This will initiate a search from the F:\Data folder.

Example 07:

```
LAUNCH CMD_PRINT|C:\ReadMe.txt
```

this will print the ReadMe.txt file to the printer.

Example 08: (LAUNCH command with Parameters and "Wait Until Finished" parameter)

```
LAUNCH 'winword|"C:\Word Documents\TestFile.DOC" "/mMacroName"|W'
```

This will invoke the MS Word with TestFile.DOC running the MacroName macro.

This will invoke MS Word with Document.doc using MacroName as a main window while R:BASE sleeps in the background. You will have to Exit MS Word in order to give control back to the R:BASE window/application. Notice the additional pipe "|" symbol with "W" right after /mMacroName to specify Wait Until Finish option.

Since the folder path contains spaces, double quotes (") are added.

Example 09:

```
LAUNCH CMD_PRINT|N:\Sales\OrderForm.pdf
```

This prints the OrderForm.pdf document.

Example 10: (To automate e-mails via R:BASE)

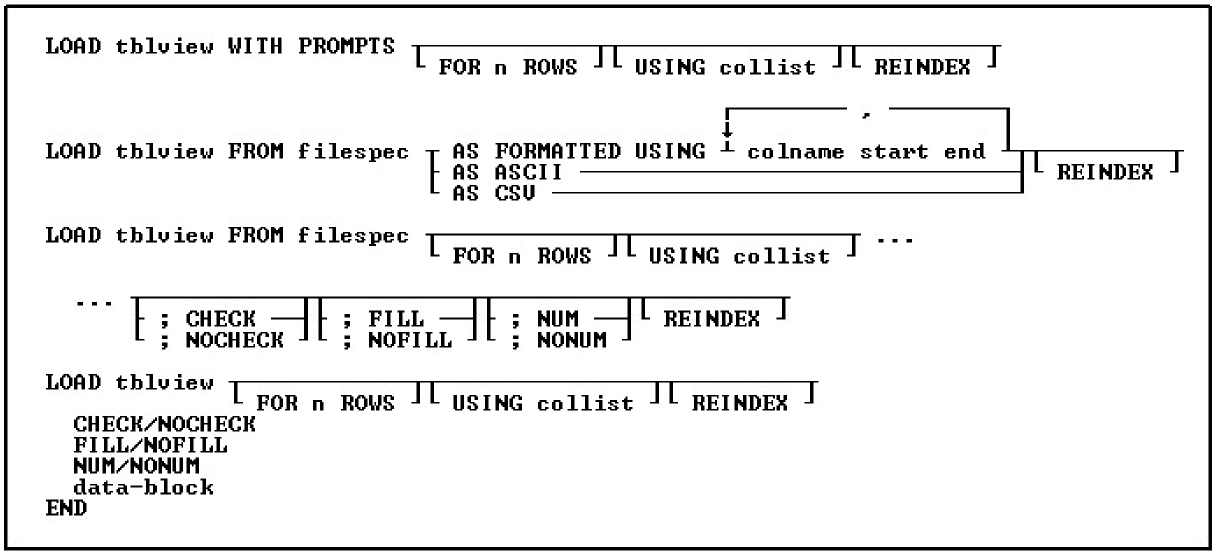
If you know the e-mail tags of your default e-mail program, using the LAUNCH command you could easily achieve that task of automating e-mails.

```
SET VAR vMailTo TEXT = NULL
SET VAR vCustID INTEGER = 2001
SELECT ('mailto:'+EMailAddress) INTO vMailTo INDIC IvMailTo +
FROM Contacts WHERE CustId = .vCustID
LAUNCH .vMailTo
```

This will launch the default e-mail program with the "To:" filled in. You can follow the same logic for other e-mail tags (e.g. Subject, Cc, Bcc, etc.).

6.10.3 LOAD

Use the LOAD command to add data to a table or to a single table view that can be updated.



```
LOAD tblview WITH PROMPTS [FOR n ROWS] [USING collist] [REINDEX]

LOAD tblview FROM filespec  AS FORMATTED USING colname start end ,  [REINDEX]
                            AS ASCII
                            AS CSV

LOAD tblview FROM filespec [FOR n ROWS] [USING collist] ...

  ... [; CHECK  ] [; FILL  ] [; NUM  ] [REINDEX]
      [; NOCHECK] [; NOFILL] [; NONUM]

LOAD tblview [FOR n ROWS] [USING collist] [REINDEX]
  CHECK/NOCHECK
  FILL/NOFILL
  NUM/NONUM
  data-block
END
```

Options

,
Indicates that this part of the command is repeatable.

AS ASCII
The AS ASCII command parameter is designed strictly for speed of operation. AS ASCII checks rules and constraints. However, following is a list of the limitations of the AS ASCII command parameter:

- It does not check the data types; therefore, invalid data will be loaded as null values into columns; no error messages about this conversion are displayed.
- It does not display error messages when columns must be truncated, or when excess data exists on any line.
- It does not echo data to the screen regardless of the setting for SET ECHO.

To achieve maximum speed of loading, the data must look like the data that R:BASE unloads with the UNLOAD DATA AS ASCII command. That is, the data must conform to the following:

- The carriage return/line feed characters define the end of the line for a given row.
- The maximum row size is 32,768 characters.
- The data cannot include variables.
- The data cannot include comments.

AS FORMATTED USING
Loads data from an ASCII file when the data is formatted in fixed column locations, with the following restrictions:

- The carriage return/line feed characters define the end of the line for a given row.
- The maximum row size is 32,768 characters.
- The data cannot include variables.
- The data cannot include comments.
- You must specify the name of each column of the table to be loaded, and the starting and ending position of its data in the line, which is specified in the USING clause of this command.

AS CSV
Loads data from a comma-separated values (CSV) file. AS CSV checks rules and constraints. The following is a list of the limitations of the AS CSV command parameter:

- It does not check the data types; therefore, invalid data will be loaded as null values into columns; no error messages about this conversion are displayed.
- It does not display error messages when columns must be truncated, or when excess data exists on any line.
- It does not echo data to the screen regardless of the setting for SET ECHO.

To achieve maximum speed of loading, the data must look like the data that R:BASE unloads with the UNLOAD DATA AS CSV command. That is, the data must conform to the following:

- The carriage return/line feed characters define the end of the line for a given row.
- The maximum row size is 32,768 characters.
- The data cannot include variables.
- The data cannot include comments.

CHECK
NOCHECK
CHECK turns on rule checking. When rule checking is on, R:BASE checks input against data validation rules. NOCHECK turns off rule checking. CHECK and NOCHECK override the current setting of the SET RULES condition. The default is CHECK.

colname start end
Specifies the name of a column in the table and the starting and ending position of its data in the line; this option is used with the AS FORMATTED option.

data-block
Includes lines of data to be loaded, as well as the LOAD subcommands.

For these data types...	**Use this format for *data-block***
All data types except BIT, BITNOTE, LONG VARBIT, and VARBIT	'string' or value
BIT, BITNOTE, LONG VARBIT, LONG VARCHAR, VARBIT, and VARCHAR	['filename.ext'] or ['filename.ext', filetype, offset, length] **Note:** When you use VARCHAR, the filetype is always TXT. When you use VARBIT, BIT, and BITNOTE, filetype refers to the standard graphical file types.

FILL
NOFILL
FILL makes null any columns that have not been assigned values. All of the missing values must be at the end of the row. If a rule specifies that a column requires an entry other than null, do not use FILL. NOFILL turns off FILL and requires a value for each column. The default is NOFILL.

FOR n ROWS
Directs R:BASE to stop processing after loading *n* rows, where *n* is a positive whole number. In the fourth syntax diagram, END is not used if FOR *n* ROWS is included.

FROM filespec
Loads data into the specified table with data from an external ASCII delimited file.

NUM
NONUM
NUM specifies that autonumbering columns will be numbered as they are loaded. NONUM turns off autonumbering while loading, thereby allowing loading of a specific value for autonumber columns. The default is NUM.

tblview
Specifies a table or view name to load.

USING collist
Specifies the column(s) to use with the command.

WITH PROMPTS
Loads data into the specified table from keyboard entries. R:BASE asks for the values of each column by displaying the column name and its data type. To end the loading session, press [Esc].

About the LOAD Command

You cannot load data into a multi-table view.

Instead of using LOAD, you can also use INSERT, the Data Editor, or a Form to add data to a table.

You can use the LOAD command to load data into R:BASE from a file that was not created by R:BASE. The file must be an ASCIIfile, either delimited or fixed.

The LOAD command will differentiate between END and 'END'; FILL and 'FILL'; NOFILL and 'NOFILL'; CHECK and 'CHECK'; NOCHECK and 'NOCHECK'; NUM and 'NUM'; NONUM and 'NONUM'. So, make sure to use the proper syntax when creating LOAD statements.

To ensure that data is placed in the intended column, use the following guidelines:

- Do not embed commas within entries for CURRENCY, DATE, DATETIME, DOUBLE, INTEGER, NUMERIC, or REAL data types. R:BASE automatically inserts commas and the current currency symbol.
- When values for CURRENCY, DOUBLE, NUMERIC, or REAL or data types are decimal fractions, you must enter the decimal point. When values are whole numbers, R:BASE adds a decimal point for you at the end of the number. R:BASE adds zeros for subunits in whole currency values. For example, using the default currency format, R:BASE loads an entry of 1000 as $1,000.00.
- When values for NOTE or TEXT data types contain commas, you can either enclose the entries within quotes, or use SET DELIMIT to change the default delimiter (comma) to another character.
- When values for NOTE or TEXT data types contain single quotes ('), and you are using the default QUOTES character ('), use two single quotes ('') in the text string. For example, 'Walter Finnegan''s order.'
- When a value you specify for a column is not the same data type as the column's data type, R:BASE displays an error message and you need to re-enter the entire row.
- When values for NOTE or TEXT data types exceed the maximum length of a column, R:BASE truncates the value and adds it to the table. A message is displayed that tells you which row has been truncated.

Loading with a USING Clause
A USING clause is helpful when you do not have all the information that is to be added to a table. The following example lets you enter some information for a product but does not require that all columns be entered. The *model* and *listprice* columns are the first and last columns in the *product* table. The *prodname* and *proddesc* columns are not included in the command and are loaded with null values. You can later edit the *product* table to enter data into the columns that have null values.

```
LOAD product USING model listprice
```

Loading with the CHECK Option
The SET RULES condition does not have any effect on the CHECK option because CHECK has precedence over a RULES setting. When RULES is set off, the CHECK option still verifies data entry against existing rules.

When a user identifier has been assigned to the database owner, you must enter the owner's user identifier with the CONNECT or SET USER command before you use the CHECK or NOCHECK option. R:BASE does not accept the CHECK or NOCHECK option unless the owner's user identifier has been entered.

Loading Computed Columns
You cannot load data directly into a computed column. After you load the column values that are used to calculate the computed column, R:BASE fills the computed column with the computed value.

The setting of the SET ZERO condition affects the calculation of numeric computed columns. To have null values treated as zeros in expressions, set ZERO on. When ZERO is set off, if the value of a column used in a expression is null, the computed value will be null.

Loading Negative CURRENCY values
When loading negative CURRENCY values into a table, the format must include the hyphen, i.e -$500.00. Negative CURRENCY values encased in parenthesis are not recognized, e.g. ($500.00).

Loading with Prompts

When you run the LOAD command using prompts, you load one row of data at a time into the table you specified. (See Example 1). For each new row you add, R:BASE displays the name and data type of the row's column as prompts. At each prompt, you enter the value that you want the column to contain. You are prompted for each column in the row beginning with the first column, unless you used a USING *collist* clause to limit the number of the columns to load, or to change the order in which the columns are loaded. Any columns not listed in the *collist* are given null values when the rows are entered.

When you load data with prompts, the default length for a text entry is 80 characters. To enter columns with a NOTE or TEXT data types that contain more than 80 characters, load the data without prompts, make a custom data-entry form, or set the WIDTH so you can enter more characters.

R:BASE does not prompt you for computed or autonumbered column values.

Loading without Prompts
Loading without prompts is faster but requires that you remember the order of the columns in the table. When you load without using prompts and not from an ASCII file, the LOAD command provides its own distinctive prompt. The following options can be entered at this prompt: CHECK, NOCHECK, FILL, NOFILL, NUM, and NONUM.

Loading from an ASCII File
Use the LOAD command from the R> Prompt or a command file to load data into an existing table from both delimited and fixed field ASCII files. Each record in the ASCII file corresponds to one row of data in a table, and each item of data in a record corresponds to one column value in a row. Therefore, organize data in the file in the same order as the columns in the table to be loaded.

Items of data in a line of the ASCII file must be delimited to be properly placed within the columns of a row. The delimiter character must be the same as the current delimiter character specified with the SET DELIMIT setting. (The default delimiter is a comma.) R:BASE also accepts a blank space as a delimiter, regardless of the setting of the DELIMIT setting.

Data can be loaded in a fixed-field formatted ASCII file with the AS FORMATTED option. The column name and the start and end positions within the file must be specified for each value in the row of data that is to be loaded. When the start and end positions are specified, the delimiter character does not have any effect because the start and end positions for each column identify the data.

When loading from a file, be sure that the current null symbol is not a blank. If the first four characters of a field in a file are blank, R:BASE adds the column as a null column and does not read any additional characters that make up the field value.

When loading data from an ASCII file, make sure the file meets the following requirements listed in the table below.

Elements in an ASCII File	**Requirement**
INTEGER data types	Items of data to be loaded into columns with INTEGER data types cannot contain internal commas unless the item is enclosed in quotes. The default QUOTES character in R:BASE is a single quote ('); if your ASCII file uses double quotes ("), change the QUOTES setting before you load the file. If the file does not have quotes around the integer values containing commas, you must edit the ASCII file to remove any commas from the integer values, or enclose each integer value in quotes.
Embedded punctuation	Items of data containing ampersands, commas, embedded blanks, plus signs, equal signs, or semicolons must be enclosed in quotes if they are to be loaded into columns with a TEXT or NOTE data type. The default QUOTES character in R:BASE is a single quote ('); if your ASCII file uses double quotes ("), change the QUOTES setting before you load the file.
Embedded quotes	Items of data requiring quotes can also contain embedded quotes. For example, the item '*Basic*' *Keyboard* contains both a blank space and embedded quotes. Using single quotes ('), which is the default QUOTES setting, to add enclosing quotes, the item would looks like this: '''Basic'' Keyboard'
Currency	R:BASE automatically adds a currency symbol, commas, and zeros for currency units. For example, using the default currency format,

	R:BASE loads an entry of *1000* as *$1,000.00.*
Dates	The SET DATE SEQUENCE command sets the sequence for the date-dates in the file are loaded if the dates match the current date sequence established with the SET DATE command.
Computed columns	If the table being loaded has computed columns and the file contains values for the computed columns, R:BASE tries to load the computed column's value from the file into the column following the computed column. This results in an error because the data type of the next column might not be the correct data type, or the file will have too many values for the table because R:BASE does not load the computed column's value from the file.
Rules processing	Unless you run the SET RULES OFF condition before loading the file, rules processing is in effect. When an incoming data item violates a rule, R:BASE does not load the row. Instead, R:BASE displays the message for the rule that has been violated. To see the data that causes a rule violation, SET ECHO ON when loading a table and use the [Pause] key to stop the screen from scrolling when the rule violation occurs.

Loading a Data Block
The data block shown in the diagram can include lines of data and any of the options available with LOAD-CHECK/NOCHECK, FILL/NOFILL, and NUM/NONUM. You can intersperse the options with data lines, and you can enter more than one option on a line if you separate the options with semicolons. However, you cannot combine data and options on the same line.

R:BASE displays the dialog prompt to accept data-block entry. LOAD adds data to a table, row by row, without using a data-entry form and without prompting for each data item.

You can enter the options for the LOAD command at the dialog prompt at any time during data loading. Or you can include them on the command line, separated from the command by semicolons, as shown in the example below. (Do not use this format in command or procedure files. All options must follow the LOAD command on separate lines in command or procedure files.)

```
LOAD transdetail ; CHECK ; NUM
```

You can use global or system variables instead of constant values in the data block.

To enter values properly, use the following guidelines.

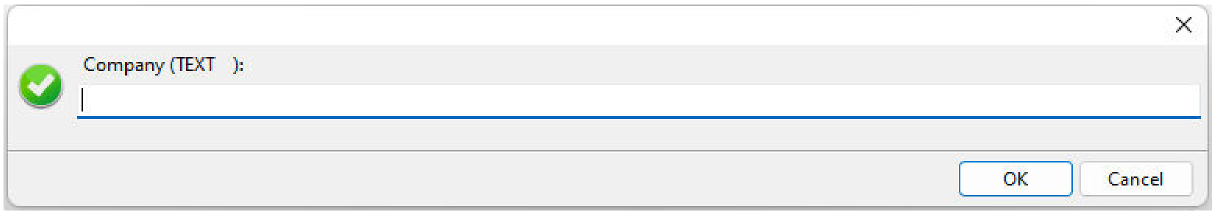
- Enter column values in the order that columns are defined in the table, and separate the values with a delimiter character. The default delimiter character is the comma.
- You can enter up to 75 characters on a single line. If a row is longer than 75 characters, continue on to the next line by typing past the end of the current line or by entering a plus (+) sign at any point on the current line. The plus sign must be the last entry on the line. The new line will begin with a +> prompt to indicate the continuation of the current line. If you are using this form of the LOAD command in a command file, you must use a + to continue on the next line; the lines will not automatically wrap.
- For other requirements on loading data, see "Loading from an ASCII File" earlier in this entry.

Examples

Example 01:
The following command line allow you to load rows containing new customer information to the *customer* table. R:BASE asks with prompts for each column by column name and data type. Two columns in the table, *custid* and *custphone*, are omitted from the list. R:BASE automatically supplies a number for the *custid* column because it is an autonumbered column. R:BASE leaves the *custphone* column empty (null) when data is loaded, and does not prompt for either column.

```
  LOAD customer WITH PROMPTS USING company, +
  custaddress, custcity, custstate, custzip
```

After the above command is run, the WITH PROMPTS option displays the message below.



If you press the [Esc] key or the "Cancel" button before you have finished entering data in a row, the row is not added to the table. You will be prompted to add another row.

To continue, press the "Yes" button. To exit, press the "No" button or the [Esc] key.

Example 02:
The following command loads five rows of data into the *customer* table from CUST.DAT, a delimited ASCII file. The data in the ASCII file must be in the same order as the columns in the *customer* table. Only the first five lines from the file will be loaded:

```
LOAD customer FROM cust.dat FOR 5 ROWS
```

Example 03:
In the following example, the command line tells R:BASE to start loading data for the *customer* table. A dialog prompt is displayed for each new row. Each column value would be entered in this one dialog and separated with a comma, or the current delimiter. The legnth of the text available to fit in the dialog is 4096 characters.

```
LOAD customer
```

Example 04:
After the command line in the following example is run, R:BASE expects the next five lines entered at the dialog prompt to contain data to be loaded into the *customer* table. After the fifth line of data is entered, the loading ends. To end loading before five rows of data are entered, enter END.

```
LOAD customer FOR 5 ROWS
```

Example 05:
The following command lines show you how to load data into the *company*, *custaddress, custcity, custstate*, and *custzip* columns of the *customer* table. The *custid* and *custphone* columns in the *customer* table will not have data loaded and will be given null values.

```
LOAD customer FROM customer.fix AS FORMATTED +
USING company 11 50, custaddress 51 80, +
custcity 81 100, custstate 101 102, custzip 103 112
```

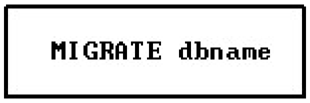
Example 06:
When you use the LOAD command, you must omit values for computed or autonumbered columns. Instead, enter the value for the next column in the data list. In the following example, to add a row to the *transdetail* table, which has a computed column, you would only enter data for the first five columns; the sixth column is a computed column based on the fourth and fifth columns. The columns entered are *transid*, *detailnum*, *model*, *units*, and *price*. The computed column is *extprice* and has the expression (*units* * *price*).

```
LOAD transdetail
  6000,1,'CX3000',100,$1900
END
```

6.11 M

6.11.1 MIGRATE

The MIGRATE command is used to convert R:BASE eXtreme 9.5 (64) databases to R:BASE 11.



```
MIGRATE dbname
```

Options

dbname
Specifies the database to be converted.

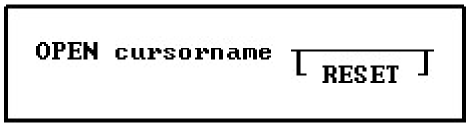
About the MIGRATE Command

R:BASE Enterprise requires the conversion of existing 9.5 (64) database for structural changes. Once the database is converted, it CANNOT be accessed by any previous version of R:BASE. Be sure and backup your database before you migrate it.

6.12 O

6.12.1 OPEN

Use the OPEN command before using the cursor designated by the DECLARE CURSOR command.



```
OPEN cursorname [ RESET ]
```

Options

cursorname
Specifies a 1 to 18 character cursor name that has been previously specified by the DECLARE CURSOR command.

RESET
Reopens a cursor with the current values of any variables referenced in the DECLARE CURSOR statement. This improves performance by eliminating the need to re-optimize the query.

About the OPEN Command

OPEN evaluates the SELECT clause of the DECLARE CURSOR command using the current values of any variables that it contains. Then OPEN stores that copy of the cursor definition and places the cursor before the first row.

After you close a cursor with the CLOSE command, you can reopen it by repeating the OPEN command. Every time you open a cursor, R:BASE reads the rows again, so that any changes you previously made through the cursor are visible when you look at the rows.

When using the RESET option, the WHERE clause is evaluated with the current values of any referenced variables, and the cursor is reopened without requiring a CLOSE command. The cursor is positioned at the beginning of the result set when the FETCH command is run.

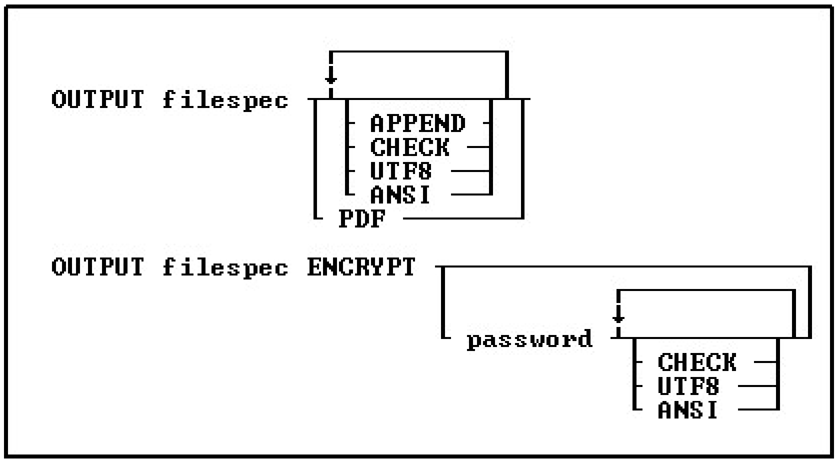
Example

The following command lines show the OPEN command with the RESET option.

```
   DROP CURSOR c1
   DROP CURSOR c2
   SET VAR vc1custid INTEGER
   DECLARE c1 CURSOR FOR SELECT custid FROM customer
-- Selects the transaction rows for the customer
   DECLARE c2 CURSOR FOR SELECT transid, invoicetotal +
      FROM transmaster WHERE custid = .vc1custid
-- Process the query in cursor c1 and get the first custid
   OPEN c1
   FETCH c1 INTO vc1custid IND c1ind1
   WHILE sqlcode <> 100 THEN
   -- Process the query in c2. As each row is fetched in the
   -- customer table, the custid changes; each time
   -- the "OPEN c2 RESET command" processes the c2's query it
   -- retrieves different rows
      OPEN c2 RESET
   -- Fetch the transid (invoicenumber) and invoice total amount
      FETCH c2 into vtransid, vamt
      WHILE sqlcode <> 100 THEN
         WRITE .vtransid, .vamt
         FETCH c2 into vtransid, vamt
      ENDWHILE
   FETCH c1 INTO vc1custid
   ENDWHILE
   CLOSE c1
   CLOSE c2
   DROP CURSOR c1
   DROP CURSOR c2
```

6.12.2 OUTPUT

Use the OUTPUT command to direct messages and results of commands to a file.



Options

ANSI
Converts all UTF8 characters to ANSI, to use the output in programs that do not understand UTF8 characters, but does handle the ANSI characters with code values above 127. ANSI and UTF8 would not be used simultaneously.

APPEND

Appends data to the end of an existing file without overwriting the file. If you specify APPEND when the specified file does not exist, R:BASE creates the file with that name. APPEND and CHECK would not be used simultaneously.

CHECK
Checks for file existence and prompts the user for confirmation before writing to it. APPEND and CHECK would not be used simultaneously.

ENCRYPT
A 512-bit encryption method is used to obscure any output information, making it unreadable without R:BASE and your decryption password. Immediately after using the ENCRYPT parameter in your OUTPUT command, you will be prompted for a password. The password is limited to 32 characters. When running encrypted files with R:BASE, you would run the file followed by the password. When opening an encrypted file, you will be prompted for the password.

filespec
Indicates the output device. Specify a file name, with or without an extension. You can also specify a drive and/or path.

password
Specifies the password for the encrypted file. If the password is not specified, a prompt to enter the password will be displayed.

PDF
Creates the output as a PDF file. The filespec should use the PDF file extension.

UTF8
Adds a UTF8 BOM to the front of the output file. ANSI and UTF8 would not be used simultaneously.

About the OUTPUT Command
The SET LINES command determines how many lines to display.

Examples

Example 01:
The OUTPUT command directs output to the BACKUP.DAT file on drive C:\. The UNLOAD command sends the data stored in the *transmaster* table as CSV to the file. The second OUTPUT command closes the file and redirects output to the screen.

```
OUTPUT c:\backup.dat
UNLOAD DATA FOR transmaster AS CSV
OUTPUT SCREEN
```

Example 02:
The commands send data from the Customer table to a PDF file, and open the file with the LAUNCH command.

```
OUTPUT CustomerList.PDF PDF
SELECT Company FROM Customer ORDER BY Company
OUTPUT SCREEN
LAUNCH CustomerList.PDF
```

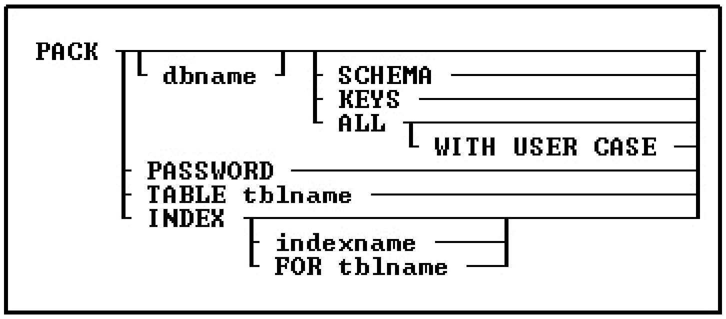
Example 03:
The commands unload the company data as ASCII into an encrypted file, and checks that the file does not already exist

```
OUTPUT COMPANY_DATA.DAT ENCRYPT pw1234 CHECK
UNLOAD DATA FOR Company AS ASCII
OUTPUT SCREEN
```

6.13 P

6.13.1 PACK

Use the PACK command on an open database to recover unusable disk space.



Options

ALL
Packs File 1 (schema information), File 2 (data), File 3 (indexes), and File 4 (large object data).

dbname
Specifies the name of the database to pack.

FOR tblname
Specifies a particular table whos indices you want to PACK.

INDEX
Use this option to PACK all indices for the currently connected database. PACK INDEX is supported in multi-user environments.

indexname
Packs a specified index from File 3 (indexes); this option will work when STATICDB is set on. PACK INDEX indexname is now supported in multi-user environments. This command will execute on the currently connected database.

KEYS
Packs only File 3 (indexes).

PASSWORD
Use this option to clean out bogus rows from SYS_PASSWORDS table.

PACK PASSWORD command is also supported in a multi-user session. Database must be connected in order to use this command.

SCHEMA
Packs only File 1 (schema information).

TABLE tblname
Packing a single table when MULTI is set ON. This parameter is very beneficial with databases that are always is use.

WITH USER CASE
Replaces the case folding/collating tables in the database with those defined in the user configuration file.

About the PACK Command

Disk space becomes unusable when you delete rows or indexes, remove columns or tables, or add or modify columns with the ALTER TABLE command. To use the PACK command, a database must be open. If the database you want to pack is not in the current directory, include the drive, path, and database name. When you pack a database that is in a different directory, R:BASE closes any open database, then opens the database you want to pack.

PACK requires the database owner's user identifier if the database is protected by the owner's user identifier.

You cannot use PACK when a database is stored on a network drive and MULTI has been set on, unless you are ONLY packing one table. Set MULTI off before packing the database.

PACK is unavailable when transaction processing is on.

Because you pack an open database, back up your database before you pack it. An interruption to a pack could cause damage to your database.

PACK Versus RELOAD
Both the PACK and RELOAD commands recover unusable disk space; however, RELOAD requires more disk space than PACK because RELOAD copies a database table by table, collects the rows of each table, then reorganizes the rows on the disk.

PACK KEYS versus PACK INDEX
PACK KEYS is to be used with MULTI set OFF as it recreates a new index file with clean indexes. PACK INDEX can be used while users are connected to the database with MULTI set ON, only adding to the current index file. After using PACK KEYS, you should see a decrease in the index file size.

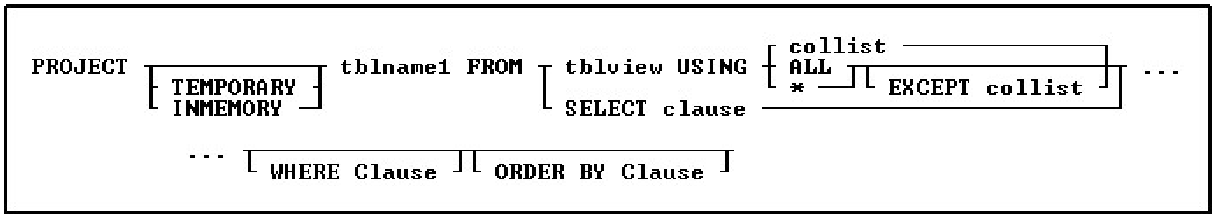
Example

The following command packs the *concomp* database in the RBTI directory on drive C:

```
PACK c:\rbti/concomp
```

6.13.2 PROJECT

Use the PROJECT command to create a new table from an existing table or view.



Options

*
Specifies to use all columns with the command.

ALL
Specifies to use all columns with the command.

collist
Specifies the column(s) to use with the command.

EXCEPT
Specifies the column(s) which will not be included in the projected table.

ORDER BY clause
Sorts rows of data. For more information, see ORDER BY.

SELECT clause
Specifies the columns and one or more tables or views from which to create the new table. Using the SELECT portion it is not necessary to create a view first in order to perform PROJECT into a new table

from multiple table joins. A USING clause is not needed as all required columns are defined in the SELECT statement.

tblname1 FROM tblview
Tblname1 is the name of the new table you want to create, and FROM *tblview* specifies the existing table or view you want to copy.

TEMPORARY
Allows you to create a Temporary Table with the PROJECT command.

WHERE clause
Limits rows of data. For more information, see WHERE.

About the PROJECT Command

The new table can be a copy of an existing table, a copy of an existing table with the rows sorted in a different order, a duplicate of a table structure without any data, or a table that contains specific rows and columns from an existing table.

You must include the USING clause with the PROJECT command. The USING clause specifies the columns to be included in the new table. If you want the new table to include all the columns from an existing table, use an asterisk (*) in the clause. If you want the new table to include only specific columns from the existing table, list them in the order you want them to appear in the new table. If you want the new table to include all columns in a different order, list them in the order you want them to appear.

Working with Computed Columns
R:BASE copies the data from each column into the new table. If a computed column is included, R:BASE transfers the current values in the computed column to the new table. In order to calculate computed values in the column in the new table, R:BASE needs the column names used in the computed column's expression. Therefore, include those column names in the USING clause before the computed column. When you do not include those column names in the USING clause before the computed column, R:BASE makes the computed column a regular column, assigns a data type, and displays a message suggesting you rename the column in the new table. If you do not rename the column, the new table has a column with the same name as the column in the original table, but does not have the designation COMPUTED. You will not be able to use the UNLOAD command, because you cannot have a computed column and a regular column with the same name.

Working with Autonumbered Columns
In a new table, R:BASE does not update the value in a row for the autonumber column. The autonumber column becomes a regular column.

Removing Columns and Rows from a Table
PROJECT is also useful if you want to remove several columns or rows from a table. To delete columns from a table, create a new table that retains the columns you want to keep, or to delete rows, create a new table using a WHERE clause. Use the DROP command to remove the table you no longer want, then use the RENAME command to give the new table the original table's name.

Transferring Default Column Definitions
Default column definitions are not transferred to a new table. When rows are added to the new table with the INSERT command, they are given a null value. If you want a default column definition, define the default column again.

Examples

The following command creates a new table that is a duplicate of the *employee* table.

```
PROJECT reps FROM employee USING *
```

The following command creates a table named *empty* that has the same structure as the *prodlocation* table but contains no rows of data.

```
PROJECT empty FROM prodlocation USING * WHERE LIMIT=0
```

Copyright © 1982-2025 R:BASE Technologies, Inc.

The example below creates a table named *gt5year*. The order of the columns in the *gt5year* table are specified in the USING clause. The WHERE clause specifies that only the information for employees hired before January 1, 1984 will be selected. The ORDER BY clause sorts the rows in alphabetical order by the employees' last names.

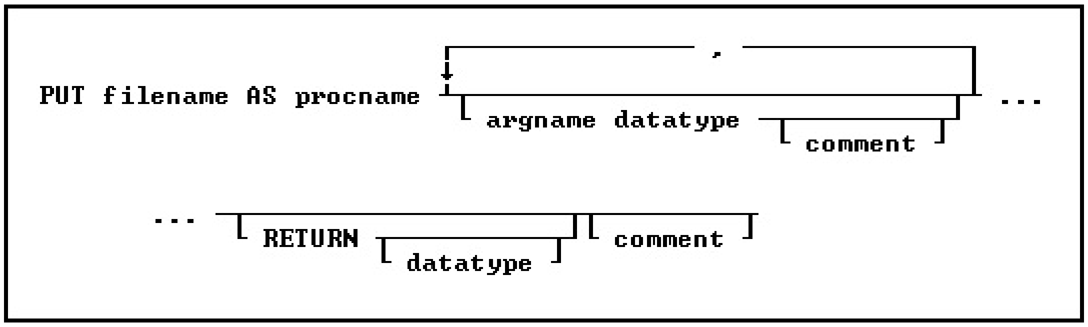
```
PROJECT gt5year FROM employee USING empfname, emplname, +
empid, empext, hiredate WHERE hiredate < '01/01/84' +
ORDER BY emplname
```

The following create a new table from the *Staff* and *Departments* table, with the SELECT statement to specify the column and tables source.

```
PROJECT StaffDepts FROM +
SELECT T2.DepartmentID,T2.Description,T2.OwnerDept,+
T1.LastName,T1.FirstName,T1.PhoneExt,T2.DeptShape +
FROM Staff T1,Departments T2 +
WHERE T1.DepartmentID = T2.DepartmentID
```

6.13.3 PUT

Creates or replaces Stored Procedures into the database.



Options

argname datatype
The parameter name and data type. This portion may be repeated.

comment
An optional comment for the parameter or, if placed after RETURN, an optional comment for the entire procedure. The comment must be enclosed in the current quote setting.

filename
The filename in ASCII text format, with full path, to load as the Stored Procedure.

procname
Specifies the procedure name. If a procedure by this name already exists in the database, an error is generated. The procedure name is limited to 128 characters.

RETURN datatype
Determines the data type of the value returned by the procedure.

About the PUT Command

Argument List
When you load a Stored Procedure into a database, you specify arguments to be passed to it. These arguments are used within the procedure. When the procedure is called, the number and type of arguments passed must match the number and type specified when the procedure was stored in the database. When an argument name is referenced in the Stored Procedure code, the argument name must be preceded by a period unless it is a table or column name, then it must be preceded by an ampersand (&). For example:

```
        UPDATE &p1 SET col  = 99 WHERE col = .p2
```

The arguments names are specified when the procedure is stored in the database with the PUT command.

Return Values
The value to be returned by a Stored Procedure is specified in the procedure code following the keyword RETURN. For example, RETURN 'Los Angeles'. The value returned must match the data type specified when the procedure was stored.

Replacing a Procedure
If you are replacing an existing procedure, you must LOCK the procedure first either with the GET LOCK or the SET PROCEDURE command. Once the procedure is locked, it is replaced by an updated file using the PUT command. A procedure cannot be replaced unless it is locked. A procedure is automatically unlocked when replaced with the PUT command.

Example

Use the PUT command as follows to store a command file as a Stored Procedure:

```
PUT INS.RMD AS SP_ContCheck p1 INT, p2 TEXT RETURN INTEGER
```

The contents of INS.RMD could be something like:

```
--INS.RMD
IF (.p1 > 105) THEN
  INSERT INTO contact (custid, contlname) VALUES (.p1, .p2)
  RETURN 1
ELSE
  RETURN 0
ENDIF
```

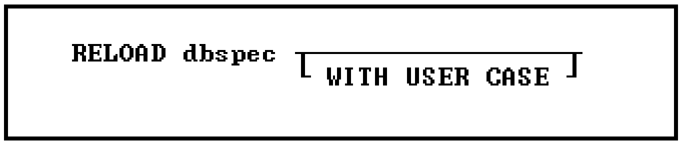
See Also:

Stored Procedures & Triggers

6.14 R

6.14.1 RELOAD

Use the RELOAD command to copy an open database without copying any unusable space.



Options

dbspec
Specifies the new database name.

WITH USER CASE
Incorporates case folding and collating tables defined in the user's configuration file into the reloaded database.

About the RELOAD Command

Disk space becomes unavailable in a database when the following actions are performed:

Copyright © 1982-2025 R:BASE Technologies, Inc.

- Deleting rows or indexes.
- Removing tables or columns.
- Adding columns or modifying tables with the ALTER TABLE command.

RELOAD copies a database table by table, and places all rows for each table in a single area on the disk, which improves database-response time.

When you use RELOAD to reload a database on the same disk and directory as the original database, enter a different name for the new database. When you reload a database from a different disk or directory onto the current disk, you can use the same database name for the copy. Be sure to specify the new drive or directory when you enter the command.

If there is not enough available disk space to copy a database using the RELOAD command, use the PACK command instead. Back up the database before packing. PACK eliminates unused space in a database; however, PACK does not rearrange the rows-only the RELOAD command rearranges rows.

RELOAD is available when MULTI is set on and a user has not set any locks on the database.

RELOAD is unavailable when RELOAD is unavailable when is on.

When you reload data that has a NOTE data type, the rows are adjusted according to the current setting of the SET NOTE_PAD command.

RELOAD is unavailable when STATICDB is set on, which activates a read-only schema mode.

Database Access Rights with RELOAD

When access rights for a table have been assigned using the GRANT command, RELOAD requires the database owner's user identifier to RELOAD a database.

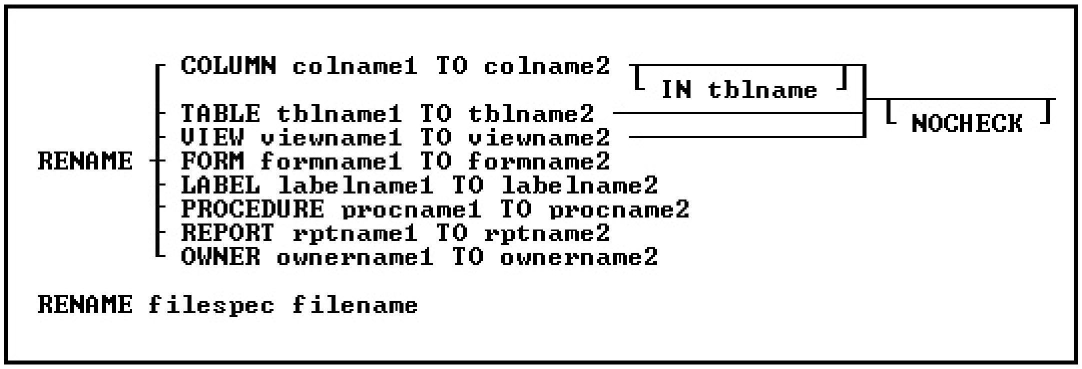
Example

The following command reloads an open database and gives it the name *newbase* in the RBASE directory of drive C:.

```
RELOAD c:\rbase/newbase
```

6.14.2 RENAME

Use the RENAME command to change a form, report, label, table, view, or column name, and the database owner's user identifier. You can also use RENAME to change the name of an existing file.



```
         COLUMN colname1 TO colname2   [ IN tblname ]
         TABLE tblname1 TO tblname2                     [ NOCHECK ]
         VIEW viewname1 TO viewname2
RENAME   FORM formname1 TO formname2
         LABEL labelname1 TO labelname2
         PROCEDURE procname1 TO procname2
         REPORT rptname1 TO rptname2
         OWNER ownername1 TO ownername2

RENAME filespec filename
```

Options

COLUMN colname1 TO colname2
Renames a column in one table or in all tables in the open database.

filename
Specifies the new name of the file.

filespec
Specifies the file you want to rename. Optionally, include a drive and path specification in the form D:\PATHNAME/FILENAME.EXT.

FORM formname1 TO formname2
Renames a form in the open database.

IN tblname
Specifies the table in which you want to rename a column.

LABEL labelname1 TO labelname2
Renames a label in the open database.

NOCHECK
Does not update references to views, tables, and columns in forms, reports, labels, access rights, and rules.

OWNER ownername1 TO ownername2
Renames an owner in the open database.

PROCEDURE procname1 TO procname2
Renames a Stored Procedure in the open database.

REPORT rptname1 TO rptname2
Renames a report in the open database.

TABLE tblname1 TO tblname2
Renames a table in the open database.

VIEW viewname1 TO viewname2
Renames a view in the open database.

About the RENAME Command

If you do not want R:BASE to update references to views, tables, and columns when you rename them, include the NOCHECK option with the command.

Renaming Columns
You can rename a column in an entire database or in a single table. R:BASE does not update column references in rules. When you rename a column, R:BASE automatically updates references to the column in the following instances:

- If the column has a description.
- If it is used in a form, report, label, computed column, UPDATE access right, or autonumbered column. However, because of possible size problems, R:BASE does not change column references inside an expression in a form, report, or label. These column references must be modified manually through the Form, Report, or Label Designer.

Updating Views and Tables
R:BASE automatically updates references to views and tables in the following instances:

- If you rename a view used in a report, label, or access right.
- If you rename a table that has a description, or is used in a form, report, label, or access right.

R:BASE does not update table or column references in views.

To update a view, delete it with the DROP command and define it again with the CREATE VIEW command, or QBE. To update a rule, you can use the RULES command, or the **Database Designer**. If you use the RULES command, you must first delete the rule with the DROP command and then add it again with the RULES command.

Renaming Tables
When you rename a table that is used as the rule table in a data-entry rule, R:BASE updates the rule definition. However, if you rename a table used in the WHERE clause of a rule definition, you must update the rule yourself. R:BASE does not update table references in views.

Renaming Files
When you use RENAME to change the name of a disk file, only the name of the file is changed. The file remains in the same directory on the same drive. You can include a file specification for the file you are renaming but not for the new file name. If you want the file to reside in a different drive or directory, use the COPY command. This command is similar to the operating system command RENAME.

On a workstation with multiple drives (local or mapped), especially when the files are on the different drive, it is always the best practice to define a drive letter when copying, deleting, renaming or running files, unless the specified files are located in the working directory. You will not need to specify the drive letter if all of the files are located in the default directory when using the copy, delete, rename or run commands.

Updating Command Files
R:BASE also does not update column, table, view, form, report, or label references in command files or applications. To update command files, use RBEdit or another text editor. To update applications, use the Application Designer.

Assigning User Identifiers
You can assign or change the database owner's user identifier with RENAME. The default user identifier is PUBLIC. Until this default is changed, any user can modify the database structure, read, enter, change, or delete data.

An owner's user identifer can be a maximum of eighteen characters. It must begin with a letter, and can contain letters, numbers, and the symbols #, $, _, and %, and must be unique among all user identifiers.

Examples

The following command renames a column from *transid* to *transxno* in the *transmaster* table.

```
RENAME COLUMN transid TO transxno IN transmaster
```

The following command changes the database owner's user identifier from the default PUBLIC to *june*.

```
RENAME OWNER PUBLIC TO june
```

The following command changes the name of the CUSTOMER file to CUSTOMER.DAT.

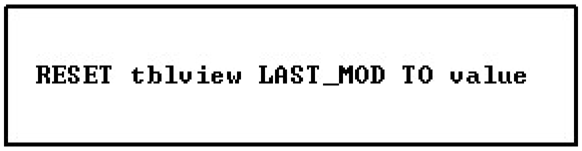
```
RENAME customer customer.dat
```

The following command renames all four database files to NYC.RX1, NYC.RX2, NYC.RX3, and NYC.RX4.

```
RENAME newyork.rx? nyc.rx?
```

6.14.3 RESET

The RESET command adjusts the last modification date/time stamp for tables and views.



```
RESET tblview LAST_MOD TO value
```

When copying table and view information with the UNLOAD command, the output automatically generates RESET commands after the table/view definition, to reset the date/time stamp and retain the original value.

tblview
Specifies a table or view.

value
Specifies the date/time stamp for tables or views.

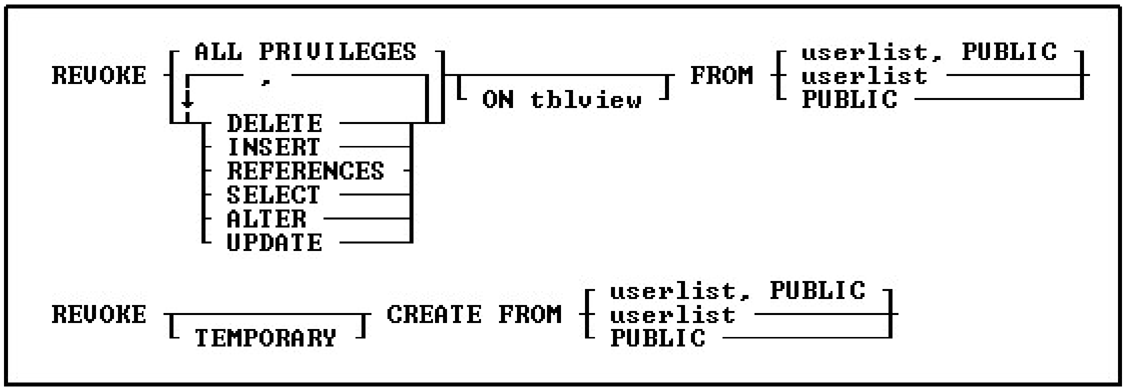
Examples

The below resets the last modification date and time stamp for the *LicenseInformation* table.
```
RESET LicenseInformation LAST_MOD TO '12/15/2022 12:00'
```

The below resets the last modification date and time stamp for the *SalesDataByCompany* view.
```
RESET SalesDataByCompany LAST_MOD TO '01/01/2022 08:00'
```

6.14.4 REVOKE

Use the REVOKE command to remove privileges provided to users with the GRANT command.



Options

,
Indicates that this part of the command is repeatable.

ALL PRIVILEGES
Removes all user privileges granted for all tables and views or for one table or view.

ALTER
Removes permission from users to modify the structure of all tables or specified tables.

CREATE
Removes permission from users to create new tables. Do not specify any tables or views when removing this permission.

DELETE
Removes permission to remove rows from all tables and views, or from a specified table or view.

FROM PUBLIC
Specifies PUBLIC. If, for example, *Ralph*, *Sam*, *Jane*, and PUBLIC have been granted certain user privileges, revoking privileges from PUBLIC would not affect the three listed users.

FROM userlist
Specifies individual users whose access is to be revoked. You must separate user identifiers with a comma (or the current delimiter). For a value with spaces, the userid must be enclosed in quotes.

FROM userlist, PUBLIC
Specifies both individual users and PUBLIC, whose access is to be revoked. You must separate user identifiers with a comma (or the current delimiter). For a value with spaces, the userid must be enclosed in quotes.

INSERT

Removes permission to add rows to all tables and views or to a specified table or view.

ON tblview
Specifies a table or view from which to remove user privileges.

REFERENCES
Removes permission to create a table with a foreign key that references a table with a primary key.

SELECT
Removes permission to view and print data from all tables and views, or from a specified table or view.

TEMPORARY
Removes permission from users to create new temporary tables.

UPDATE
Removes permission to change the value of all columns on all tables and views, or on a specified table or view. You cannot specify columns when revoking UPDATE permission.

About the REVOKE Command

If you are the owner of a database, you can revoke any user privileges granted to users. If the database owner or other users have assigned you user privileges with the WITH GRANT OPTION, you can revoke only the user privileges that you have granted to other users.

To remove the WITH GRANT OPTION, you must first revoke the privilege(s) to which the WITH GRANT OPTION has been assigned.

REVOKE ALL PRIVILEGES revokes all user privileges that have been granted. However, REVOKE combined with ALTER, CREATE, DELETE, INSERT, REFERENCES, SELECT or UPDATE only applies to those privileges.

You can remove more than one user privilege in a REVOKE command. Separate the user privileges with a comma (or the current delimiter).

Examples

Assume that the following sequence of GRANT commands represents all the user privileges granted for the *ConComp* database.

```
GRANT INSERT ON Employee TO Ralph, Sam
GRANT SELECT, INSERT ON TransMaster TO Jane WITH GRANT OPTION
GRANT UPDATE ON TransMaster TO Sam
GRANT UPDATE (Company, CustAddress, CustCity) ON Customer TO Sam, PUBLIC
```

The following command revokes permission granted to *Jane* to display or print data, or add rows to the *TransMaster* table.

```
REVOKE SELECT, INSERT ON TransMaster FROM Jane
```

The following command revokes the UPDATE user privilege granted to *Sam* for all tables and views in the database.

```
REVOKE UPDATE FROM Sam
```

The following command revokes all user privileges granted to *Sam*, except those granted to him as a member of PUBLIC.

```
REVOKE ALL PRIVILEGES FROM Sam
```

The following command revokes all user privileges for all tables and views for the users Sam, Jane, and Ralph; and the PUBLIC account.

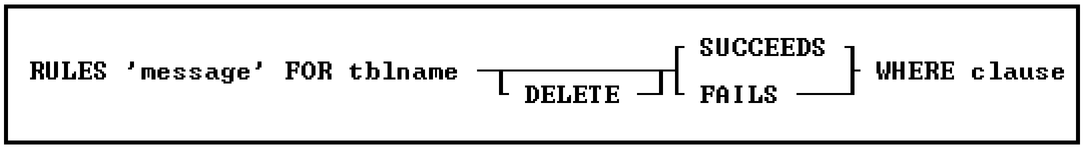
```
REVOKE ALL PRIVILEGES FROM Sam, Jane, Ralph, PUBLIC
```

The following command revokes CREATE privileges for temporary tables for the user.

```
REVOKE TEMPORARY CREATE FROM Noah
```

6.14.5 RULES

Use the RULES command to regulate data entry in a database.



Options

DELETE

- **DELETE SUCCEEDS** deletes a row from a database when the conditions in the WHERE clause are met.
- **DELETE FAILS** deletes a row from a database when the conditions in the WHERE clause are not met.

FOR tblname
Specifies the name of the table for which you are defining rules.

FAILS
Specifies that a row must not meet the conditions included in the WHERE clause in order to be added to the database.

'message'
Specifies a message to be displayed when a rule is violated.

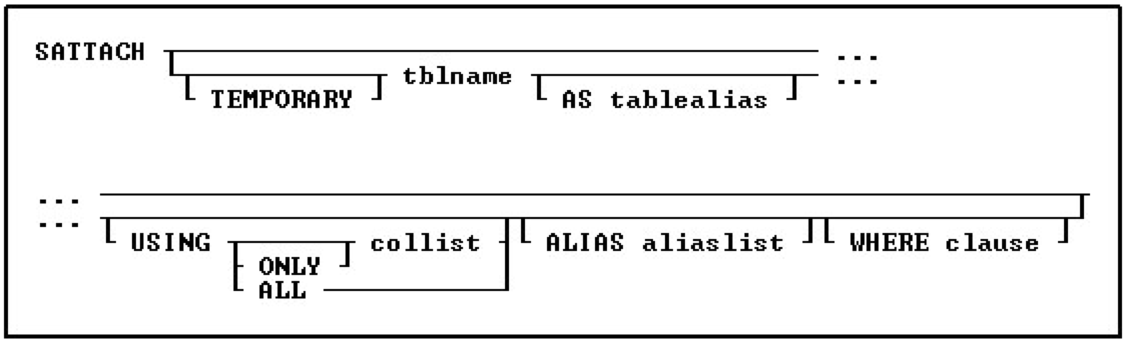
SUCCEEDS
Specifies that a row must meet the conditions included in the WHERE clause in order to be added to the database.

WHERE clause
Limits rows of data. For more information, see the WHERE command.

6.15 S

6.15.1 SATTACH

Use the SATTACH command to attach a specified table from a foreign database to a connected R:BASE database.



Options

ALIAS AliasList
To specify alias names for columns. The alias list is separated by commas. Only the changed column names can be specified in the alias list, with other column name to be retained as is left blank.

AS tablealias
Specifies an alias, or temporary name, for the foreign table. A table alias is sometimes required when attaching foreign data sources that do not follow the same table name restrictions as R:BASE.

tblname
Specifies the table in the foreign database to attach.

TEMPORARY
Allows you to create a temporary name with the SATTACH command. The temporary tables will disappear when the database is disconnected. **NOTE:** Any changes made to the temporary table will not be reflected upon the original SQL data source.

USING ALL
Specifies all columns that uniquely identify the rows in an attached table if no primary or unique keys are defined. With USING ALL, the QUALCOLS setting is ignored to determine the number of columns to identify the rows.

When performing a direct UPDATE to the foreign table, the USING ALL approach is the slowest in processing, which means that to qualify a row for updating, all of the column values must match. Rather, if a primary key exists, specify the column with USING *PrimaryKeyColName* instead, as this way, only the primary key value must match (which is all that should be needed). When updating a row on the foreign table, R:BASE must count how many rows match that row, and there should only be one matching row.

USING collist
If the foreign table has no primary or unique key, specify the column(s) that uniquely identify the rows in the table. The primary key should be specified as the *collist* value. The *collist* is not limited to a single column, but the more columns that are specified, then those column values must also match.

USING ONLY collist
Specifies that only the columns listed will be attached. When using the ONLY option the word "ONLY" must immediately follow the word "USING". R:BASE will determine the key columns by querying the special columns of source table, to find primary key or unique key columns. If the query fails, then all columns will be used.

WHERE clause
Limits rows of data. For more information, see the WHERE Clause.

About the SATTACH Command

Before you can attach a foreign data source table, an R:BASE database must be connected. Also, your workstation must be connected to the data source.

If you use SATTACH without the *tblname* option, R:BASE displays the "**Attach Table(s)**" dialog box with names of tables in the data source. You can then select a table to attach.

When attaching external tables by selecting "Utilities" > "Attach SQL Database Tables" from the menu bar, or using the SATTACH command (without the "USING ALL/collist" keywords), the QUALOCOLS setting is used to assign what columns uniquely identifies a row. If a primary key or unique key was not found for the table being SATTACHed, and the USING collist clause was not used to specify what columns uniquely identifies a row, then R:BASE assigns primary and unique key qualkeys for the attached table. R:BASE assigns a set of columns to identify the rows starting with the first column of the table. The number of columns used is limited by the value for QUALCOLS. The (CVAL('QUALKEYS')) function may be used to capture the columns assigned as a QualKeys for the current database. The (CVAL('QUALKEY TABLES')) function may be used to capture the tables assigned with QualKey columns.

After you have exited R:BASE or disconnected the database to which the foreign table is attached, you do not need to reconnect to the table's data source when you open the database again. The data source is connected when you use the attached table. The data source table remains attached until you detach it with the SDETACH command, or use the **Utilities: Detach SQL Database Table** menu option.

When a foreign table is attached, R:BASE writes a table description that identifies the table as a data source table and names its data source. Use the LIST TABLES *tblname* command to review table descriptions.

Notes:

- SATTACH requires an owner password if one has been defined, or permission to create tables.
- When you attach a foreign table, R:BASE only includes the columns with legal names. For example, R:BASE does not include columns that have spaces in the name, or column names that exceed the character limit for the R:BASE version installed. Table and column names are limited to 128 characters.
- When you attach a foreign table and select the columns that uniquely identify its rows in the "Select Column Set" dialog box, do not select columns that have LOB data types, as unpredictable results might occur.
- When running applications that connect to foreign data sources, you should always disconnect from the R:BASE database before running the application again.
- When using the keyword "ONLY", to limit the columns attached with a table, the ability to SATTACH temporary tables and column alias names is supported.

Notes for [Alias] Parameters:

- Syntax has been extended to specify only the changed columns. For example, if you only need to alias the second column out of four columns you can use *... ALIAS ,,location,,*
- Any missing alias names will use the default name.
- If there is a conflicting column name, a warning will be displayed you will be prompted for a new column name.
- If the name conflicts with another name then you get the error message first explaining the conflict, then the dialog box.
- If no qualkey is specified, automatic qualkeys will be assigned based on information from the ODBC source

Examples:

Example 01:
Attaches a foreign data source table using an alias table name

```
SATTACH CustomerDetails AS tCustomerDetails
```

Example 02:
Attaches a foreign data source table using an alias table name, and specifies the unique column name for the source table

```
SATTACH CustomerDetails AS tCustomerDetails USING CustomerID
```

Example 03:
Attaches a foreign data source table using alias names for the table and columns, and specifies the unique column name for the source table

```
SATTACH Orders AS tOrders USING +
OrderID ALIAS +
OrderID, +
CustomerID, +
EmployeeID, +
OrderDate, +
RequiredDate, +
ShippedDate, +
ShipVia, +
tFreight, +
ShipName, +
ShipAddress, +
ShipCity, +
ShipRegion, +
ShipPostalCode, +
ShipCountry
```

Example 04:
Attaches a foreign data source table using alias names for the table and only the Freight column, and specifies the unique column name for the source table

```
SATTACH Orders AS tOrders USING +
OrderID ALIAS ,,,,,,,,tFreight,,,,,,
```

Example 05:
Attaches a foreign data source table using an alias name for the table which contains spaces, and specifies only two columns to be included. The table `Order Details` is surrounded by IDQUOTES.

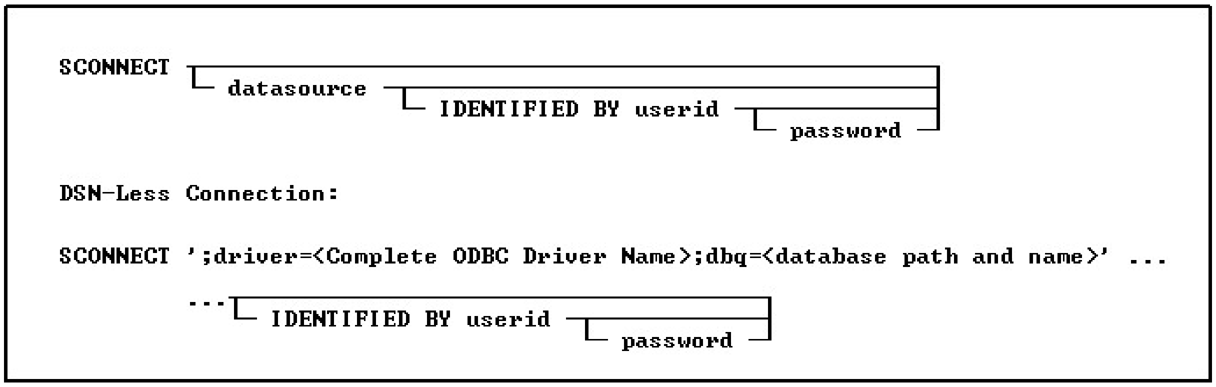
```
SATTACH `Order Details` AS tOrderDetails USING ONLY +
OrderID, ProductID
```

Example 06:
Attaches a foreign data source table using an alias name for the table, and specifies a WHERE clause for limited results

```
SATTACH Artists AS tArtists WHERE ALastName = 'Ford'
```

6.15.2 SCONNECT

Use the SCONNECT command to connect R:BASE to a foreign data source.



Options

datasource
Specifies the name of the data source name (DSN) that contains the table to access.

IDENTIFIED BY userid
Specifies the user account name for the data source. You can use '' (two single quotes) in this position if there is no User ID.

password
Specifies the password for the data source. You can use '' (two single quotes) in this position if there is no password.

About the SCONNECT Command
To access an R:BASE database, the R:BASE or Oterro ODBC driver must be installed. The R:BASE/Oterro ODBC driver version must match the version of the database files that are being connected.

Omitting the *datasource* option will display a dialog box, listing data sources from which to choose, and continues with prompts for a *userid* and *password*.

Use the SATTACH command to attach tables to a database. Added data source tables, or SERVER tables, within an R:BASE database will remain defined as part of the database structure, and will appear in the Database Explorer table list. To disconnect a DSN connection, use the same syntax with the SDISCONNECT command.

About the DSN-Less Connection
A data source name (DSN) is a data structure that contains the information about a specific database that an Open Database Connectivity (ODBC) driver needs in order to connect to it. Included in the DSN, which resides either in the registry or as a separate text file, is information such as the name, directory and driver of the database, and, depending on the type of DSN, the ID and password of the user. The developer creates a separate DSN for each database. To connect to a particular database, the developer specifies its DSN within a program. In contrast, DSN-less connections require that all the necessary information be specified within the command. DSN-Less connection requires no server setup, just a carefully constructed connection string.

There are three kinds of DSN: user DSNs (sometimes called machine DSNs); system DSNs; and file DSNs. User and system DSNs are specific to a particular computer, and store DSN information in the registry. A user DSN allows database access for a single user on a single computer, and a system DSN for any user of a particular computer. A file DSN contains the relevant information within a text file with a .DSN file extension, and can be shared by users of different computers who have the same drivers installed.

DSN-less connections demand that you know the name of the file (e.g. file based databases like R:BASE) or the address of the data server (SQL Server for example).

Armed with appropriate information you could open a data source without a DSN. Normally on the SCONNECT command you specify the DSN you want to use. ODBC looks up this DSN and determines the driver to use and what connection it needs. With this method, R:BASE allows the ability to specify the ODBC driver instead of a DSN.

About the Connection String
The first character in the connection string is a semi-colon. This is the flag which states the string is not specifying a DSN.

The value after "driver" is the actual name of the driver as defined in the ODBC Data Source Administrator. Make sure to spell and space the driver name exactly as it is defined in ODBC Administrator interface under the "Drivers" tab.

The database name and exact path is specified in the "dbq" portion of the string.

The entire string must be surrounded by the database QUOTE character. Single quotes are used in the examples below.

Examples

Example 01 (Using R:BASE 11 ODBC Driver):

```
SCONNECT ';driver=R:BASE 11 Database Driver (*.RX1);dbq=d:\SampleData\RRBYW20'
```

Example 02 (Using Oterro 11 ODBC Driver):

```
SCONNECT ';driver=Oterro 11 Database Driver (*.RX1);dbq=d:\SampleData\RRBYW20'
```

Example 03 (To use an Access database in "dsnless" mode):
Uses a database called db1.mdb in the "My Documents" folder. The connection can specify additional items like user id with the "UID=" parameter and a password with the "PWD=" parameter.

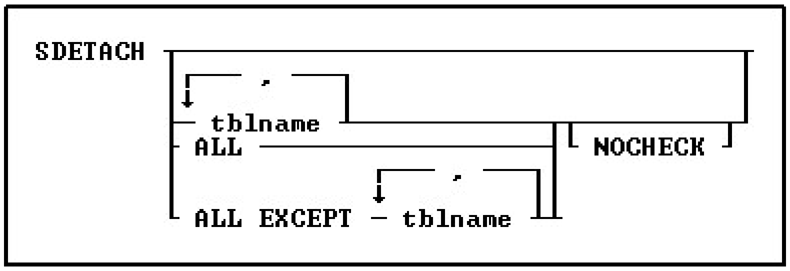
```
SCONNECT ';Driver={Microsoft Access Driver (*.mdb)};DBQ=c:\Documents and
Settings\Administrator\My Documents\db1.mdb;'
```

Example 04 (To use a SQL Server database):
The server and database are specified along with the uid and pwd.

```
SCONNECT ';driver={SQL
Server};server=corpseadb0d;uid=my_user_name;pwd=my_pw;database=JohnDoe;'
```

6.15.3 SDETACH

Use the SDETACH command to remove a foreign data source table from a connected R:BASE database.



Options

,
Indicates that this part of the command is repeatable.

ALL
Specifies all tables.

ALL EXCEPT tblname
Specifies all tables except those specified.

NOCHECK
Eliminates the confirmation message.

tblname
Specifies the table to detach.

For information on how to attach foreign data source tables, see SATTACH.

Examples:

Example 01:
Detaches the tOrderDetails foreign table, without displaying the confirmation dialog

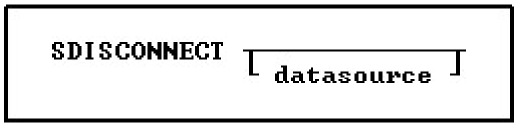
```
SDETACH tOrderDetails NOCHECK
```

Example 02:
Detaches all foreign data source tables except ServerFreight and ServerShipping

```
SDETACH ALL EXCEPT ServerFreight, ServerShipping
```

6.15.4 SDISCONNECT

Use the SDISCONNECT command to disconnect a foreign data source from an R:BASE database.



Options

datasource
Specifies the data source to disconnect.

About the SDISCONNECT Command

If you omit the *datasource* option, a dialog box opens listing data sources from which to choose.

To disconnect the DSN-less connection, use the exact same data source syntax as the SCONNECT command, only with SDISCONNECT.

Examples:

Example 01:

```
SDISCONNECT ';driver=R:BASE 11 Database Driver (*.RX1);dbq=d:\SampleData\RRBYW20'
```

Example 02:

```
SDISCONNECT ';driver=Oterro 11 Database Driver (*.RX1);dbq=d:\SampleData\RRBYW20'
```

Example 03:

```
SDISCONNECT ';Driver={Microsoft Access Driver (*.mdb)};DBQ=c:\Documents and
Settings\Administrator\My Documents\db1.mdb;'
```

Example 04:

```
SDISCONNECT
';driver={SQLServer};server=corpseadb0d;uid=my_user_name;pwd=my_pw;database=JohnDoe;'
```

See also: DSN-less connections

6.15.5 SELECT

Use the SELECT command to display rows of data from a table or view. To display the data in the order you want, modify the SELECT command by using various clauses.

The SELECT command is a very powerful data retrieval command. By learning this command, and all of its parts you can greatly enhance your ability to work with any other R:BASE command that uses those same portions. For example, learning to use a WHERE clause with SELECT will help you work with WHERE clauses on other commands.

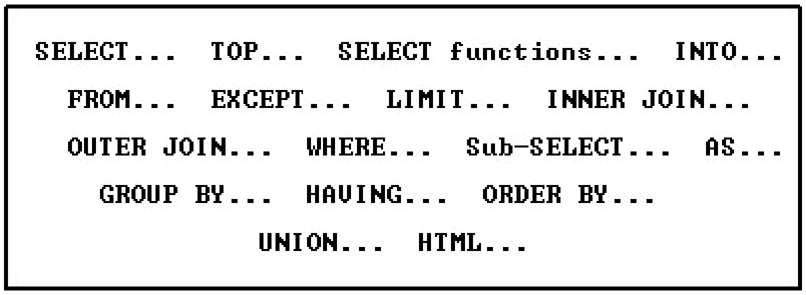
You can use the SELECT command to do the following:

- Display rows of information from a table or view
- Extract information from a table or view by using a sub-SELECT command (a nested SELECT statement) in a WHERE command
- Extract information from a table or view by using a SELECT clause in another command

A SELECT command is essentially a process of elimination. A SELECT command can contain a number of clauses (two are required), each of which begins with a keyword, such as FROM or WHERE.

The diagram below shows the different clauses in a SELECT command.

Copyright © 1982-2025 R:BASE Technologies, Inc.


SELECT... TOP... SELECT functions... INTO...
FROM... EXCEPT... LIMIT... INNER JOIN...
OUTER JOIN... WHERE... Sub-SELECT... AS...
GROUP BY... HAVING... ORDER BY...
UNION... HTML...

Each of the SELECT clauses has a specific purpose for determining what data you want. The operators are processed in the order in which they appear in the preceding diagram.

N**ote:**

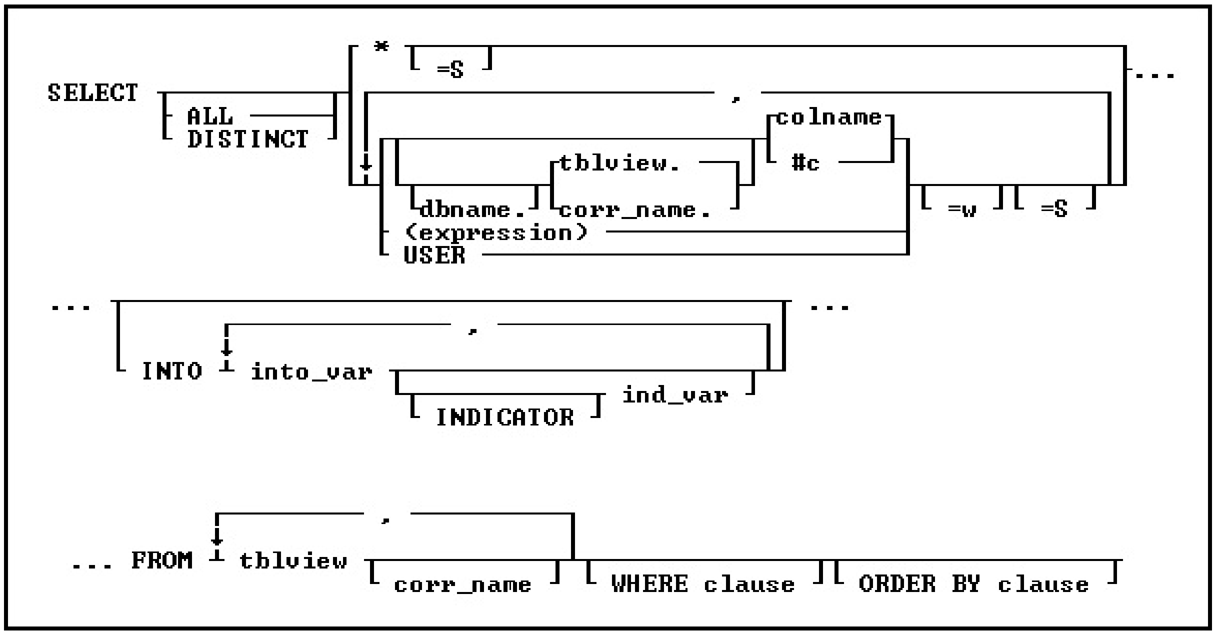
- Many of the SELECT clauses use the same options, such as *expression* or *colname*. These common options are described only once in "SELECT Command Clause" below.

SELECT Command Clause

The required SELECT command clause specifies which columns to include. You can:

- Select all columns by entering SELECT with an asterisk.
- Name the columns you want to select.
- Use expressions and SELECT functions to perform calculations whose results will also appear as a column in the final result.
- Select the column or expression values and load them into variables.

Syntax:



Options

*
Specifies all columns.

,
Indicates that this part of the command is repeatable.

ALL
Specifies all rows returned by the other clauses.

#c
Specifies a column, where *#c* is the column number shown in the output of the LIST TABLES command. You can enter a table or correlation name before the *#c*.

colname
Specifies a column name. In a command, you can enter *#c*, where *#c* is the column number shown when the columns are listed with the LIST TABLES command. In an SQL command, a column name can be preceded by a table or correlation name and a period (*tblname.colname*). You can enter *tblname.** to specify all columns in the table.

corr_name
Correlation name. A nickname or alias for a table or view name. Use *corr_name* to refer to the same table twice within the command, or to explicitly specify a column in more than one table. The correlation name must be at least two characters.

dbname
Currently connected database name, plus the drive and directory if the database is not on the current directory. It has the form D:\PATHNAME/DBNAME where D: is the optional drive letter, /PATHNAME is the optional directory path, and /DBNAME is the database name.

DISTINCT
Eliminates duplicate rows from the resulting data set.

(expression)
Determines a value using a text or arithmetic formula. The expression can include other columns from the table, constant values, functions, or system variables such as *#date*, *#time*, and *#pi*.

FROM
Lists the tables from which data is to be displayed.

ind_var
Specifies a variable result indicator to be used with an INTO clause in a SELECT command. This variable stores the status of the variable: non-null (0) or null (-1).

INDICATOR
Indicates the following variable is an indicator variable, which is used to indicate if a null value is retrieved.

INTO
Selects information directly from a table and puts it into variables. You must include a WHERE clause so the SELECT command finds only one row.

into_var
Specifies a variable whose value is assigned with an INTO clause in a SELECT command.

ORDER BY clause
Sorts rows of data. See ORDER BY.

=S
Calculates the sum of a column that has CURRENCY, DOUBLE, INTEGER, NUMERIC, or REAL data type values, or the results of an expression using CURRENCY, DOUBLE, INTEGER, NUMERIC, or REAL data type values.

tblview
Specifies a table or view name.

USER
Retrieves the current user as a constant.

=w
Specifies a display width.

Copyright © 1982-2025 R:BASE Technologies, Inc.

WHERE clause
Limits rows of data. See "WHERE.

Examples

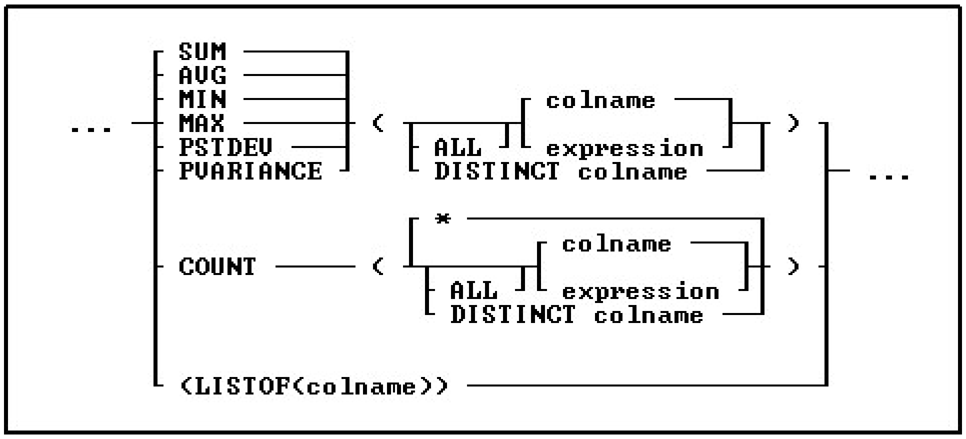
The following command selects the company name and ID for companies in Washington state:

```
SELECT custid, company FROM customer +
WHERE custstate = 'WA' ORDER BY company
```

```
custid company
122    Data Solutions
119    Datacrafters Infosystems
130    MIS by Design
114    Softech Database Design
```

6.15.5.1 SELECT Functions

This clause, determines which columns to include.



Options

*****
Specifies all rows.

AVG
Computes the numeric average. R:BASE rounds averages of INTEGER values to the nearest integer value and CURRENCY values to their nearest unit.

COUNT
Determines how many non-null entries there are for a particular column item.

DISTINCT
Eliminates duplicate rows from the calculation. The DISTINCT keyword is only needed to be specified for the first column in order to be applied to all columns in a query.

LISTOF
Creates a text string of the values separated by the current comma delimiter character. The text string is limited to 10,000 characters. As LISTOF is an aggregate-style function, any sorting (ORDER BY) needed for the listed results must be performed at a lower level, in which a view may be created to perform the sorting first.

The LISTOF function can be used with the "SELECT ... INTO ..." to populate a variable with a list of values which can then be used in a CHOOSE command with the #LIST option. It can also be used in Forms, Reports or Labels to look up values from multiple rows.

MAX
Selects the maximum numeric, time, date, or alphabetic value in a column.

MIN
Selects the minimum numeric, time, date, or alphabetic value in a column.

PSTDEV
Calculates population standard deviation.

PVARIANCE
Determines population variance.

SUM
Computes the numeric sum.

About SELECT Functions

A SELECT function can be used to provide summary data about a group of rows in a table or for all rows in a table. These functions may only be used with the GROUP BY clause or when only SELECT functions are specified.

Using MIN and MAX functions for NOTE data type

Selecting aggregate functions, such as MIN and MAX, requires that R:BASE keeps an accumulator and choose to only use the first 80 characters for NOTE values. This matches the fact that if you sort on NOTE fields, the sort will be based on the first 80 characters only.

Examples

The COUNT function works in two different ways, depending on its argument. COUNT(*) counts all rows in a table, but (OUNT(*colname*) counts only rows in which the value in the specified column is not null. For example:

```
SELECT emptitle, COUNT(*), COUNT(emptitle) +
FROM employee GROUP BY emptitle
```

The final result contains both forms of the COUNT function.

```
emptitle          COUNT(*)          COUNT(emptitle)
Manager           2                 2
Representative    3                 3
Sales Clerk       1                 1
-0-               2                 0
```

If you wanted to compute the difference between each employee's average sales and the average sales for all employees, you would first use a SELECT function to calculate the average for all employees and store the result in a variable. For more information, see INTO.

The following command assigns the value $71,571.88 to the *vaverage* variable.

```
SELECT AVG(netamount) INTO vaverage FROM transmaster
```

Next, you can use the variable and another SELECT function in an expression to calculate the difference for each employee, and display the average net amount for each employee.

```
SELECT empid, AVG(netamount), +
   (.vaverage - (AVG(netamount)))=30 +
          FROM transmaster GROUP BY empid
```

```
empid  AVG (netamount)     (.vaverage - AVG(netamount))
102    $64,510.00          $7,061.88
129    $69,555.00          $2,016.88
```

```
131    $118,000.00         -$46,428.12
133    $44,380.00          $27,191.88
160    $114,850.00         -$43,278.12
165    $14,685.00          $56,886.88
167    $3,830.00           $67,741.88
```

Examples Using the LISTOF Function

```
SELECT (LISTOF(ColumnName)) INTO VariableName INDICATOR IndVar +
FROM TableView WHERE ...
```

In a Form, Report or Label Expression:

```
VariableName = (LISTOF(ColumnName)) IN LookUpTableView WHERE +
KeyColumn = KeyColumn
```

Example 01:

```
SET VAR vValueList TEXT = NULL
SELECT (LISTOF(ColumnName)) INTO vValueList INDIC IvValueList +
FROM TableName WHERE ...
```

The variable vValueList will be a text string of the values separated by the current comma delimiter character.

If you would like to add a single space after each value, then:

```
SET VAR vValueList TEXT = NULL
SELECT (SRPL(LISTOF(ColumnName),',',', ',0)) INTO +
vValueList INDIC IvValueList FROM TableName WHERE ...
```

Notice the additional space after comma in ReplaceString.

If you would like to use a carriage return after each value, then:

```
SET VAR vValueList TEXT = NULL
SELECT (SRPL(LISTOF(ColumnName),',',(CHAR(10)),0)) INTO +
vValueList INDIC IvValueList FROM TableName WHERE ...
```

Example 02:

```
CONNECT Concomp IDENTIFIED BY NONE
SET CAPTION ' '
SET AUTODROP OFF
SET RBGSIZE CENTER CENTER 800 600
SET VAR vLines INTEGER = 0
SET VAR vValueList TEXT = NULL
SET VAR vLastName TEXT = NULL
SET VAR vTitle TEXT = 'List Created Using LISTOF Function'
SET VAR vCaption TEXT = 'Using #LIST Options in CHOOSE Command!'
CLS
PAUSE 3 USING 'Collecting Values ...' CAPTION .vCaption AT 16 30
SELECT (COUNT(*)), (LISTOF(EmpLName)) INTO +
vLines INDIC IvLines, vValueList INDIC IvValueList FROM Employee
IF vLines > 18 THEN
   SET VAR vLines = 18
ENDIF
CLS
```

```
CHOOSE vLastname FROM #LIST .vValueList AT 6 30 +
TITLE .vTitle CAPTION .vCaption LINES .vLines FORMATTED
IF vLastName IS NULL OR vLastName = '[Esc]' THEN
   GOTO Done
ELSE
   CLEAR ALL VAR EXCEPT vLastName
ENDIF
-- Do what you have to do here ...
LABEL Done
CLS
CLEAR ALL VAR
QUIT TO MainMenu.RMD
RETURN
```

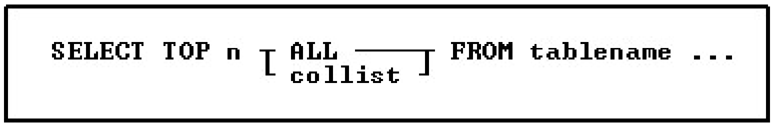
Example 03:

If you would like to retrieve the list of DISTINCT values, then:

```
SET VAR vValueList TEXT = NULL
SELECT (LISTOF(DISTINCT ColumnName)) INTO +
vValueList INDIC IvValueList FROM TableName WHERE ...
```

6.15.5.2 TOP

This parameter includes the support to specify the TOP n qualifier for the SELECT command. The TOP n will retrieve the top number of records from the table.



The "TOP n" goes between the word SELECT and the column list for the command.

n
Specifies the number of records to retrieve

collist
Specifies a list of one or more column names, separated by a comma (or the current delimiter)

Example:

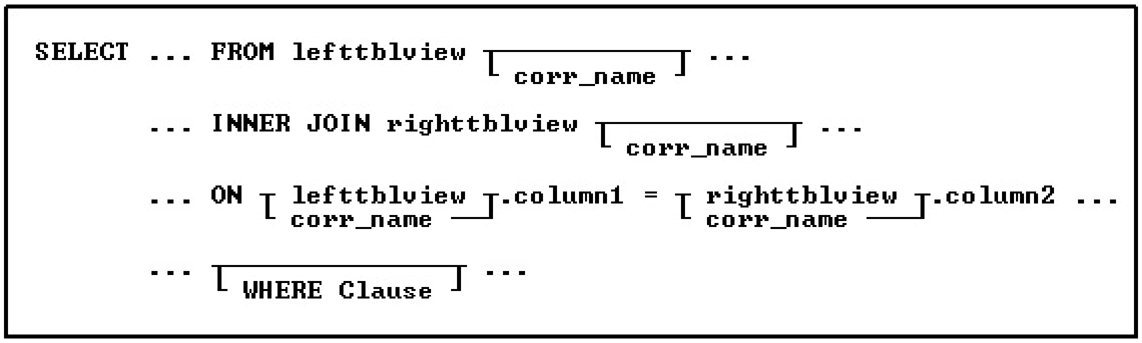
To show the top 5 bonuses where the bonus is under $500 from the SalesBonus table in the ConComp:

```
SELECT TOP 5 EmpID,Bonus FROM SalesBonus WHERE Bonus < 500 ORDER BY Bonus=DESC
```

```
 EmpID      Bonus
 ---------- ---------------
        131         $456.75
        131         $326.25
        102         $175.00
        131         $157.50
        129         $153.60
```

6.15.5.3 INNER JOIN

This clause is used to retrieve data from two or more tables.



```
SELECT ... FROM lefttblview [corr_name] ...
    ... INNER JOIN righttblview [corr_name] ...
    ... ON [lefttblview | corr_name].column1 = [righttblview | corr_name].column2 ...
    ... [WHERE Clause] ...
```

Options

.column1
Defines the column on which to link.

.column2
Defines the column on which to link.

corr_name
A correlation name is an alias or nickname for a table. It lets you refer to the same table twice in one command, use a shorter name, and explicitly refer to a column when referring to the same column if that column appears in more than one table. The correlation name must be at least two characters.

FROM lefttblview
Specifies the left table or view.

lefttblview
Explicitly defines the column on which to link the left table name or view.

INNER JOIN righttblview
Specifies the right table or view.

righttblview
Explicitly defines the column on which to link the right table name or view.

WHERE clause
Limits rows of data. See WHERE.

About JOIN

When you perform a SQL JOIN, you specify one column from each table to join on. These two columns contain data that is shared across both tables. You can use multiple joins in the same SQL statement to query data from as many tables as you like.

JOIN Types

Depending on your requirements, you can do an "**INNER**" join or an "OUTER" join. The differences are:

- **INNER JOIN**: This will only return rows when there is at least one row in both tables that match the join condition.
- **LEFT OUTER JOIN**: This will return rows that have data in the left table (left of the JOIN keyword), even if there's no matching rows in the right table.
- **RIGHT OUTER JOIN**: This will return rows that have data in the right table (right of the JOIN keyword), even if there's no matching rows in the left table.
- **FULL OUTER JOIN**: This will return all rows, as long as there's matching data in one of the tables.

Nested JOINs

Any of the JOIN types can be mixed in any sequence to create a nested join. The nested joins still require that you specify one column from each table to join on. When nesting joins, it is important to use the correct sequence of parenthesis, along with a correlation for each join.

In the example below, notice the two sets of parenthesis, which all begin after the FROM keyword and end after the linking columns. Also note the "J1" and "J2" correlations specified for each join.

```
SELECT ALL FROM (( TABLE1 t1 +
INNER JOIN TABLE2 t2 ON t1.FieldT2=t2.FieldT2) J1 +
INNER JOIN TABLE3 t3 ON t3.FieldT3=j1.FieldT3) J2
```

Examples

INNER JOIN Example:

The following example list an employee's total sales for each day.

```
SELECT t1.empid, t2.netamount, t2.transdate FROM employee t1 +
INNER JOIN transmaster t2 ON t1.empid = t2.empid +
WHERE empid = 129
```

```
 t1.empid   t2.netamount    t2.TransDate
 ---------- --------------- ------------
        129       $3,080.00 07/02/2003
        129       $5,385.00 07/08/2003
        129       $6,160.00 07/11/2003
        129       $5,575.00 08/24/2003
        129      $10,445.00 08/24/2003
        129      $10,175.00 08/25/2003
        129       $2,195.00 08/27/2003
```

Nested INNER JOIN Example:

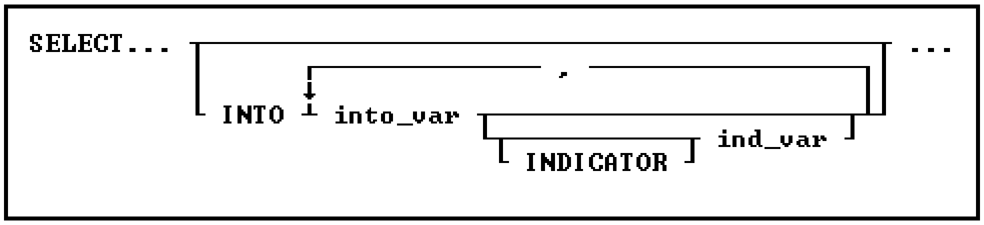
The following example lists a specific product, all of the locations where it resides, and the components used within the product.

```
SELECT ProdName, Location, CompID FROM  +
((Product t1 INNER JOIN ProdLocation t2 ON t1.Model=t2.Model) J1 +
 INNER JOIN CompUsed t3 ON t3.Model=j1.Model) J2 +
WHERE Model = 'CX3000'
```

```
 ProdName                           Location CompID
 ---------------------------------- -------- --------
 Standard SVGA Color PC             A-1      X1010
 Standard SVGA Color PC             A-1      X2000
 Standard SVGA Color PC             A-1      X3000
 Standard SVGA Color PC             B-1      X1010
 Standard SVGA Color PC             B-1      X2000
 Standard SVGA Color PC             B-1      X3000
 Standard SVGA Color PC             C-10     X1010
 Standard SVGA Color PC             C-10     X2000
 Standard SVGA Color PC             C-10     X3000
```

6.15.5.4 INTO

If the result consists of one row, this clause loads the data into one or more variables, one for each column value in the result.



Options

ind_var
Stores an INTEGER value (-1 or 0) that indicates whether the preceding *into_var* received a null value or a non-null value; this is an optional indicator variable. If you omit indicator variables, R:BASE displays a message and assigns a negative integer to SQLCODE if it encounters a null value. The command continues to process rows.

INDICATOR
Indicates the following variable is an indicator variable, which is used to indicate if a null value is retrieved.

into_var
Assigns the result associated with a column, expression, or function named in the command clause to the corresponding variable named in the INTO clause. The number of items or variables named in the command and INTO clauses, as well as their data types, must be the same.

About the SELECT INTO command

This optional clause loads the results of a SELECT command into variables, but does not display the results on screen.

An INTO clause loads the resulting value of each column, expression, or function included in the command clause into a variable. If previous clauses have returned more than one row, the values assigned to the variables are unpredictable. You should make sure you are returning only one row. Either test the results before using an INTO clause or check the value of the variable *sqlcode* after executing the command. If the clause is successful, *sqlcode* is 0.

Comments

The INTO clause must have a corresponding variable for every item in the command clause; values are assigned to variables in the order of items in the command clause. The data type of each command clause item and its corresponding *into_var* must be compatible. For example:

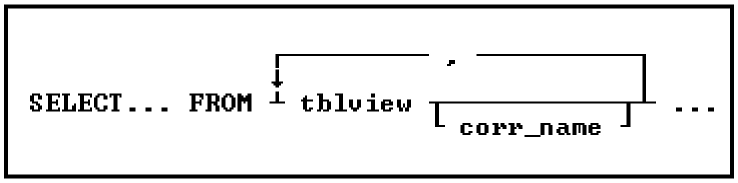
```
SELECT MAX(listprice), MIN(listprice) +
   INTO vmaxprice INDICATOR vind_max, +
               vminprice INDICATOR vind_min +
                     FROM product
```

The MAX and MIN functions assign the value $3,100.00 to the variable *vmaxprice* and $1,900.00 to *vminprice*. These values are the maximum and minimum values for the *listprice* column in the *product* table. Since both functions returned values, the value of both indicator variables is 0. Also, since only SELECT functions are specified, a GROUP BY clause is not required.

If you select and load a value into an undefined numeric variable, that variable acquires the precision and scale of the column from which the value is selected.

6.15.5.5 FROM

Starting with all the tables, views, rows, and columns in the database, this clause specifies one or more tables or views from which you want data.



Options

,
Indicates that this part of the command is repeatable.

corr_name
A correlation name is an alias or nickname for a table. It lets you refer to the same table twice in one command, use a shorter name, and explicitly refer to a column when referring to the same column if that column appears in more than one table. The correlation name must be at least two characters.

tblview
A table or view containing one or more columns named in the command clause.

About the FROM Clause

The FROM clause names one or more tables and/or views from which the information is used in a SELECT command or other command. It is one of the two **REQUIRED** portions of a SELECT statement. The other required portion being the column listing. Some other commands that may use a FROM clause include TALLY, COMPUTE and CHOOSE.

Examples

The following command selects all columns from the *transmaster* table in the R:BASE sample database, *concomp*.

```
SELECT * FROM transmaster
```

The result of this command appears in the following table. The *transid* column is the primary key for this table; that is, *transid* contains a unique value for each row in the table. Columns that are not primary keys can have the same value in more than one row. The result shown here is used in the discussions of other SELECT clauses later in this section.

transid	custid	empid	transdate	netamount	freight
4760	100	133	01/02/94	$32,400.00	$324.00
4780	105	160	01/08/94	$9,500.00	$95.00
4790	104	129	01/09/94	$6,400.00	$64.00
4795	101	102	01/11/94	$176,000.00	$1,760.00
4800	105	160	02/22/94	$194,750.00	$1,947.50
4865	102	129	02/22/94	$34,125.00	$341.25
4970	103	131	02/23/94	$152,250.00	$1,522.50
4975	101	102	02/26/94	$87,500.00	$875.00
4980	101	102	02/27/94	$22,500.00	$225.00
5000	101	102	02/28/94	$40,500.00	$405.00
5010	107	131	03/02/94	$108,750.00	$1,087.50
5015	103	131	03/05/94	$80,500.00	$805.00
5050	104	129	03/06/94	$56,250.00	$562.50
5060	101	102	03/07/94	$57,500.00	$575.00
5065	106	160	03/13/94	$140,300.00	$1,403.00
5070	104	129	03/14/94	$95,500.00	$955.00

Copyright © 1982-2025 R:BASE Technologies, Inc.

```
5075      102      129      03/15/94     $155,500.00    $1,555.00
5080      100      133      03/19/94     $88,000.00     $880.00
5085      107      131      03/18/94     $130,500.00    $1,305.00
5045      100      102      09/26/94     $3,060.00      $30.60
5046      101      165      09/27/94     $3,060.00      $30.60
5047      102      167      09/27/94     $3,830.00      $38.30
5048      103      133      -0-          $12,740.00     $127.40
5049      102      165      04/21/94     $26,310.00     $263.10
```

When a column appears in more than one table, enter the table name and a period preceding each column name to specify the column you want. For example:

```
SELECT transmaster.transid, transmaster.netamount,+
transdetail.model FROM transmaster, transdetail +
WHERE transmaster.transid = transdetail.transid
```

Or, you can assign a correlation name to a table. The following command is equivalent to the previous example:

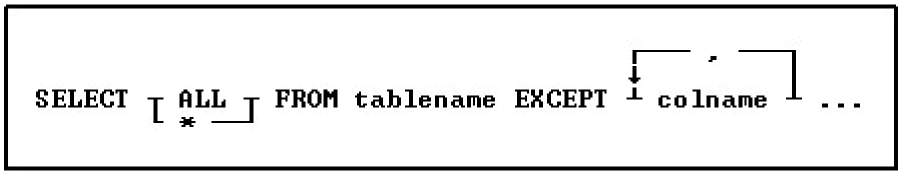
```
SELECT t1.transid, t1.netamount, t2.model +
FROM transmaster t1, transdetail t2 +
WHERE t1.transid = t2.transid
```

In this SELECT command, the FROM clause assigns correlation names to the *transmaster* and *transdetail* tables. Because the *transid* column appears in both tables, the correlation names, *t1* and *t2*, clarify which table each column is from.

Because R:BASE processes the FROM clause first, you must use correlation names, if you have assigned them, throughout the SELECT command.

6.15.5.6 EXCEPT

The EXCEPT clause allows for specified columns to be excluded from a SELECT query.



The EXCEPT clause is another clause you can have in a SELECT command like WHERE and ORDER BY.

SELECT ... EXCEPT selects data from all records in the table, except for columns in exclusion list. When using SELECT ... EXCEPT there can only be one table in the FROM clause.

The EXCEPT clause is helpful with tables which have many columns, to exclude columns from the result set, where the syntax This saves time to type the long list of column names in a SELECT statement.

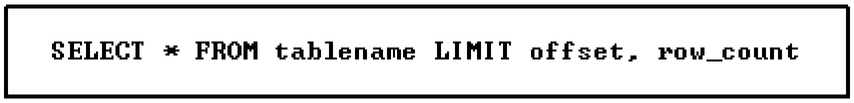
Examples

The below selects all columns from the *Customer* table, except *CustURL* and *CustEMail*
```
SELECT ALL FROM Customer EXCEPT CustURL, CustEMail
```

The below selects all columns from the *Employee* table where the sales bonus are greater than $200.00
```
SELECT ALL FROM Employee EXCEPT HireDate, EntryDate WHERE EmpID IN (SELECT EmpID FROM
SalesBonus WHERE Bonus > 200.00)
```

6.15.5.7 LIMIT

This parameter includes the support to LIMIT the SELECT results.



```
SELECT * FROM tablename LIMIT offset, row_count
```

The LIMIT clause can be used to constrain the number of rows returned by the SELECT statement. LIMIT takes one or two numeric arguments, which must be integer.

With two arguments, the first argument specifies the offset of the first row to return, and the second specifies the maximum number of rows to return. The offset of the initial row is 0 (not 1):

```
SELECT * FROM table LIMIT 5,10
```
Retrieves rows 6-15

To retrieve all rows from a certain offset up to the end of the result set, you can use some large number for the second parameter. This statement retrieves all rows from the 96th row to the last:

```
SELECT * FROM table LIMIT 95,99999999
```

With one argument, the value specifies the number of rows to return from the beginning of the result set:

```
SELECT * FROM table LIMIT 5
```
Retrieve first 5 rows

In other words, LIMIT n is equivalent to LIMIT 0,n.

Examples:

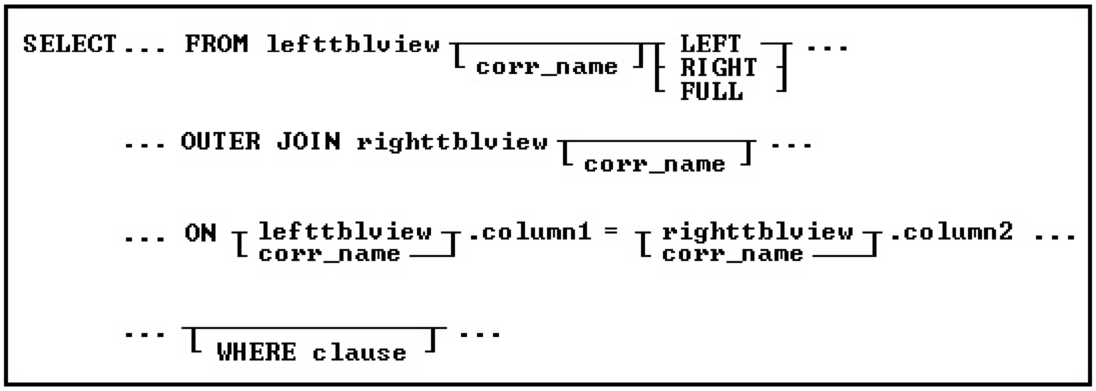
```
SELECT * FROM tablename LIMIT 0,30
```
Gets rows 1-30 from a table

```
SELECT * FROM tablename LIMIT 5,10
```
Gets rows 6-15 from a table

```
SELECT * FROM tablename LIMIT 10
```
Gets the first 10 rows from a table

6.15.5.8 OUTER JOIN

This clause is used to retrieve data from two or more tables.



Options

.column1
Defines the column on which to link.

.column2
Defines the column on which to link.

corr_name
A correlation name is an alias or nickname for a table. It lets you refer to the same table twice in one command, use a shorter name, and explicitly refer to a column when referring to the same column if that column appears in more than one table. The correlation name must be at least two characters.

FROM lefttblview
Specifies the left table or view.

LEFT
RIGHT
FULL
Specifies the type of outer join.

lefttblview
Explicitly defines the column on which to link the left table name or view.

OUTER JOIN righttblview
Specifies the right table or view.

righttblview
Explicitly defines the column on which to link the right table name or view.

WHERE clause
Limits rows of data. See WHERE.

About JOIN

When you perform a SQL JOIN, you specify one column from each table to join on. These two columns contain data that is shared across both tables. You can use multiple joins in the same SQL statement to query data from as many tables as you like.

JOIN Types

Depending on your requirements, you can do an "INNER" join or an "**OUTER**" join. The differences are:

- **INNER JOIN**: This will only return rows when there is at least one row in both tables that match the join condition.
- **LEFT OUTER JOIN**: This will return rows that have data in the left table (left of the JOIN keyword), even if there's no matching rows in the right table.
- **RIGHT OUTER JOIN**: This will return rows that have data in the right table (right of the JOIN keyword), even if there's no matching rows in the left table.
- **FULL OUTER JOIN**: This will return all rows, as long as there's matching data in one of the tables.

About OUTER JOIN

When you use an outer join, rows are not required to have matching values. The table order in the FROM clause specifies the left and right table. You can include a WHERE clause and other SELECT clause options such as GROUP BY. The result set is built from the following criteria:

- In all types of outer joins, if the same values for the linking columns are found in each table, R:BASE joins the two rows.
- For a left outer join, R:BASE uses each value unique to the left (first) table and completes it with nulls for the columns of the right (second) table when the linking columns do not match.
- A right outer join uses unique values found in the right (second) table and completes the rows with nulls for columns of the left (first) table when the linking columns do not match.
- A full outer join first joins the linking values, followed by a left and right outer join.

Nested JOINs

Any of the JOIN types can be mixed in any sequence to create a nested join. The nested joins still require that you specify one column from each table to join on. When nesting joins, it is important to use the correct sequence of parenthesis, along with a correlation for each join.

In the example below, notice the two sets of parenthesis, which all begin after the FROM keyword and end after the linking columns. Also note the "J1" and "J2" correlations specified for each join.

```
SELECT ALL FROM (( TABLE1 t1 +
LEFT OUTER JOIN TABLE2 t2 ON t1.FieldT2=t2.FieldT2) J1 +
RIGHT OUTER JOIN TABLE3 t3 ON t3.FieldT3=j1.FieldT3) J2
```

Examples

The following example lists all of the employees and their total sales, including those employees who have not yet completed a sale.

```
SELECT t1.empid, SUM(t2.netamount) FROM employee t1 +
FULL OUTER JOIN transmaster t2 ON t1.empid = t2.empid +
GROUP BY t1.empid
```

```
t1.empid      SUM(t2.netamount)
102           $387,060.00
129           $347,775.00
131           $472,000.00
133           $133,140.00
160           $344,550.00
165           $29,370.00
166           $0.00
167           $3,830.00
```

Nested INNER JOIN Example:

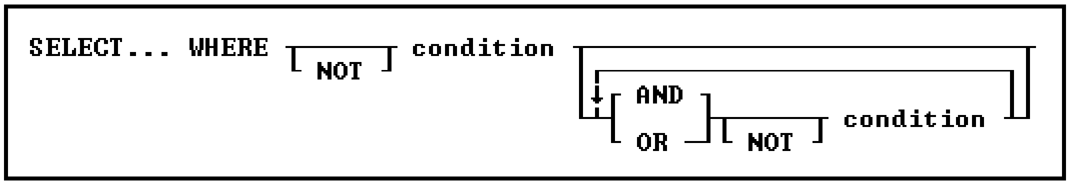
The following example lists a specific product, all of the locations where it resides, and the components used within the product.

```
SELECT ProdName, Location, CompID FROM  +
((Product t1 RIGHT OUTER JOIN ProdLocation t2 ON t1.Model=t2.Model) J1 +
 RIGHT OUTER JOIN CompUsed t3 ON t3.Model=j1.Model) J2 +
WHERE Model = 'CX3000'
```

```
 ProdName                            Location CompID
 ----------------------------------- -------- --------
 Standard SVGA Color PC              A-1      X1010
 Standard SVGA Color PC              B-1      X1010
 Standard SVGA Color PC              C-10     X1010
 Standard SVGA Color PC              A-1      X2000
 Standard SVGA Color PC              B-1      X2000
 Standard SVGA Color PC              C-10     X2000
 Standard SVGA Color PC              A-1      X3000
 Standard SVGA Color PC              B-1      X3000
 Standard SVGA Color PC              C-10     X3000
```

6.15.5.9 WHERE

This clause determines which rows of data to include.



Options

AND
Indicates the following condition must be met along with the preceding condition.

condition
Identifies requirements to be in the WHERE syntax.

NOT
Reverses the meaning of a connecting operator. AND NOT, for example, indicates that the first condition must be met and the following condition must not be met.

OR
Indicates the following condition can be met instead of the preceding condition.

About the WHERE Clause

In most commands, a WHERE clause follows the syntax diagram above.

The two main elements in any WHERE clause are conditions and connecting operators.

We now support "COUNT = LAST" in two different ways. If the entire WHERE clause is "WHERE COUNT = LAST" then R:BASE works like it always has to quickly fetch the last row of the table. The NEW functionality is to have other conditions in the WHERE clause and you want the last row of whatever qualifies.

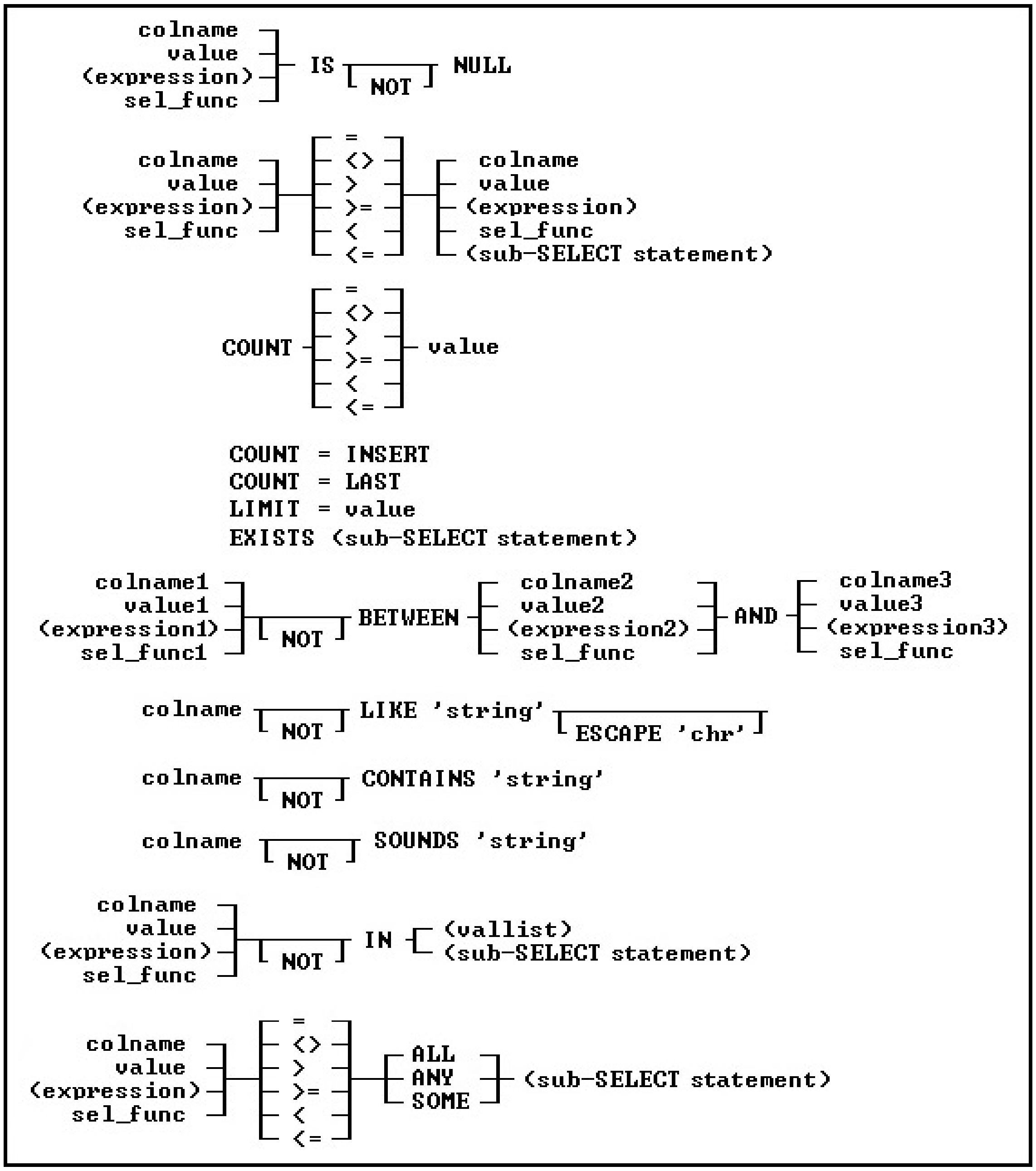
To make it work this way specify the other conditions and then add "AND COUNT = LAST".

Here is an example:

SELECT * FROM Customer WHERE CustID > 100 AND COUNT = LAST

WHERE Clause Conditions

The following syntax diagram and table show the basic formats for WHERE clause conditions, which can be used alone or together.



Basic WHERE Clause Conditions

Condition Syntax	Description
colname op DEFAULT	True if a column value compares correctly with the DEFAULT value for the column. *Op* can be =, <>, >=, >, <=, or <.
colname = USER	True if a column value equals the current user identifier.
item1 IS NULL	True if *item1* has a null value. *Item1* can be a column name, value, or expression. A null value cannot be used in a comparison with an operator.
item1 op item2	True if the relationship between two items is true as defined by an operator. *Item1* can be a column name, value, or expression; *item2* can be a column name, value, expression, or sub-SELECT statement.

COUNT=INSERT	Refers to the last row inserted in a table by the current user, even if it has been modified by another user. The COUNT=INSERT condition can be used with a single-table view, but not with a multi-table view. If there is not a newly inserted row in the table, then COUNT=INSERT performs the same action as COUNT=LAST, and fetches the current end row of the table.
COUNT=LAST	Refers to the last row in a table. The COUNT=LAST condition can be used with a single-table view, but not with a multi-table view.
COUNT *op value*	Refers to a number of rows defined by *op* and *value*.
LIMIT=*value*	Specifies a number of rows affected by a command. A LIMIT condition should be the last condition in a WHERE clause.
EXISTS (sub-SELECT statement)	True if sub-SELECT statement returns one or more rows.
item1 BETWEEN *item2* AND *item3*	True if the value of *item1* is greater than or equal to the value of *item2*, and if the value of *item1* is less than or equal to the value of *item3*.
colname LIKE '*string* '	True if a column value equals the text string. With LIKE, a string can also be a DATE, TIME, or DATETIME value. The text string can contain R:BASE wildcard characters.
colname LIKE '*string* ' ESCAPE '*chr* '	True if a column value equals a text string. If you want to use a wildcard character as a text character in the string, specify the ESCAPE character 1*chr*. In the string, use *chr* in front of the wildcard character.
colname CONTAINS '*string* '	True if a column value contains the text string.
colname SOUNDS '*string* '	True if the soundex value of a column matches the soundex value of the text string.
item1 IN (*vallist*)	True if *item1* is in the value list.
item1 IN (sub-SELECT statement)	True if *item1* is in the rows selected by a sub-SELECT.
item1 op ALL (sub-SELECT statement)	True if the relationship between *item1* and every row returned by a sub-SELECT statement matches an operator.
item1 op ANY(sub-SELECT statement)	True if the relationship between *item1* and at least one value returned by a sub-SELECT statement matches an operator.
item1 op SOME (sub-SELECT statement)	ANY and SOME are equivalent.

Notes:

- Placing NOT before most text operators (such as NULL or BETWEEN) reverses their meaning.
- When a SELECT statement is part of a WHERE clause, it is called a sub-SELECT clause. A sub-SELECT clause can contain only one column name (not a column list or *), expression, or function. The INTO and ORDER BY clauses in a sub-SELECT are ignored.

You can only use the current wildcard characters to compare a column to a text value when using the LIKE comparison. The default wildcard characters are the percent sign (%), which is used for one or more characters, and the underscore (_), which is used for a single character.

If you compare a column with a value, you can either enter the value or specify a global variable. If you specify a variable, R:BASE compares the column with the current value of the variable.

To significantly reduce processing time for a WHERE clause, use INDEX processing. To use indexes, the following conditions must be met:

- A condition in the WHERE clause compares an indexed column.
- If the WHERE clause contains more than one condition, R:BASE selects the condition that places the greatest restriction on the WHERE clause.
- Conditions are not joined by the OR operator.
- The comparison value is not an expression.

Connecting Operators

When you use more than one condition in a WHERE clause, the conditions are connected using the connecting operators AND, OR, AND NOT, and OR NOT.

The connecting operator AND requires that both conditions it separates must be satisfied. The connecting operator OR requires that either condition it separates must be satisfied.

The connecting operator AND NOT requires that the preceding condition must be satisfied, and the following condition must not be satisfied. The connecting operator OR NOT requires that either the preceding condition must be satisfied, or any condition except the following condition must be satisfied.

In WHERE clauses with multiple conditions, conditions that are connected by AND or AND NOT are evaluated before those connected by OR or OR NOT. However, you can control the order in which conditions are evaluated by either placing parentheses around conditions or using the SET AND command. If you set AND off, conditions are always evaluated from left to right.

Examples

The following WHERE clause chooses sales amounts that are less than the value of a variable containing the daily average.

```
... WHERE amount < .dailyave
```

The following WHERE clause specifies the seventh row.

```
... WHERE COUNT = 7
```

The following WHERE clause specifies each row from the *employee*table that contains both the first name *June* and the last name *Wilson*.

```
SELECT * FROM employee WHERE empfname = 'june' AND emplname = 'wilson'
```

The following WHERE clause selects dates in the *actdate* column that are greater than dates in the *begdate* column or are less than dates in the *enddate* column.

```
... WHERE actdate BETWEEN begdate AND enddate
```

The next three WHERE clauses use the following data:

```
empfname emplname
-------- --------
    Mary Jones
    John Smith
   Agnes Smith
    John Brown
```

In both of the following clauses, R:BASE first evaluates the conditions connected by AND, selecting John Smith. Then R:BASE adds any Marys to the list because the connecting operator is OR. The final result includes John Smith and Mary Jones.

```
...WHERE empfname = 'Mary' OR empfname = 'John' +
 AND emplname = 'Smith'
```

```
...WHERE empfname = 'Mary' OR (empfname = 'John' +
 AND emplname = 'Smith')
```

By moving the parentheses around the conditions connected by OR, you can select only John Smith. In the following WHERE clause, the first name can be either Mary or John, but the last name must be Smith.

```
...WHERE (empfname = 'Mary' OR empfname = 'John') AND +
 emplname = 'Smith'
```

The following example illustrates a sub-SELECT in a WHERE clause. Assume you wanted a list of all sales representatives that had transactions greater than $100,000, and the information for such a list was contained in two tables, *employee* and *transmaster*. The relevant columns in these tables are:

```
employee transmaster
empid  emplname empid    netamount
----- --------- ----- ------------
  102 Wilson      133   $32,400.00
  129 Hernandez   160    $9,500.00
  133 Coffin      129    $6,400.00
  165 Williams    102  $176,000.00
  166 Chou        160  $194,750.00
  167 Watson      129   $34,125.00
  160 Smith       131  $152,250.00
  131 Simpson     102   $87,500.00
  102                   $22,500.00
  102                   $40,500.00
  131                  $108,750.00
  131                   $80,500.00
  129                   $56,250.00
  102                   $57,500.00
  160                  $140,300.00
  129                   $95,500.00
  129                  $155,500.00
  133                   $88,000.00
  131                  $130,500.00
  102                    $3,060.00
  165                    $3,060.00
  167                    $3,830.00
  133                   $12,740.00
  165                   $26,310.00
```

To display a list of employees in the *transmaster* table with a transaction larger than $100,000, enter the following command:

```
SELECT empid, emplname FROM employee WHERE empid IN +
   (SELECT empid FROM transmaster WHERE netamount > 100000)
```

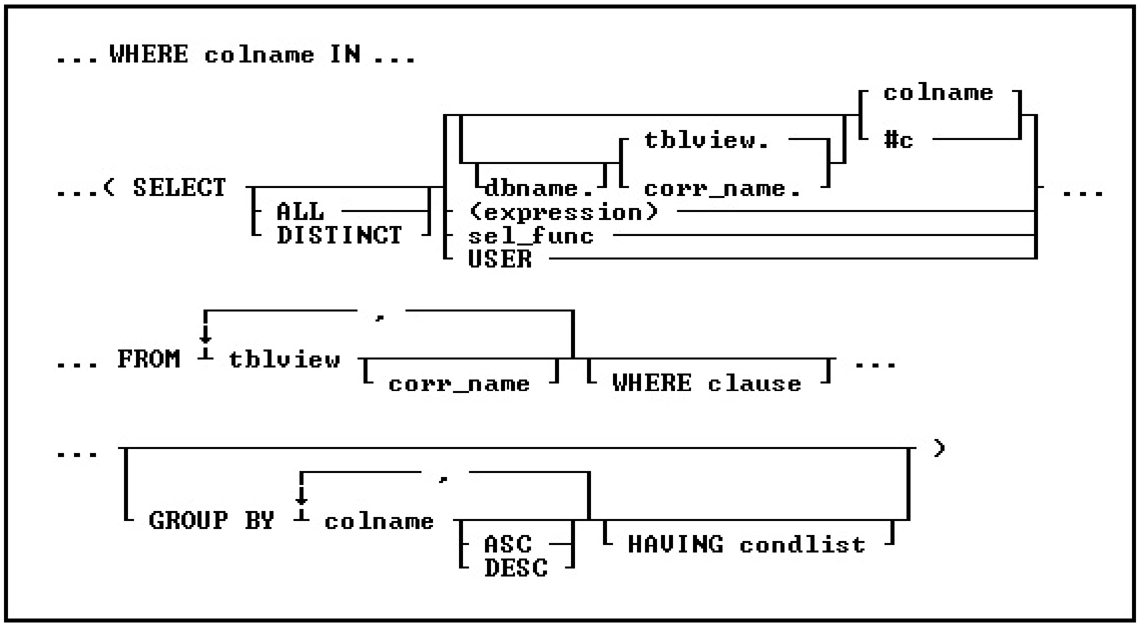
R:BASE displays the following list:

```
    empid emplname
--------- ----------------
      102 Wilson
      129 Hernandez
      131 Simpson
      160 Smith
```

Note: You can use a sub-SELECT in any command that allows a full WHERE clause.

6.15.5.10 Sub-SELECT

This clause works in conjunction with the WHERE clause to determine which rows of data to include.



Options

For a description of the options, see SELECT.

About the Sub-SELECT Clause

A sub-SELECT command, which is a SELECT command nested within another command, always appears in a WHERE clause, whether the sub-SELECT command is nested in the WHERE clause of a SELECT command or in another command such *as EDIT USING form*. R:BASE processes the clauses in a sub-SELECT in the same order as in a SELECT command.

Example

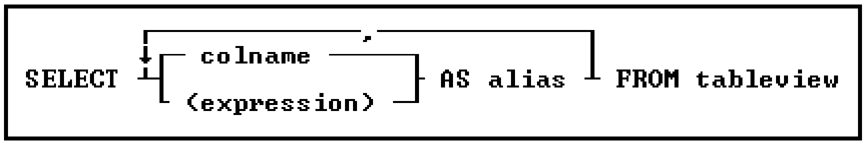
The following example selects customers that have purchased items in the month of January.

```
SELECT company FROM customer WHERE custid +
   IN (SELECT custid FROM transmaster +
   WHERE (IMON(transdate)=1))
```

company
```
PC Distribution Inc.
Computer Distributors Inc.
Industrial Concepts Inc.
PC Consultation and Design
```

6.15.5.11 AS

This clause dynamically renames columns in a SELECT clause.



Options

,
Indicates that this part of the command is repeatable.

colname
Specifies a column name. The column name is limited to 128 characters.

(expression)
Determines a value using a text or arithmetic formula. The expression can include other columns from the table, constant values, functions, or system variables such as *#date*, *#time*, and *#pi*.

AS alias
Determines the alias of the column. This may be used to refer to the column in other locations.

FROM tableview
Specifies the table or view to draw information from.

About the SELECT AS command

The SELECT AS command functions exactly like any other SELECT command and will accept all other SELECT options. The one noticeable exception is that this allows you to give a column an alias. This can be most useful when used in conjunction with the CREATE VIEW command.

Examples

The following 3 examples are based on the *ConComp* database.

The following command selects the EmpID and the EmpName columns from the Employee table and renames them to "EmployeeID" and "Name".

```
SELECT EmpID AS EmployeeID,EmpName AS Name FROM Employee
```

The following command creates a VIEW using the SELECT AS notation. This view contains a column for Employee ID, Employee Name (which is a single column based on the EmpFName and EmpLName columns) from the Employee table and the Transdate and NetAmount columns from the Transmaster table. The immediately following command browses the Employee Name, Transdate and Netamount column. For more on Views please see CREATE VIEW.

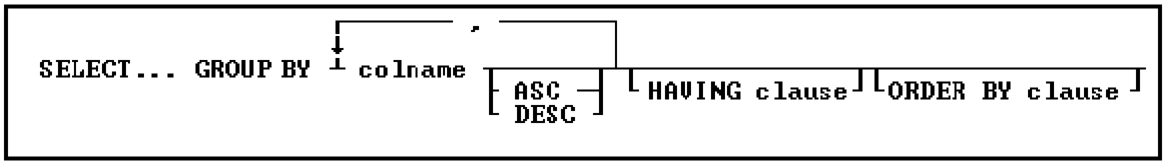
```
CREATE VIEW EmpAmount AS SELECT T1.EmpID, +
   (T1.EmpFName + ' ' + T1.EmpLName) AS EmpName, +
   T2.TransDate,T2.NetAmount FROM Employee T1, +
   TransMaster T2 WHERE T1.EmpID = T2.EmpID

BROWSE EmpName,TransDate,NetAmount FROM EmpAmount
```

This final example uses IDQuotes to create a column name with spaces in it and then uses the SELECT HTML option to turn that into an HTML table with the column name. The OUTPUT commands redirect output to a file called "Emp.HTM" and then back to the screen. We do NOT recommend using this method to create VIEWS or TABLES with names that contain spaces as this could lead to Database Corruption.

```
OUTPUT EMP.HTM
SELECT EmpID as `Employee ID` FROM Employee HTML
OUTPUT SCREEN
```

6.15.5.12 GROUP BY

This clause determines which rows of data to include.



Options

,
Indicates that this part of the command is repeatable.

ASC
DESC
Specifies whether to sort a column in ascending or descending order.

colname
Specifies a column name. The column name is limited to 128 characters.

In a command, you can enter *#c*, where *#c* is the column number shown when the columns are listed with the LIST TABLES command. In an SQL command, a column name can be preceded by a table or correlation name and a period (*tblname.colname*).

GROUP BY
Returns a groups of rows as a summary resulting in only unique rows. This option is generally used with SELECT Functions.

HAVING clause
Limits the rows affected by the GROUP BY clause.

ORDER BY clause
Sorts rows of data.

About the GROUP BY command

This optional clause groups rows according to the values in one or more columns and sorts the results. GROUP BY consolidates the information from several rows into one row. This results in a table with one row for each value in the named column or columns and one or more values per column.

The columns listed in the GROUP BY clause are related to those listed in the command clause. Any column named in the GROUP BY clause can also be named in the command clause, but any column not named in the GROUP BY clause can be used only in the command clause if the column is used in a SELECT function.

Examples

The SELECT command clause can contain the columns named in the GROUP BY clause, and SELECT functions that refer only to columns not named in the GROUP BY clause. Because the GROUP BY clause processes information resulting from a WHERE clause, you can add a GROUP BY clause to see the sales each employee has made:

```
SELECT empid FROM transmaster WHERE netamount < $100,000 +
GROUP BY empid
```

The following intermediate result table contains columns not named in the command clause because the command clause has not been processed yet (not all the columns fit in the display, however). The first part of the processing is to group the rows by *empid.* Because seven different employees are included, the intermediate result table includes seven rows.

Intermediate Result Table-GROUP BY empid

transid	custid	empid	netamount
4975, 4980, 5000, 5060, 5045	101, 101, 101, 101, 100	102	$87,500, $22,500, $40,500, $57,500, $3,060
4790, 4865, 5050, 5070	104, 102, 104, 104	129	$6,400, $34,125, $56,250, $95,500
5015	103	131	$80,500
4760, 5080, 5048	100, 100, 103	133	$32,400, $88,000, $12,740
4780	105	160	$9,500
5046, 5049	101, 102	165	$3,060, $26,310
5047	102	167	$3,830

You can include more than one column in a GROUP BY clause. If you group the rows in the above example by *custid* as well as *empid*, the command looks like this:

```
SELECT empid, custid FROM transmaster +
WHERE netamount < $100,000 GROUP BY empid, custid
```

In the following table, rows are now grouped by both *empid* and *custid*, resulting in eleven groups.

Intermediate Result Table-GROUP BY empid and custid

transid	custid	empid	netamount
5045	100	102	$3,060
4975, 4980, 5000, 5060	101	102	$87,500, $22,500, $40,500, $57,500
4865	102	129	$34,125
4790, 5050, 5070	104	129	$64,000, $56,250, $95,500
5015	103	131	$80,500
4760, 5080	100	133	$32,400, $88,000
5048	103	133	$12,740
4780	105	160	$9,500
5046	101	165	$3,060
5049	102	165	$26,310
5047	102	167	$3,830

If one or more of the columns named in the GROUP BY clause contain null values, R:BASE forms a separate group for null values. Review the result of this SELECT command for the *employee* table:

```
SELECT empid, emplname, hiredate, emptitle FROM employee
```

```
empid       emplname     hiredate     emptitle
102         Wilson       03/18/90     Manager
129         Hernandez    08/28/91     Manager
131         Smith        04/14/92     -0-
133         Coffin       11/26/93     Representative
160         Simpson      01/09/94     -0-
165         Williams     07/05/92     Representative
167         Watson       07/10/92     Representative
166         Chou         07/10/93     Sales Clerk
```

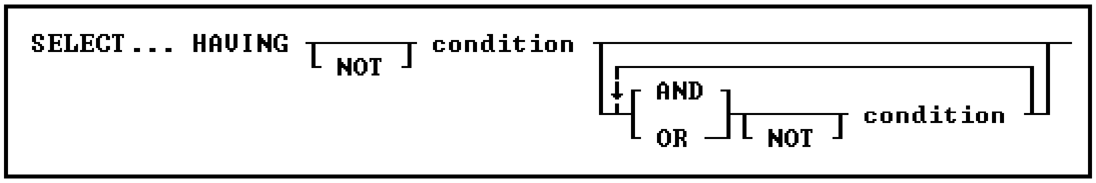
If you group these rows by the *emptitle* column, which contains null values, you get the following intermediate result table:

Intermediate Result Table-GROUP BY emptitle

empid	emplname	hiredate	emptitle
102, 129	Wilson, Hernandez	03/18/90, 08/28/91	Manager
133, 165, 167	Coffin, Williams, Watson	11/26/93, 07/05/92, 07/10/92	Representative
166	Chou	07/10/93	Sales Clerk
131, 160	Smith, Simpson	04/14/94, 01/09/94	-0-

6.15.5.13 HAVING

This clause determines which rows of data to include based on the results of a prior GROUP BY clause.



Options

AND
OR
AND indicates two conditions must both be true.
OR indicates either condition must be true.

condition
Specifies a combination of one or more expressions and/or operations that would evaluate to either true or false. See "HAVING Conditions" below.

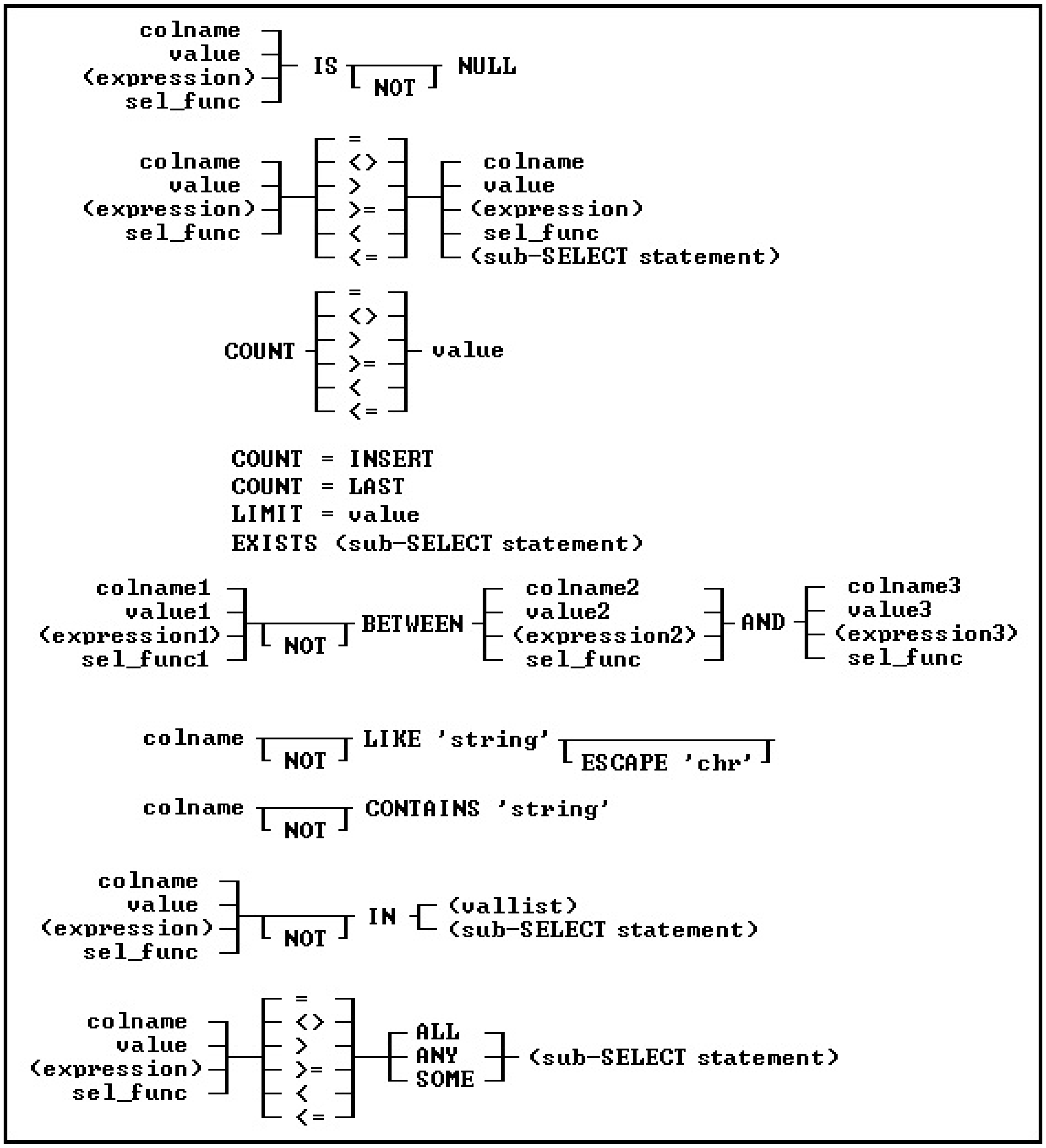
NOT
Reverses the meaning of an operator or indicates that a condition is not true.

About the HAVING command

The optional HAVING clause selects rows that meet one or more conditions from among the results of the GROUP BY clause. HAVING works the same as a WHERE clause with the following exceptions:

- A WHERE clause modifies the intermediate results of a FROM clause; a HAVING clause modifies the intermediate results of a GROUP BY clause.
- A HAVING clause can include SELECT Functions.

HAVING Conditions:



Examples

To display sales information for only those employees who have made more than one sale to the same customer, add a HAVING clause such as the following to one of the examples shown in GROUP BY. When used in a HAVING clause, SELECT functions compute results based on the values grouped in the specified column. In this HAVING clause, COUNT returns the number of values grouped in the *transid* column.

```
SELECT empid, custid FROM transmaster +
WHERE netamount < $100,000 +
GROUP BY empid, custid HAVING COUNT(transid) > 1
```

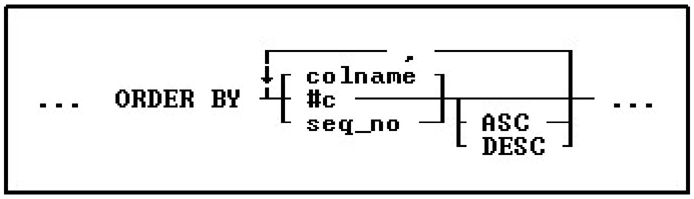
Intermediate Result Table-HAVING COUNT(transid) > 1

transid	custid	empid	netamount
4975, 4980, 5000, 5060	101	102	$87,500, $22,500, $40,500, $57,500

```
4790, 5050, 5070            104    129    $6,400, $56,250, $95,500
5080                        100    133    $32,400, $88,000
```

6.15.5.14 ORDER BY

This clause specifies how to sort the result of the SELECT command.



Options

,
Indicates that this part of the command is repeatable.

ASC
DESC
Specifies whether to sort a column in ascending or descending order.

#c
Takes the place of a column name and refers to the column numbers displayed with the LIST TABLE command.

colname
Sorts by any column name or combination of column names.

seq_no
Refers to the items listed in the SELECT command that is using the ORDER BY command, ordered from left to right. An item can be a column name, expression, or SELECT function.

About the ORDER BY Command

The syntax for the ORDER BY clause is the same for all commands. ORDER BY must refer to only one table or view.

You can significantly reduce the time R:BASE takes to process an ORDER BY clause when the column or columns listed in the ORDER BY clause are included in an index with the same column sort order as that specified in the ORDER BY clause.

Using the SET SORT Command

The ORDER BY command uses the R:BASE automatic sort optimizer. If you are sorting extremely large tables, and if your disk space is limited, the automatic sort optimizer might be unable to sort the data. Instead, use the SET SORT ON condition because it uses the least disk space necessary to sort data; however, the SET SORT ON condition is slower than the automatic sort.

Examples

The following command displays data from the *custid*, *company*, and *custcity* columns from the *customer*table.

```
SELECT custid, company, custcity FROM customer
```

The ORDER BY clause in the command below arranges the *custid*values in descending order.

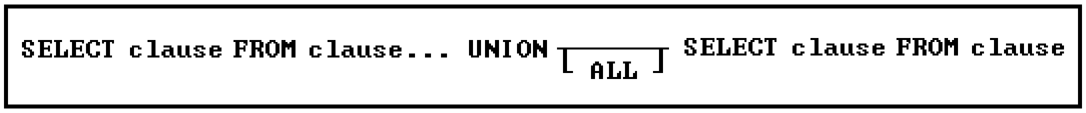
```
SELECT custid, company, custcity FROM customer +
ORDER BY custid DESC
```

You can substitute a column's sequence number for a column named in the ORDER BY clause. You must use a sequence number when referring to an expression, function, constant, or when a UNION operator is used. The following command is equivalent to the command example above.

```
SELECT custid, company, custcity FROM customer ORDER +
BY 1 DESC
```

6.15.5.15 UNION

You can use this operator to combine the results of two or more SELECT statements.



About the UNION SELECT command

This optional operator combines the results of two SELECT commands or clauses, displaying the results of the second SELECT command below those of the first. By default, UNION deletes duplicate rows. Include the optional keyword ALL to include duplicate rows in the final result. You cannot combine sub-SELECT commands using UNION.

The UNION operator requires the following three conditions:

- The SELECT statements must specify an equal number of columns.
- Columns that are being combined must have the same data type.
- Only the last SELECT statement can contain an ORDER BY clause.

Examples

The following example lists all employees and the sales transactions for each, including those employees who have not yet completed a sale.

```
SELECT employee.empid, transid +
   FROM employee, transmaster +
   WHERE employee.empid = transmaster.empid +
UNION SELECT empid, 0 +
   FROM employee +
   WHERE employee.empid NOT IN +
      (SELECT empid FROM transmaster)
```

The first SELECT displays the *empid* column from the *employee* table and *transid* from the *transmaster* table, linking the tables by the common column, *empid*. In short, the first SELECT displays all employees who have made a sale.

The second SELECT command selects the *empid* column from *employee*, including rows only for those employees who are not listed in the *transmaster* table. Because the results of the second SELECT are appended to those of the first (by the UNION operator), those employees who have not yet made a sale are shown at the bottom of the results with a zero in the *transid* column. The final results look like this:

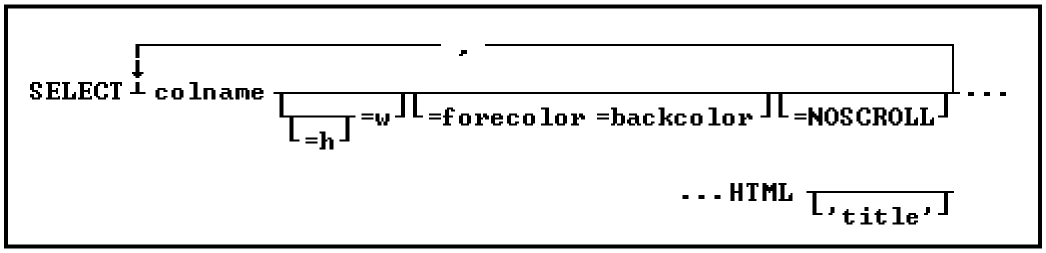
```
empid  transid
102    4795
102    4975
102    4980
102    5000
102    5045
102    5060
129    4790
129    4865
129    5050
```

```
129    5070
129    5075
131    4970
131    5010
131    5015
131    5085
133    4760
133    5048
133    5080
160    4780
160    4800
160    5065
165    5046
165    5049
166    0
```

6.15.5.16 HTML

This clause sends the SELECT command output into HTML format.

The rows and columns that each clause selects produce an intermediate result table that exists only in temporary memory. One after another, the clauses restrict the rows or columns included in the intermediate result table. After R:BASE has processed all the clauses, the intermediate result table becomes the final result table.



Options

=backcolor
Is currently not supported; it must be set to: =default.

colname
Specifies a column name. The column name is limited to 128 characters.

=forecolor
Specifies the text color for the data in the column.

=h =w
Specifies height and width parameters. For both, the keywords LEFT, RIGHT, and CENTER can be used. For width, the range is from 0 - 255 pixels. 0 is no width specification. The height is the addition of the following values, one value from each item:

data justification:	0=default, 1=left, 2=center, 3=right
heading justification:	0=default, 4=left, 8=center, 12=right
vertical alignment:	0=default, 16=top, 32=middle, 48=bottom
HTML format flag:	0=no, 64=yes

HTML
Converts the data to HTML code.

'title'
This is the text that appears in the caption at the top of the Web Browser window. Adding '*title*' creates the beginning and ending table tag as well as putting the text you enter in the caption. Without '*title*', a partial HTML file with the selected data in table row format is generated.

About the SELECT...HTML Command

The SELECT...HTML command is a modification of the SELECT command to output data in HTML format. If a title is specified, a full HTML file will be generated. Otherwise, a partial HTML file with the selected data in table row format will be generated.

6.15.6 SET

Use SET to change the current status of R:BASE special characters and operating conditions for any SET category. The SET operating conditions are used to set the database environment while you build and run commands from the R> Prompt or command files and applications.

About the SET Command

To change the setting from the R> Prompt for a special character, enter:

```
SET character_name = value
```

Do not use spaces on either side of the equal sign (=). For example:

```
SET QUOTES='
```

To change the setting from the R> Prompt for an operating condition, enter:

```
SET keyword setting
```

For example:

```
SET BELL ON
```

To remap keys on the keyboard from the R> Prompt, enter:

```
SET KEYMAP keyname = remapped keys
```

For example, if you want to remap [SHIFT] [F6] to [F2], enter:

```
SET KEYMAP [SHIFT] [F6] = [F2]
```

To reset a key to its original default value from the R> Prompt, enter:

```
SET KEYMAP keyname OFF
```

Saving Settings

Settings can be changed at the R> Prompt for the current session. They will revert to the default upon exiting R:BASE. Users can save settings in the R:BASE configuration file. Some settings are stored within the database itself and only the database owner can save changes to the settings that are stored in the database. If you are not the database owner, you can only change the settings that are stored in the database for the current database session.

The R:BASE/Oterro database provides the following categories of SET Keywords:

- Data Integrity
- Display Control
- Environment
- Format
- Programming
- Special Characters
- Transaction Processing and Multi-User
- Database Specific

6.15.6.1 AND

Operating Condition

Syntax: SET AND ON/OFF

Default: ON

SET AND gives the connecting operator AND precedence over OR in WHERE, IF, and WHILE conditions. When on, R:BASE processes conditions in the following order: all AND NOT followed by all AND OR, and all AND before all OR conditions.

SET AND OFF directs R:BASE to process conditions from left to right. The WHERE clause A AND B OR C AND D is evaluated as (((A AND B) OR C) AND D). SET AND ON directs R:BASE to give the operator AND precedence over OR. The WHERE clause A AND B OR C AND D is evaluated as ((A AND B) OR (C AND D)), which is a different result set.

6.15.6.2 ANSI

Operating Condition

Syntax: SET ANSI ON/OFF

Default: ON

Set ANSI off to restrict R:BASE to the list of reserved words shown in the table below. To change the default, save the setting to the configuration file.

R:BASE reserved words with ANSI set off:

ABS	**ACOS**	**AINT**	**ALL**
AND	**ANINT**	**ASIN**	**ATAN**
ATAN2	**AVERAGE**	**CHAR**	**CHKKEY**
COS	**COSH**	**COUNT**	**CTR**
CTXT	**CURRENT**	**CVAL**	**DATE**
DATETIME	**DEXTRACT**	**DIM**	**ENVVAL**
EXP	**FLOAT**	**FOR**	**FORMAT**
FROM	**FULL**	**FV1**	**FV2**
GETKEY	**HELP**	**ICAP1**	**ICAP2**
ICHAR	**IDAY**	**IDWK**	**IFEQ**
IFGT	**IFLT**	**IFRC**	**IHASH**
IHR	**IMIN**	**IMON**	**IN**
INT	**ISEC**	**ISTAT**	**IYR**
JDATE	**LAST**	**LASTKEY**	**LAVG**
LIMIT	**LIST**	**LJS**	**LMAX**
LMIN	**LOG**	**LOG10**	**LUC**
MAXIMUM	**MINIMUM**	**MOD**	**NEW**
NEWPAGE	**NEXT**	**NINT**	**NOT**
OR	**OUTER**	**PASSTAB**	**PMT1**
PMT2	**PROMPT**	**PROMPTS**	**PV1**
PV2	**RATE1**	**RATE2**	**RATE3**
RDATE	**RJS**	**RTIME**	**SFIL**
SGET	**SIGN**	**SIN**	**SINH**
SLEN	**SLOC**	**SMOVE**	**SPUT**
SQRT	**SRPL**	**SSUB**	**STDEV**
STRIM	**SUM**	**TAN**	**TANH**
TDWK	**TERM1**	**TERM2**	**TERM3**
TEXT	**TEXTRACT**	**TMON**	**UDF**
ULC	**UNNAMED**	**VARIABLE**	**VARIANCE**
WHERE			

6.15.6.3 AUTOCOMMIT

Operating Condition

Syntax: SET AUTOCOMMIT ON/OFF

Default: OFF

Mode: Transaction Processing

SET AUTOCOMMIT toggles AUTOCOMMIT processing on and off. When transaction processing and AUTOCOMMIT are on, each command that is executed successfully is immediately made permanent and visible to network users. If transaction processing is on and AUTOCOMMIT is off, you must enter a COMMIT command to make changes permanent. Also, leaving the database causes R:BASE to issue the COMMIT command.

When transaction processing is on and AUTOCOMMIT is off, you can enter a series of commands (a transaction) that change data or the database structure, then enter either a COMMIT or ROLLBACK command. COMMIT makes permanent all changes executed by commands in the transaction. ROLLBACK deletes all the changes, restoring the database to its state before the transaction began.

If you have started a transaction when you set AUTOCOMMIT on, R:BASE commits the transaction and turns AUTOCOMMIT on. You cannot open a cursor while AUTOCOMMIT is set on, and you cannot set AUTOCOMMIT on while a cursor is open.

AUTOCOMMIT can affect system performance. You can increase performance by setting AUTOCOMMIT to on when you do not need to enter commands in groups.

6.15.6.4 AUTOCONVERT

Operating Condition

Syntax: SET AUTOCONVERT ON/OFF

Default: OFF

Set AUTOCONVERT on to automatically convert R:BASE databases created in versions prior to 6.0 to the current R:BASE release. The user is not given the option to halt the conversion process.

6.15.6.5 AUTODROP

Operating Condition

Syntax: SET AUTODROP ON/OFF

Default: OFF

Controls the feature for a combo-box in a form to automatically drop-down the list when it gets focus.

When AUTODROP is SET to ON, this new setting controls the feature for a combo-box in a form to automatically drop-down the list when it gets focus. The editable or non-editable automatic drop-down list of combo box will
allow the user to place the value into a column or variable.

This new setting can be saved in your R:BASE configuration file or in your individual application startup file.

Supported environments:

- Configuration File
- Command File

Notes:

- SHOW AUTODROP will display the current setting of AUTODROP.
- (CVAL('AUTODROP')) will return the current setting of AUTODROP

6.15.6.6 AUTORECOVER

Operating Condition

Syntax: SET AUTORECOVER ON/OFF

Default: OFF

Mode: Transaction Processing

If AUTORECOVER is set on, errors that can occur during transaction processing when the program in interrupted, for example from a network or power failure, are automatically corrected.

6.15.6.7 AUTOROWVER

Operating Condition

Syntax: SET AUTOROWVER ON/OFF

Default: OFF

AUTOROWVER is used for Oterro compatibility only. If AUTOROWVER is set on, every CREATE TABLE or ALTER TABLE command will add the SYS_ROWVER column if it does not already exist. The SYS_ROWVER column is not comapatible with R:BASE 6.0 and lower databases.

6.15.6.8 AUTOSKIP

Operating Condition

Syntax: SET AUTOSKIP ON/OFF

Default: OFF

Set AUTOSKIP on to move the cursor automatically to the next data-entry field in a form after filling the entire field. Specify off to press [Tab] after each entry. R:BASE stores the setting with the database.

6.15.6.9 AUTOSYNC

Operating Condition

Syntax: SET AUTOSYNC ON/OFF

Default: OFF

If AUTOSYNC is set on, connecting to a database will automatically synchronize the database files if necessary. If AUTOSYNC is set off and an error occurs during the connect because the files are out of sync, the database is not connected.

6.15.6.10 AUTOUPGRADE

Operating Condition

Syntax: SET AUTOUPGRADE ON/OFF

Default: OFF

AUTOUPGRADE converts R:BASE 6.0 databases to the current R:BASE release and adds the new system tables for handling Stored Procedures and Triggers.

6.15.6.11 BELL

Operating Condition

Syntax: SET BELL ON/OFF

Default: ON

Set BELL on to sound the bell when an error occurs. Specify off to suppress the bell. R:BASE stores the setting with the database.

6.15.6.12 BLANK

Special Character

Sets the character for spaces. The BLANK character is used to separate words in a command string.

Syntax: SET BLANK=NULL

Syntax: SET BLANK=char
(Use NULL to disable the special character.)

Default: (space)

6.15.6.13 BOOLEAN

Operating Condition

Syntax: SET BOOLEAN ON/OFF

Default: OFF

Set BOOLEAN to ON will specify that constants (e.g. TRUE, FALSE) in expressions will be treated as type BOOLEAN values.

6.15.6.14 CAPTION

Operating Condition

Syntax: SET CAPTION 'window title'

SET CAPTION specifies a title for an application or a command file. This title appears in the title bar when you run the application or command file.

6.15.6.15 CASE

Operating Condition

Syntax: SET CASE ON/OFF

Default: OFF

SET CASE sets the uppercase or lowercase distinction when a comparison is used with WHERE clauses, IF structures, WHILE loops, the TALLY command, and in the RULES command where comparisons are equal or not equal. R:BASE stores the setting with the database. If CASE is set off, both uppercase and lowercase text are displayed for a comparison regardless of how you enter the text. For example, if you enter "case", you could find "Case" and "CASE".

6.15.6.16 CHECKPROP

Operating Condition

Syntax: SET CHECKPROP ON/OFF

Default: OFF

CHECKPROP displays or suppresses error message for PROPERTY and GETPROPERTY commands during processing. The PROPERTY/GETPROPERTY errors can be displayed if a Component ID is not found or if a property value is invalid. The setting applies to use in forms, reports, and label. The errors are only displayed if CHECKPROP is set ON.

6.15.6.17 CLEAR

Operating Condition

Syntax: SET CLEAR ON/OFF

Default: ON

Mode: Single-user

SET CLEAR determines when R:BASE clears the internal buffers and transfers the data to disk.

When CLEAR is set on, the internal buffers are cleared and data is transferred to disk after each modification. Setting CLEAR on does not always make R:BASE automatically write each new or changed row to disk. For example, when you use a form, R:BASE writes the edits to disk when you finish using the form.

Set CLEAR off to write modified data to disk only when the buffer is full, a database is closed, or you exit R:BASE. If CLEAR is set off, repetitive modifications to a database can run faster, but you could lose all of your changes stored in the buffer if an accident, such as a fluctuation in power supply, occurs.

In multi-user mode, the CLEAR setting has no effect and always acts as though it is set on.

6.15.6.18 CLIPBOARD

Environment

Syntax: SET ClipBoard <TextString or Variable>

About the SET CLIPBOARD Command

Places the specified string onto the Windows clipboard, where it can be accessed from other programs.

EXAMPLE 01:

```
SET CLIPBOARD 'Here is text for the clipboard'
```

EXAMPLE 02:

Assuming you have already created a MS Word Document (TestDoc.DOC) or WordPerfect Document (TestDoc.WPD).

```
-- ClipBrd.RMD
CONNect ConComp
SET VAR vAddressBlock TEXT = NULL
SET VAR vCustID INTEGER = 100
SELECT (Company+(CHAR(10))+CustAddress+(CHAR(10)) +
  +CustCity+','&CustState&CustZip+(CHAR(10))) +
   INTO vAddressBlock INDIC IvAddressBlock +
   FROM Customer WHERE CustID = .vCustID
SET CLIPBOARD .vAddressBlock
LAUNCH TestDoc.DOC
  or
LAUNCH TestDoc.WPD
```

Once the MS Word or WordPerfect is launched and the document is opened, you could either use Edit > Paste or Ctrl+V to paste the windows clipboard text!

The resulting pasted block of text would look like:

Pc Distribution Inc.
3200 Westminster Way
Boston, MA 02178

6.15.6.19 CMPAUSE

Operating Condition

Syntax: SET CMPAUSE ON/OFF

Default: OFF

CMPAUSE (Cascade Modal Pause) determines if R:BASE will use a local PAUSE dialog window for modal PAUSE displays instead of the global PAUSE form. This means multiple modal PAUSE dialog windows will appear on top of each other instead of prematurely closing the current modal PAUSE form.

6.15.6.20 COLOR

Operating Condition (**R:BASE for DOS ONLY**)

```
Syntax:  SET COLOR
         SET COLOR FOREGRND color
         SET COLOR BACKGRND color
         SET COLOR BACKGRND (redvalue, greenvalue, bluevalue)
```

You can specify foreground and background colors for the DOS R> Prompt window. You can alter the R:BASE for Windows R> Prompt background and font color by adjusting the settings available from the main Menu Bar under "Settings" > "R> Prompt".

You can also specify colors using the SET command. For example, to change the background color to cyan, enter:

```
SET COLOR BACKGRND cyan
```

You can even select a custom color for the background using a combination of red, blue, and green values. For example, to change the background to orange, enter:

```
SET COLOR BACKGRND (255, 128, 64)
```

6.15.6.21 COMPATIB

Operating Condition

Syntax: SET COMPATIB ON/OFF

Default: ON

Compatibility with R:BASE Transactions

SET COMPATIB toggles COMPATIBILITY with R:BASE transactions on and off.

The first "compatibility" setting in R:BASE goes back to the 3.1 versions where it was used to allow concurrent access to a database from both 3.1 and 2.11 at the same time. Remember that database files back then were still "rbf" files. Once we went to the "rb1,rb2,..." files the setting was obsolete.

In 1997 when Oterro was first released, it supported a different scheme for managing transactions (when TRANSACTIONS are set on) than R:BASE itself used at that time. This new scheme used a different file to track transactions and had a larger allocated buffer size on the file to manage ongoing transactions. To make R:BASE compatible with Oterro when transactions were on it needed to support the new method, but it also needed to be able to support the older style that previous versions of R:BASE used (version 6.0 and older).

The first R:BASE version that could support the two methods was 6.1. When the "compatibility" setting is on, R:BASE will use the older "non-Oterro" style of transaction. When "compatibility" is off, the Oterro method will be used. If you never run with TRANSACTIONS ON then the compatibility setting does not have any effect.

6.15.6.22 CURRENCY

Operating Condition

Syntax: SET CURRENCY $ PREF 2 B

SET CURRENCY sets the symbol, location, subunits, and format for currency values. R:BASE stores the setting with the database.

Changing the CURRENCY parameters affects all columns in the database that have a CURRENCY data type. You must enter the parameters in the following order: SYMBOL, PREF or SUFF, digits, and format; that is, even if you want to change the digits only, you must also enter the symbol and its position.

- **Symbol** (default $)--A symbol is any ASCII character or string of one to four characters. You can include a space in place of one character at the beginning or end of the string, but if you do, enclose the string in quotation marks.

- **PREF and SUFF** (default PREF)--Specify the position of the symbol as before (PREF) or after (SUFF) the currency value. In the SET CURRENCY command, enter a space between the symbol and its position, PREF or SUFF.

- **Subunit digits** (default 2)--Indicates the number of digits from 0 to 16 to be displayed in a currency subunit. In the case of dollars, the subunit is cents, so the digits setting for dollars is 2. For example, setting digits to 3 will display currency values similar to these:

  ```
  20.000,000DM
  20,000.000
  2,000.000
  ```

  If you change digits when the database contains data, the new digit setting affects how R:BASE displays and uses the data already entered. For example, if you change the setting from 2 to 4, an existing value such as 1,234.00 becomes 12.3400.

- **Format** (default B)--Format specifies how R:BASE displays the thousands and decimal delimiters. *A*, *B*, and *C* specify how the thousands and decimal delimiter displays for values with CURRENCY, REAL, and DOUBLE data types. Before you change the format, you must change the DELIMIT character.

  Delimiter Conventions for CURRENCY, REAL, and DOUBLE Values

Convention	Thousands Delimiter	Decimal Delimiter	Example
A	.	,	123.456.793,01
B	,	.	123,456,793.01
C	(blank)	,	123 456 793,01
D	N/A	.	123456793.01

  For example, to display currency in two digits with a prefix of DM (deutsche marks) with the display format *A*, at the R> Prompt, enter:

```
SET DELIMIT=!
SET CURRENCY DM PREF 2 A
```

This command displays 1,500 deutsche marks and 25 pfennigs in the format like this:

```
DM 1.500,25.
```

If you set CURRENCY to the delimiter formats A or C without changing DELIMIT to a character other than the comma, the default setting for DELIMIT, R:BASE displays the following message: "Decimal character cannot be the same as DELIMIT."

In this case, use SET DELIMIT to change the delimiter to a less commonly used character such as an exclamation point; then set CURRENCY to the delimiter formats *A* or *C*.

6.15.6.23 DATE

Operating Condition

```
Syntax:  SET DATE CENTURY value
         SET DATE YEAR value
         SET DATE MM/DD/YY (date sequence and format)
         SET DATE SEQ MMDDYY (date sequence)
         SET DATE FOR MM/DD/YY (date format)
```

SET DATE sets the date sequence for entry, and format for display. A valid date can have up to 30 characters. R:BASE stores the setting with the database.

Use the SET DATE CENTURY *value* command to set the default century (the first two digits of a four-digit year). For example, if you enter a two-digit year and you want it to default to the twenty-first century, enter the following command:

```
SET DATE CENTURY 20
```

A year such as "25" would be stored as "2025."

Use the SET DATE YEAR *value* command to have two default centuries for dates entered, depending on the year. All years from 00 to (YEAR *value*-1) are stored with the next century (CENTURY *value*+1), and all years from *value* to 99 are stored with the default century (CENTURY *value*). For example, you can have all dates from the year 50 to 99 default to the twentieth century, and all dates from 00 to 49 default to the twenty-first century by entering the following commands:

```
SET DATE CENTURY 19
SET DATE YEAR 50
```

The SET DATE CENTURY 19 command sets the default century to 19. The SET DATE YEAR 50 command stores all years from 50 to 99 with the default century, 19. All dates from 0 to 49 are stored with the next century, 20. Therefore, years entered from 50 to 99 are stored as 1950 to 1999, and years entered from 00 to 49 are stored as 2000 to 2049.

Note: The DATE CENTURY and DATE YEAR options are only effective when the DATE SEQUENCE includes a two-digit year (MMDDYY, DDMMYY, etc.).

R:BASE accepts a date between January 1, 3999 BC and December 31, 9999 AD. You can set the date sequence and format separately. R:BASE displays the date based on the format. When setting the format to display numerals for the month, day, and year, use a separator such as the slash (/), hyphen (-), comma (,) or space (blank). For example, if you set the date format to MM/DD/YY and enter 061193, R:BASE displays 06/11/93.

You can also include text for the weekday and month in the date format to a maximum of 30 characters. Include WWW for a three-letter day abbreviation, WWW+ for the full day name, MMM for a three-letter month abbreviation, and MMM+ for the full month name. If the date format contains spaces or commas, enclose the format in quotes. For example, if the special character for QUOTES is set to the R:BASE default ('), the format 'MMM DD, YYYY CC' displays Jun 11, 1993 AD.

If you use YY in the date format, R:BASE displays only the last two digits of the year. To view dates in other centuries, use a date format with a four-digit year such as 'MM DD, YYYY'. If you use BC dates, add CC to the format. Dates entered with BC are shown with BC; otherwise, the date is shown with AD. For example, you could use the sequence MMDDYY and the format 'MMM DD, YYYY CC' to accept and display BC dates. If you enter '06 11 93BC,' R:BASE displays Jun 11, 0093 BC.

Example: Valid date formats, using June 11, 1993

Date Format	Display
'MMM+ DD' MM/YY	June 11 06/93
'WWW the DD' 'WWW+, MMM+ DD, YYYY CC'	Sun the 11 Sunday, June 11, 1993 AD

You can omit the SEQ and FOR keywords to set both sequence and format in a single SET DATE command. For example, enter SET DATE MM/DD/YYYY to set both date sequence and format to a four-digit year.

Enter the date in any form as long as the sequence of M's, D's, Y's, and C's are in the same order defined for the date sequence. The display, however, is always exactly as defined by the DATE format.

If, for example, you set the date sequence to a four-digit year with SET DATE SEQ MMDDYYYY, set the date format to a two-digit year with SET DATE FOR MM/DD/YY, and later enter a two-digit year, R:BASE will store and might display a date you do not expect. As the following table shows, if you enter 06/11/93, R:BASE stores the date as 06/11/0093 and displays 06/11/93.

How R:BASE stores and displays dates

DATE Sequence	Date Entered	Date Stored
MMDDYYYY	6/11/94	06/11/0094
MMDDYY	6/11/0094	06/11/1994
MMYY	6/11	06/01/1994
MMDD	6/11	06/11/1994
DDYY	6/11	01/06/1911
DDYY	11/94	01/11/1994

DATE Format (2-digit Year)	DATE Format (4-digit Year)
06/11/94	06/11/0094
06/11/94	06/11/1994
06/01/94	06/11/1994
06/11/94	06/11/1994
01/06/11	01/06/1911
01/11/94	01/11/1994

If the sequence is set to a four-digit year and the format is set to a two-digit year, R:BASE stores the date you enter, such as 06/11/95, as a four-digit year. As a result, if you use a WHERE clause to display rows that have dates greater than 06/11/95, R:BASE returns all rows greater than 06/11/0095.

The DATE format can affect date functions. For best results, set the format to the default MM/DD/YY and then use a date function.

6.15.6.24 DEBUG

Operating Condition

Syntax: SET DEBUG ON/OFF

Default: OFF

You can use SET DEBUG as follows:

1. Precede any R:BASE command you want to control with the DEBUG modifier in a command file.

2. Set DEBUG on to have R:BASE run the command; set DEBUG off to have R:BASE ignore the command.

6.15.6.25 DELIMIT

Special Character

Separates a character, string, or items in a list used in commands. Also used to separate repeatable parts of a command.

Syntax: SET DELIMIT=NULL
SET DELIMIT=char
(Use NULL to disable the special character.)

Default: ,

6.15.6.26 ECHO

Operating Condition

Syntax: SET ECHO ON/OFF

Default: OFF

SET ECHO displays or suppresses commands as they are processed from the current ASCII input device. Specify on or ECHO to display commands; specify ECHO off or NOECHO to turn off the command display.

Use SET ECHO as a debugging technique as you develop a command file. With ECHO set on, you can see the commands as they are processed when you run a command file. SET ECHO works only when the command file is an ASCII file; it will not display commands that were run from a binary procedure file.

Enter a SET ECHO ON command at the beginning of the program. Then, when the program runs, R:BASE displays each command as it is interpreted and, if needed, runs it. Sometimes the commands scroll on the screen faster than you can read, especially if they are read by R:BASE but not run. You can temporarily stop the display by pressing [Ctrl]+[Break], which stops the file from running, and restart it by pressing [Enter], or stop completely by pressing [Esc]. Set ECHO off to suppress command display.

Even more useful, you can direct output to a printer or a file before you set ECHO to on. Then, when the command file runs, the commands and any errors are saved either in printed form or in a file you can look at.

When R:BASE runs the commands below in an ASCII file, you see the commands on lines two, three, and four displayed at the top of the screen and the message displayed on line 10 beginning at column 20.

```
CLS
SET ECHO ON
WRITE 'This is a message' at 10 20
SET ECHO OFF
```

6.15.6.27 EDITOR

Environment Setting

Syntax: SET EDITOR RBEDIT/filespec

Default: RBEDIT

The SET EDITOR setting allows you to specify the internal R:BASE Editor or some other text editor as your default text editor for R:BASE command files.

For example, if you wish to alter the default text editor to the external R:BASE Editor program, you would use the following syntax:

```
SET EDITOR C:\RBTI\RBEdit\RBEdit.exe
```

6.15.6.28 EOFCHAR

Operating Condition

Syntax: SET EOFCHAR ON/OFF

Default: ON

If EOFCHAR is set OFF, a control-Z character will not be appended to the end of output files.

6.15.6.29 EQNULL

Operating Condition

Syntax: SET EQNULL ON/OFF

Default: OFF

This Command determines whether or not NULL = NULL.

Compare these code samples:

```
SET VAR v1 TEXT = NULL
SET VAR v2 TEXT = NULL


SET EQNULL OFF
IF v1 = .v2 THEN
  -- will not be a hit
ENDIF
IF v1 <> .v2 THEN
   -- will not be a hit
ENDIF
IF v1 <> 'This' THEN
   -- will not be a hit (it used to be before this fix)
ENDIF


SET EQNULL ON
IF v1 = .v2 THEN
   -- will be a hit
ENDIF
IF v1 <> .v2 THEN
   -- will not be a hit
ENDIF
IF v1 <> 'This' THEN
   -- will be a hit
ENDIF
```

Before this fix, the comparison "IF v1 <> 'This' THEN" would be a hit with EQNULL set ON or FALSE when it should only be a hit when EQNULL is ON. This means that now "IF (.v1) <> 'This' THEN" and "IF v1 <> 'This' THEN" will both process the same way. In the past they would be different because of this problem.

In your code if you want the comparison of a NULL variable and a non-NULL constant to be a hit then you should run with EQNULL set ON.

6.15.6.30 ERROR MESSAGE

Special Condition

Syntax: SET ERROR MESSAGE Error# ON/OFF

Default: ON

SET ERROR MESSAGE Error# ON/OFF displays or suppresses a particular error message when a system error occurs.

You can selectively turn OFF any -ERROR- message(s) in your command file (very handy for debugging) by doing the following:

```
SET ERROR MESSAGE Error# OFF
```

To turn it back ON:

```
SET ERROR MESSAGE Error# ON
```

For example, to not see the:

-WARNING- No rows exist or the specified clause.

You can do the following:

```
SET ERROR MESSAGE 2059 OFF
```

This new feature has a limit of 50 -ERROR- numbers to set OFF and each one requires a separate command.

NOTE: Each turned OFF message must be turned back ON before turning it OFF again. If not, you'll get the error message.

6.15.6.31 ERROR MESSAGES

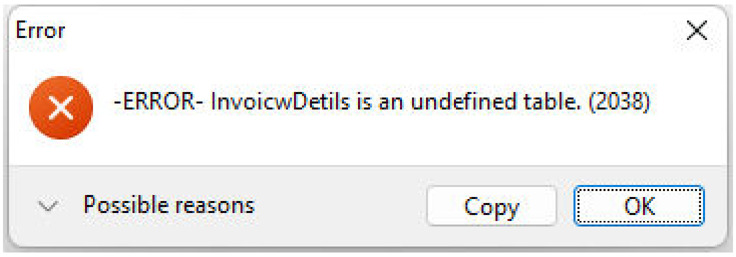
Operating Condition

Syntax: SET ERROR MESSAGES ON/OFF

Default: ON

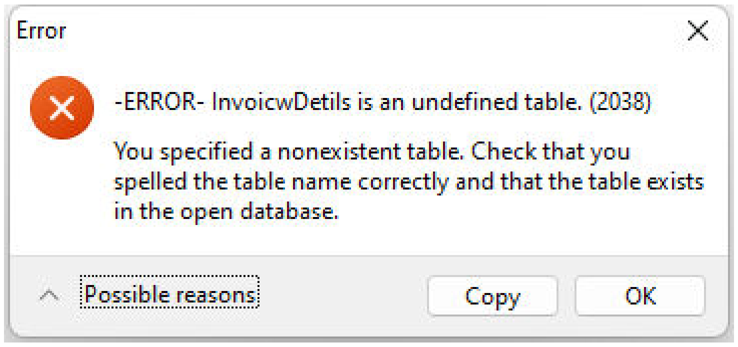
SET ERROR MESSAGES displays or suppresses an error message when a system error occurs. Specify off to suppress error and rule violation messages.

Error messages can also be suppressed after an initial error occurs. To do so, place a check within the "Suppress Error Messages" check box. After doing so, error messages will not be displayed for the instance of R:BASE until the program is restarted, or if the ERROR MESSAGES operating condition is set ON.


Error

-ERROR- InvoicwDetils is an undefined table. (2038)

The Error Message dialog contains "Help" button to display possible reasons for the error, and a "Copy" button to capture the error.


Error

-ERROR- InvoicwDetils is an undefined table. (2038)

You specified a nonexistent table. Check that you spelled the table name correctly and that the table exists in the open database.

6.15.6.32 ERROR VARIABLE

Operating Condition

Syntax: SET ERROR VARIABLE varname
          SET ERROR VARIABLE OFF

SET ERROR VARIABLE defines an error variable to hold error message numbers. The variable name (*varname*) defines the variable R:BASE uses to hold R:BASE error codes. If set to off (the default), error variable processing is removed.

When an error occurs in a command file, R:BASE normally displays a system error message. SET ERROR enables a programmer to anticipate errors in command and procedure files and program the file to keep running even when an error occurs.
You must always set ERROR VARIABLE off, rather than clearing it with the CLEAR VARIABLES command.

R:BASE resets the error variable to zero as each command is successfully run. If an error occurs, the error variable is set to the error number value. To determine the error condition for any line, you must immediately check the value of the error variable or capture the error value in a global variable for later examination.

By checking the error variable for a non-zero value, you can detect (or trap) many errors and run a sequence of error-handling commands such as an error-recovery procedure. Once the error number is captured in an error variable, you can write error-handling command files to control a program's flow based on these errors (error values).

The error variable value is set for each command that is run, not each line in a command file. If you have placed multiple commands on a line, the last command's error value is placed in the error variable. A similar situation occurs for multi-line commands such as the subcommands you can use when loading a data block with the LOAD command. For example, a data block loaded with the LOAD command leaves the error variable with a value of zero because the END command runs successfully, whether or not the data is actually loaded.

Rule violations do not set the error variable to a non-zero value; they are not the same as errors recognized by R:BASE.

The command below defines *errvar* as the current error variable:

```
SET ERROR VARIABLE errvar
```

When a command is run, R:BASE sets the error variable *errvar* to the error code before anything else happens. The following command lines illustrate how to use *errvar* in a command file.

```
LABEL tryagain
DIALOG 'Enter the database name:' vdbname vendkey 1
```

```
CONNECT .vdbname
IF errvar <> 0 THEN
   WRITE 'Database not found.'
      GOTO tryagain
ENDIF
```

The first command establishes a label to return to, the second requests that the user enter the name of a database, and the third opens the specified database using the global variable defined by the FILLIN command.

The IF...ENDIF structure checks the error variable value. If the value is not zero (that is, if the database was not opened successfully), then it sends a message to the screen and passes control to the label *tryagain* so that the user is asked to enter the database name again.

You can also write a separate command file specifically designed to handle a variety of errors. In this case, the above code might look like this:

```
DIALOG 'Enter the database name:' vdbname vendkey 1
CONNECT .vdbname
SET VARIABLE verr1 = .errvar
IF verr1 <> 0 THEN
       RUN errhandl.cmd USING .verr1
ENDIF
```

This series of commands captures the error value in the global variable *verr1* so that it can be passed through the USING clause of the RUN command to an error-handling routine. The routine itself determines the nature of the error and how to take care of the problem.

You can use the WHENEVER command to run status-checking routines for SQL commands. WHENEVER uses the special R:BASE variable SQLCODE.

6.15.6.33 ESCAPE

Operating Condition

Syntax: SET ESCAPE ON/OFF

Default: ON

SET ESCAPE allows you to use [Ctrl]+[Break] to escape or abort command file processing or database file access. Specify on to enable users to abort processing in the middle of command files, WHILE loops, and database access. Specify off to prevent users from prematurely aborting a command file or an application such as when R:BASE runs processes that create new tables (such as the relational commands) from within a command.

6.15.6.34 EXPLODE

Operating Condition (**R:BASE for DOS only**)

Syntax: SET EXPLODE ON/OFF

Default: OFF

Controls how DOS dialogs are displayed.

When EXPLODE is set on, dialog boxes are displayed in full size instantly. When EXPLODE is set off, dialog boxes are displayed in an expanding fashion from the center.

6.15.6.35 FASTFK

Operating Condition

Syntax: SET FASTFK ON/OFF

Default: OFF

This setting, when on, permits R:BASE to operate a foreign key index using a condensed index for maintaining that foreign key. If the foreign key is not used for retrieving data or linking columns, a complete index is unnecessary and actually inhibits speed. When set to on, R:BASE creates a condensed index for any existing foreign keys.

To switch to condensed indexes on existing foreign keys, you need to run a PACK, PACK KEYS, or RELOAD command with FASTFK on; these actions cause R:BASE to rebuild the database with condensed foreign key indexes.

Keep in mind, however, that you might need complete indexes on foreign keys where such indexes are needed for retrieving data. Retaining a separate index on columns used in foreign keys that link tables is preferred. Indexes are also needed on foreign keys that you use for selecting column values; therefore, use the CREATE INDEX command to explicitly create indexes for columns used in foreign keys in a database where FASTFK is set on.

The command SHOW FASTFK displays the FASTFK state and whether FASTFK is operational in the current database. For example:

```
SHOW FASTFK
(FASTFK ) ON Use fast Foreign Key (FK) structures on rebuild.
OFF FASTFK setting for current database
```

Once you rebuild the keys in a database with the FASTFK setting on, SHOW FASTFK displays the following:

```
SHOW FASTFK
(FASTFK ) ON Use fast Foreign Key (FK) structures on rebuild
ON FASTFK setting for current database
```

6.15.6.36 FASTLOCK

Operating Condition

Syntax: SET FASTLOCK ON/OFF

Mode: Multi-user and STATICDB

Set FASTLOCK on for faster multi-user performance while modifying data. With FASTLOCK on, R:BASE does not place a table lock on the table, allowing for greater throughput. A table lock is only needed to prevent structure changes.

FASTLOCK can only be set on when STATICDB is set on, and both FASTLOCK and STATICDB must be set on before the database is connected. Like other R:BASE database modes (SET MULTI and SET STATICDB), FASTLOCK requires all users to be connected with the same setting.

The following command lines set STATICDB and FASTLOCK correctly.

```
SET STATICDB ON
SET FASTLOCK ON
CONNECT concomp
```

6.15.6.37 FEEDBACK

Operating Condition

Syntax: SET FEEDBACK ON/OFF

Default: OFF

This setting displays processing results when either calculating or editing rows.

With this setting on, R:BASE displays the number of rows processed and the elapsed time to completion in a dialog window. Displays occur while in the Data Browser, when printing reports and labels, and when using certain R:BASE commands including:

1. ALTER TABLE
2. AUTONUM
3. CREATE INDEX
4. DELETE
5. INSERT
6. LOAD
7. The data transfer for a PROJECT command
8. SELECT
9. SORTing a large record set
10. UNLOAD

When using any of the various commands at the R> Prompt with FEEDBACK set ON, the FEEDBACK system variables are generated to hold the processed row count and elapsed time for the command.

```
RBTI_RowsInserted        = 101010          INTEGER
RBTI_RowsDeleted         = 0               INTEGER
RBTI_RowsUpdated         = 0               INTEGER
RBTI_ElapsedTime         = 0:00:01.468     TEXT
```

6.15.6.38 FILES

Operating Condition

Syntax: SET FILES value

Range: 1 to 255 files

Default: 5

SET FILES sets the maximum number of files that can be open at a time. The maximum, depending on available memory, is 255.

6.15.6.39 FIXED

Operating Condition

Syntax: SET FIXED ON/OFF

Default: ON

Controls column width in SELECT

This controls whether R:BASE will automatically shrink column widths in SELECT commands.

6.15.6.40 FONT

Operating Condition (**R:BASE for DOS ONLY**)

Syntax: SET FONT keyword

Default: OEM

The FONT setting changes the font used in the "R> Prompt" window.

You can choose from three settings, which are mono-spaced stock fonts used in Windows:

- System--fonts compatible with the system font in Windows.
- OEM--an IBM PC character set for IBM computers.
- Ansi--a fixed-pitch font based on the Windows character set. A Courier font is typically used.

6.15.6.41 HEADINGS

Operating Condition

Syntax: SET HEADINGS ON/OFF

Default: ON

SET HEADINGS displays columns with or without headings when you enter the SELECT and TALLY commands.

6.15.6.42 IDQUOTES

Operating Condition

Syntax: SET IDQUOTES

Default: Reverse Quote (\`) Prior Versions: NULL

Controls the character that is used to set off object names.

IDQUOTES sets the character for enclosing object names, such as column or table names, in R:BASE and ODBC commands. This is especially critical when using ODBC to connect to a non-R:BASE Database that allows characters such as spaces in Column or Table names. This is also critical to SOME internal R:BASE processing.

Note: Older databases may default to NULL which is not ODBC or SQL compliant.

In general, setting IDQUOTES will have no effect on legacy applications as they will not, in most cases, know that this exists. The one exception to this is unloading data from a database that has IDQUOTES set, such as an current upgraded R:BASE database, and importing into a Legacy database. If you are attempting to do this you should set your IDQUOTES to NULL.

The Reverse Quote is located, on most standard US Keyboards, under the Tilde (~) character and to the left of the numeral 1.

6.15.6.43 INDEXONLY

Operating Condition

Syntax: SET INDEXONLY ON/OFF

Default: ON

Sets a flag to disable "index only" select retrievals.

6.15.6.44 INSERT

Operating Condition

Syntax: SET INSERT ON/OFF

Default: ON

SET INSERT turns insert/overwrite on or off. Set INSERT on to use either the insert or overwrite mode. Pressing the [Insert] key when you have specified INSERT to be on toggles you between insert mode and overwrite mode. In insert mode, the cursor indicator is larger. Press the space bar to insert a space. Set INSERT off to use only the overwrite mode; the cursor indicator is smaller.

6.15.6.45 INTERVAL

Operating Condition

Syntax: SET INTERVAL value

Default: 5

Range: 0 to 9 tenths of a second

Mode: Multi-user

The SET INTERVAL command specifies the time to elapse before R:BASE retries the command that caused a conflict within the waiting period. Also, see WAIT.

6.15.6.46 KEYMAP

Operating Condition

Syntax: SET KEYMAP keyname OFF
SET KEYMAP ALL OFF
SET KEYMAP keyname=remapped keys

You can define key maps with a single statement, which enables you to define key maps in command files. For example:

```
SET KEYMAP [SHIFT][F3] TO [F2]
SET KEYMAP [ALT]M= [F2]
```

To return to the original key mapping, enter the following:

```
SET KEYMAP [ALT]M OFF
```

6.15.6.47 LAYOUT

Operating Condition

Syntax: SET LAYOUT ON/OFF

Default: ON

SET LAYOUT switches saving layouts on or off. When LAYOUT is set on, R:BASE saves the layout of data displayed in the Data Browser when you exit. R:BASE saves layouts for single tables only. The next time you display the table with the Data Browser, the layout of data will be as you previously arranged it.

When LAYOUT is set off, R:BASE does not save the layout of data and ignores any saved layouts. If you want to see a table displayed in its default format without changing the saved layout for it, set LAYOUT off before displaying the table.

6.15.6.48 LINEEND

Special Character

Syntax: SET LINEEND = value

DOS Default: þ [Alt]+[0254]

Windows Default: ^

You can set an end of line ASCII character for NOTE and TEXT fields in forms, reports and the Data Browser. When you insert the character in those fields and then zoom in by pressing [SHIFT]+[F2] or print a report, you see lines break as established by the line end character. The default character for R:BASE database that migrated over the years from DOS versions is þ, the ASCII value 0254. The default character for R:BASE for Windows is the carat (^).

Note: If the line end character has been set to the currency character, R:BASE changes it to ASCII value 0254 when you connect to a database.

6.15.6.49 LINES

Operating Condition

Syntax: SET LINES value

Range: 0 to 32,767 lines

Default: 20

SET LINES sets the number of lines per page or screen when you use the CROSSTAB, DISPLAY, DIR, LIST, OUTPUT, SELECT, LIST RULES, SHOW VARIABLES, TALLY, or TYPE commands. LINES does not affect report generation; you can define the number of lines on a page for each report. Setting LINES to zero displays lines as continuous output.

6.15.6.50 LOCK

Operating Condition

Syntax: SET LOCK tbllist ON/OFF

Default: OFF

Mode: Multi-user

SET LOCK manually sets locks on or removes locks from tables specified in the list of tables (*tbllist*). Use the command whenever you want a procedure or transaction to have exclusive use of tables. Setting LOCK to off disables locks for each of the tables in *tbllist*.

In command or procedure files it is sometimes necessary to prevent access to a table or group of tables while certain operations are performed. Although R:BASE handles most locks automatically according to the command that is running, at times you might want more control over table locking. SET LOCK provides you explicit control over access to tables during processing by the commands that retrieve and update data.

If R:BASE cannot lock all the tables listed after SET LOCK, it issues a message saying that not all tables are available to be locked. R:BASE does not lock any tables unless it can lock all tables listed, and it sets an error code when SET LOCK fails.

Group the tables used into one SET LOCK command to avoid tying up needed resources. Be sure to issue the SET LOCK OFF command to remove the locks after processing is complete. Locks set with this command are cumulative. You need to issue one SET LOCK OFF command for each SET LOCK ON command that you have entered for a given table. The user who set the table locks must issue SET LOCK OFF; otherwise no other user can access the locked tables until the first user exits the database.

LIST displays locked tables in reverse video. With LIST TABLE, on the other hand, the type of multi-user locks is displayed. Only the highest priority lock is displayed for each table.

LIST TABLE tells you whether the lock is an edit, row, cursor, local, or remote lock. Edit, row, and cursor locks are set by R:BASE as part of its internal concurrency control. A local lock is set by a SET LOCK command issued at the workstation that issued the LIST TABLE command. And a remote lock is set by a command that obtains a table lock and is issued from a workstation other than the workstation that issued the LIST TABLE command.

The first command line below sets an exclusive lock on the *customer* table. The second command line sets additional exclusive locks on the *transmaster* and *transdetail* tables. These exclusive locks prevent access to the three tables by any user other than the one who issued the SET LOCK ON commands. The SET LOCK OFF command removes the locks on all three tables.

```
        SET LOCK customer ON
        SET LOCK transmaster, transdetail ON
        SET LOCK customer, transmaster, transdetail OFF
```

6.15.6.51 LOOKUP

Operating Condition

Syntax: SET LOOKUP value

Default: 5

SET LOOKUP tells R:BASE how many form look-up expressions to store in memory. Storing a look-up expression in memory enables R:BASE to display data more quickly in a form. The number of look-up expressions you can specify depends on the memory available in your computer. R:BASE needs approximately 500 bytes of RAM for each look-up expression. SET LOOKUP does not affect master look-up expressions or pop-up menus in a form.

For example, when your form contains 10 look-up expressions and LOOKUP is set to 5, only the first five look-up expressions are stored in memory. R:BASE must retrieve/reevaluate the remaining look-up expressions. So that the form works faster, you can set LOOKUP to a higher value to store more look-up expressions in memory.

6.15.6.52 MANOPT

Operating Condition

Syntax: SET MANOPT ON/OFF

Default: OFF

MANOPT set to OFF disables the automatic table-order optimization that R:BASE performs when running queries. This gives maximum control over the order in which columns and tables are assembled in response to a query.

With MANOPT set to ON, R:BASE uses the order of the tables in the FROM clause and the order of the columns in the column list of the SELECT clause to construct the query.

When MANOPT is set on, the **#TABLEORDER** system variable stores the table join order, and the applicable indexed columns. To display the table order, use the following:

```
PAUSE 2 USING .#TABLEORDER
```

6.15.6.53 MANY

Special Character

Sets the character for the many wildcard for R:BASE commands and clauses.

Syntax: SET MANY=NULL
SET MANY=char
(Use NULL to disable the special character.)

Default: %

6.15.6.54 MAXTRANS

Operating Condition

Syntax: SET MAXTRANS value

Range: 1 to 1295

Default: 201

Mode: Transaction Processing

SET MAXTRANS specifies the maximum number of users who can have the same database open concurrently with transaction processing on.

Only the first user to connect to a closed database can enter the MAXTRANS setting for that database. Enter the command before connecting to the database. If anyone else already has the database open, R:BASE displays a message telling you that your SET MAXTRANS command will have no effect on the database.

MAXTRANS can affect system performance. The higher the MAXTRANS setting, the more overhead the system must carry to process transactions across the network. Also, the more users who are actually entering transactions, the slower the system operates.

6.15.6.55 MESSAGES

Operating Condition

Syntax: SET MESSAGES ON/OFF

Default: ON

SET MESSAGES either displays or suppresses system messages. Set MESSAGES to off when ERROR is set on to display only error messages.

6.15.6.56 MIRROR

Operating Condition

Syntax: SET MIRROR <path>

Syntax: SET MIRROR OFF/DELETE

Default: OFF

SET MIRROR <path> maintains a duplicate copy of the database. This duplicate copy is created and maintained in the directory designated in *path.* The duplicate database will have the same name, therefore *path* must designate a backup directory. With this setting, all modifcations to the original database are duplicated in the mirrored database.Be sure all users are mapped to the save drive letters. SET MIRROR OFF turns off mirroring of the database; SET MIRROR DELETE turns off mirroring and then deletes the duplicate database.

6.15.6.57 MOUSE

Operating Condition (**R:BASE for DOS ONLY**)

Syntax: SET MOUSE -1 to 100

Default: 30

Controls how DOS dialogs are displayed.

When EXPLODE is set on, dialog boxes are displayed in full size instantly. When EXPLODE is set off, dialog boxes are displayed in an expanding fashion from the center.

Controls mouse sensitivity

Used with R:BASE for DOS only. SET MOUSE controls the period of time in hundreths of a second in which the mouse registers a double click. Setting the time too low makes it impossible to double-click the mouse. A setting of -1 disables the mouse. To set the mouse speed each time you use R:BASE, include the SET MOUSE command in a startup file.

6.15.6.58 MULTI

Operating Condition

Syntax: SET MULTI ON/OFF

Default: OFF

SET MULTI sets Multi-User capability on or off when you next connect a database. This setting must be used while you are disconnected from a database.

6.15.6.59 NAME

Operating Condition

Syntax: SET NAME network identification

Default: USER************** (USER and 14 numerals for date and time)

Mode: Multi-user

SET NAME specifies a network identification for your system when you start R:BASE. NAME is text and can contain spaces.

NAME must be saved to the configuration file.

6.15.6.60 NAMEWIDTH

Operating Condition

Syntax: SET NAMEWIDTH value

Range: 4 to 128 characters

Default: 18

SET NAMEWIDTH controls the name width of a table, column, form, report, label, etc. that R:BASE directs to the printer, screen, or file when using the SELECT and UNLOAD commands. The defined width value specifically controls the number of characters for the first column in the displayed list, whether it is the first column for a SELECT command, or the table names in a LIST TABLES command. Use the WIDTH setting to control the number of characters for the entire row/line of data.

Do not set the width to a number greater than the number of characters your printer can fit on a line; a typical page and computer screen display 80 characters. WIDTH does not affect report generation; each report defines the width of a data line.

The LIST TABLES command produces the below results when NAMEWIDTH is 18 and WIDTH is 79.

```
R>LIST TABLES

   Name                Columns      Rows   Comments
   ------------------- ------- ---------   -----------------------------------
   BonusRate                 3         7   Rates for Bonuses
   Component                 2        12   Component Identification Number and
                                           Description
   CompUsed                  2        22   Components Used in a Model
   Contact                  12        35   Customer Contact Information
   ContactCallNotes          5         3   Contact Call Notes
   Customer                 17        30   Customer Information
   DBAccess                  1         1   Database Access
   Departments               8        31   Departments
   Employee                 19        13   Employee Information
   FormTable                1         1   Dummy Table for Forms
   HourlyTemps              3         3   Temperature Data for Gauge
```

With the NAMEWIDTH value doubled to 36, the "Name" column width is increased. Notice that the "Comments" column is now wrapped.

```
R>SET NAMEWIDTH 36
R>LIST TABLES

   Name                                 Columns      Rows   Comments
   ------------------------------------ ------- ---------   ------------------
   BonusRate                                  3         7   Rates for Bonuses
   Component                                  2        12   Component
                                                            Identification
                                                            Number and
                                                            Description
   CompUsed                                   2        22   Components Used
                                                            in a Model
   Contact                                   12        35   Customer Contact
                                                            Information
   Customer                                  17        30   Customer
                                                            Information
```

Increasing the WIDTH setting will display the "Comments" column without the wrapped characters.

```
R>SET WIDTH 120
R>LIST TABLES

   Name                                 Columns      Rows   Comments
   ------------------------------------ ------- ---------
-----------------------------------------------------------
   BonusRate                                  3         7   Rates for Bonuses
   Component                                  2        12   Component Identification
Number and Description
   CompUsed                                   2        22   Components Used in a
Model
   Contact                                   12        35   Customer Contact
Information
   Customer                                  17        30   Customer Information
   DBAccess                                   1         1   Database Access
   Departments                                8        31   Departments
   Employee                                  19        13   Employee Information
   FormTable                                  1         1   Dummy Table for Forms
   HourlyTemps                                3         3   Temperature Data for
Gauge
```

6.15.6.61 NOCALC

Operating Condition

Syntax: SET NOCALC ON/OFF

Default: OFF

NOCALC suppresses or processes computed column expressions with the UNLOAD and LOAD commands.

If NOCALC is set to ON, then the UNLOAD command, as it creates LOAD statements, will generate new "load only" commands as it starts each table. These "load only" commands would be CALC and NOCALC, which operate in a fashion similar to CHECK/NOCHECK and FILL/NOFILL and NUM/NONUM commands. The UNLOAD command will output the current values for the computed columns as it unloads the data.

In processing a LOAD command for a table, if it encounters a NOCALC command, then regardless of the current setting for NOCALC, the LOAD command will expect values for every column, whether computed or not. The values from the input would then be stored in those columns that are computed, rather than doing the actual computation.

NOCALC allow users to preserve original computed values when using UNLOAD/LOAD to move data, or when rebuilding a database with UNLOAD ALL.

Other methods for adding rows to a table, such as APPEND , the Data Browser, or a form, would all still calculate each computed column. Only the LOAD command with its special CALC/NOCALC commands could input a value into a computed column without doing the computation.

R:BASE stores the NOCALC setting with the database.

6.15.6.62 NOTE_PAD

Operating Condition

Syntax: SET NOTE_PAD

Range: 0 to 100 (percent)

Default: 10

This setting allocates an additional percentage of storage space in NOTE columns to accommodate value increases (additional text), so that rows don't need to move to different disk locations. This increases performance by reducing disk reading.

The default setting is 10% of the row size, and can be set from 0 to 100%. Setting NOTE_PAD to 0% disables padding.

6.15.6.63 NULL

Special Character

Syntax: SET NULL -0- (1 to 4 characters)

Default: -0-

SET NULL sets the display symbol for null. You can use up to four characters. If you set null to a blank space, R:BASE does not display rows composed entirely of nulls with the SELECT command. R:BASE stores the setting with the database.

If you enter the following command, R:BASE displays a slash in the absence of data:

```
SET NULL /
```

6.15.6.64 ONELINE

Operating Condition

Syntax: SET ONELINE ON/OFF

Default: OFF

Controls text and note field wrapping.

When set to ON NOTE and TEXT fields will never wrap to the next line in Reports and SELECT commands. Instead they will be truncated at the end of the column.

6.15.6.65 OTDEBUG

The Oterro Engine OTDEBUG setting is available to create a log file to help understand possible issues when running commands and using ODBC with foreign data sources. The log file will contain Engine Functions as Oterro establishes and frees connections, executes SQL statements, retrieves data and values, controls transactions, and handles data and values.

To enable debugging, add the OTDEBUG setting to your Oterro product configuration file, which creates a log file of the Oterro Engine Functions.

The following provides the supported use of the OTDEBUG setting in the configuration file:

01. - Debugging is off

```
OTDEBUG OFF
```

02. - Debugging is on, where a log file is created in C:\

```
OTDEBUG ON
```

03. - Debugging is on with a file path and name. For the below, the ot_engine.log file is created in the C:\Temp\OTDEBUG\ folder.

```
OTDEBUG ON C:\Temp\OTDEBUG\ot_engine.log
```

After the OTDEBUG setting has been turned ON or OFF, the program instance using the Oterro connection must be restarted in order for the setting to be recognized.

Important: The debug setting and logging adds overhead to the Oterro engine and performance will decrease. After logging has been captured for a desired event where an issue occurs, the debug setting should be set to OFF in the configuration file.

When OTDEBUG is OFF the log file may remain in the configuration file.

```
OTDEBUG OFF C:\Temp\OTDEBUG\rbengine.log
```

The default location for the Oterro configuration files is in "C:\Users\Public\RBTI", with OTERRO11.CFG used for Oterro 11.

6.15.6.66 PAGELOCK

Operating Condition

Syntax: SET PAGELOCK ON/OFF

Mode: Multi-user

Default: ON

PAGELOCK specifies how R:BASE locks data when updating and deleting rows.

The settings for PAGELOCK are:

- **ON -** R:BASE uses page locking or row locking as appropriate. When PAGELOCK is ON and two or more users are updating rows within the same page of data, R:BASE only lets the first user update rows--the other users are locked out until the first user's update has been completed.
- **OFF -** R:BASE uses a fast row-locking method where only row locking is applied with no page locking. When PAGELOCK is OFF, you can lock rows of data instead of locking a page of data. You increase multi-user performance when PAGELOCK is OFF. And even more so when STATICDB and FASTLOCK are on.

If you know that your application mainly updates or deletes data a row at a time, rather than many rows, set PAGELOCK to OFF for row locking. In this case, R:BASE locks a row, reads the row, makes the change, and then releases the row. Otherwise, set PAGELOCK ON for page locking when you are doing an UPDATE and/or DELETE affecting many rows in a table.

Keep in mind that the PAGELOCK setting can be changed dynamically and can be different for different users using the same database.

Technically, the efficient and fastest method for updating data in multi-user environment is to SET STATICDB ON, SET FASTLOCK ON, and SET PAGELOCK OFF. This particular combination will result in the fewest contentions between users.

Notes:

- FASTLOCK and PAGELOCK can be set on at the same time.
- Setting STATICDB and FASTLOCK to ON (in that order), with PAGELOCK set to OFF will significantly increase multi-user performance with individual row changes.
- PAGELOCK is not the same as SET ROWLOCKS.
- Setting the value of PAGELOCK does not change the setting of ROWLOCKS.
- The PAGELOCK setting can be changed dynamically and can be different for different users using the same database.

Example for Testing:

```
 -- The UPDATE must alter the values for may rows
SET FEEDBACK ON
SET PAGELOCK ON -- use page locking
UPDATE <tablename> SET <columnname> = value  -- no WHERE Clause
SET PAGELOCK OFF -- use row locking
SET FEEDBACK OFF
CLS
```

6.15.6.67 PAGEMODE

Operating Condition

Syntax: SET PAGEMODE ON/OFF

Default: OFF

Mode: Single- or Multi-user

Use PAGEMODE to design reports through a custom R:BASE program. With PAGEMODE on, you create a page of a report in memory, then send the report to a printer or file. PAGEMODE is an alternative method to creating reports through the Report Designer; the two methods work in different ways.

The WRITE, SHOW VARIABLE, and SELECT commands are used to "display" data on a virtual page. To determine the row location of the cursor on the virtual page, use the ISTAT function with the keyword PAGEROW after a SHOW VARIABLE command. (ISTAT('pagerow') does not work with the WRITE command.) The DECLARE CURSOR command is usually used for retrieving data for printing. You control form feeds (new pages) in your program by using the NEWPAGE or OUTPUT SCREEN command.

At the beginning of your program, set both SET LINES and SET WIDTH to accommodate the report's size. Then set PAGEMODE to on. You cannot write to a line longer than the current LINES setting or wider than the current WIDTH setting. You also cannot change the LINES and WIDTH setting without setting PAGEMODE to off first.

To send printer control codes to a printer, use the CHAR function to define a variable containing the printer control codes. Then, that variable is sent to the printer using the SHOW VARIABLE or WRITE command. For example,

```
SET VARIABLE vLandscape = (CHAR(27) + CHAR+
                                   (38) + CHAR(108) + CHAR+
                                   (49) + CHAR(79))
     WRITE .vLandscape
```

The printer control codes can be found in the user's manual for your printer. PAGEMODE allows you to:

Create reports wider than 255 columns and longer than 84 lines.

- Produce multi-column reports.
- Design different styles for the pages of the report, such as a report with a personalized letter as the first page.
- Create reports from tables with many-to-many relationships that have two or more detail sections.
- Format odd and even pages differently throughout a report.
- Print headers and footers only on the last page or first page.
- Customize reports so break headers and footers are located on the same page.
- Place a different break header on every page or the same break header on every page.

The following command line sets PAGEMODE on:

```
SET PAGEMODE ON
```

6.15.6.68 PASSTHROUGH

Operating Condition

Syntax: SET PASSTHROUGH ON/OFF

Default: OFF

When PASSTHROUGH is set on, SELECT statements are sent directly to the foreign data source and are not processed by R:BASE. Set PASSTHROUGH on when you use special syntax, such as non-SQL syntax, or syntax that is not supported by R:BASE.

6.15.6.69 PLUS

Special Character

Sets the character for the command line continuation character.

Syntax: SET PLUS=NULL

Syntax: SET PLUS=char
(Use NULL to disable the special character.)

Default: +

6.15.6.70 POSFIXED

Operating Condition (**R:BASE for DOS ONLY**)

Syntax: SET POSFIXED ON/OFF

Default: OFF

Controls how the AT parameter works.

When you use *AT row,col* to position dialog, pause and other windows, the actual position depends on the current font size of the R> Prompt window. If you always want the calculation to use the 8x12 size of the OEM font then set POSFIXED to ON. If you want the dimensions of the current font to be used then set POSFIXED to OFF.

6.15.6.71 PRINTER

Operating Condition (**R:BASE for DOS ONLY**)

Syntax: SET PRINTER printername

Controls the DOS printer

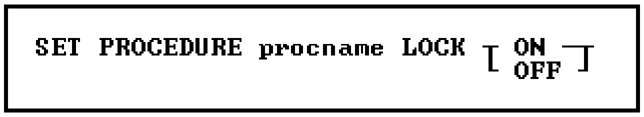
SET PRINTER specifies the printer for your system. The configuration file reads the printer file and sets the printer values as variables for the printer.

```
SET PRINTER epson.prd
```

This command specifies an Epson printer as the printer file named in the configuration file. The extension .PRD is not required.

6.15.6.72 PROCEDURE

The SET PROCEDURE command locks a procedure so it can be replaced.



```
SET PROCEDURE procname LOCK [ ON  ]
                            [ OFF ]
```

Area: Stored Procedures & Triggers

Options

ON
Enables a lock

OFF
Disables a lock set by SET PROCEDURE or GET LOCK.

About the SET PROCEDURE Command

The SET PROCEDURE works like the GET LOCK command without retrieving the Stored Procedure into an ASCII file. ON sets the lock; OFF releases the lock placed by the SET PROCEDURE or the GET command.

When a procedure is locked, only the user placing the lock can replace the procedure or remove the lock. The NAME setting is used for identification of the user.

6.15.6.73 PROGRESS

Operating Condition

Syntax: SET PROGRESS ON/OFF

Default: OFF

This setting displays processing results when building indexes, packing or reloading a database. With this setting on, R:BASE displays the process being performed, the overall progress, and the completion percentage of each.

6.15.6.74 QUALCOLS

Operating Condition

Syntax: SET QUALCOLS n

Default: 10

QUALCOLS specifies the number of qualkeys to assign to SQL attached tables.

When attaching external tables by selecting "Utilities" > "Attach SQL Database Tables" from the menu bar, or using the SATTACH command, the QUALOCOLS setting is used to assign what columns uniquely identifies a row.

If a primary key or unique key was not found for the table being SATTACHed, and the USING clause was not used to specify what columns uniquely identifies a row, then R:BASE assigns primary and unique key qualkeys for the attached table. R:BASE assigns a set of columns to identify the rows starting with the first column of the table. The number of columns used is limited by the value for QUALCOLS.

The following command line sets QUALCOLS to 5:

```
SET QUALCOLS 5
```

6.15.6.75 QUOTES

Special Character

Sets the character for quotation marks.This character is used around all text strings.

Syntax: SET QUOTES=NULL
SET QUOTES=char
(Use NULL to disable the special character.)

Default: '

6.15.6.76 RBADMIN

Operating Condition

Syntax: SET RBADMIN ON/OFF

Default: OFF

The RBADMIN setting is used for RBAdmin, the R:BASE Network Database Administrator utility.

In order for RBAdmin to disconnect users from the database, this setting must be set to ON. All users, whether their setting for RBADMIN is ON or OFF will be seen within RBAdmin. It is recommended that this setting be added to a database application startup file for ease of implementation with RBAdmin. When RBADMIN is set ON and connections are made to the database, a hidden binary file will be created in the database directory. The name of the file is unique to each database; consisting of "RBAdmin_" + dbname + ".bin". The binary file for the ConComp sample database, with RBADMIN set ON, would be "RBAdmin_ConComp.bin".

6.15.6.77 RECYCLE

Operating Condition

Syntax: SET RECYCLE ON/OFF

Default: OFF

If RECYCLE is ON, when adding new rows require a new block from file 2, a new routine is called which searches for a suitable unused block rather than always adding a new block to the end of the file

The criteria for such a block are:

- No other table uses it
- The block is further down the file than the current last block of the table

PROS
If a suitable block is found, the block will be allocated to the table requiring the additional space and File 2 will not grow as a result of this allocation. The main benefit of using RECYCLE is that the growth of File 2 will be minimized. Over time this can add up to significant savings on disk space and backup media.

CONS
Since a new routine is being called to search for a suitable block, there will be a slight performance penalty. The penalty will only be incurred when an INSERT requires a new block.

Considerations
For RECYCLE to be effective, all users should have the setting ON. Do this in the configuration file. RECYCLE will only have an impact when used in conjunction with PACK TABLE. Dead space in File 2 must first be freed up before it can be reused. RECYCLE will not be of benefit if your database does not end up with lots of deleted rows over time, providing the opportunity to recover dead space.

Conclusions
Periodic use of PACK TABLE tablename in conjunction with RECYCLE ON will retard File 2 growth and reduce fragmentation. Use of PACK INDEX FOR tablename will keep the index statistics fresh and query optimization results maximized. The need for planned downtime will be reduced.

6.15.6.78 REFRESH

Operating Condition

Syntax: SET REFRESH value

Range: 10 to 65535 seconds

Default: 0

Mode: Multi-user

SET REFRESH specifies how often R:BASE refreshes the form or the Data Browser, and displays current data. It also automatically recalculates lower tables in forms. Specify zero to turn the setting off. When REFRESH is set off, R:BASE tells you of edits when you save or delete a row.

6.15.6.79 REVERSE

Operating Condition

Syntax: SET REVERSE ON/OFF

Default: ON

SET REVERSE ON displays data-entry fields in reverse video in forms. R:BASE stores the setting with the database.

6.15.6.80 ROWLOCKS

Operating Condition

Syntax: SET ROWLOCKS ON/OFF

Mode: Multi-user

Default: ON

R:BASE uses row-level locking in a multi-user environment. This command causes R:BASE to lock only the required row for the current command instead of locking the entire table. For example, if multiple users are modifying the same table using the UPDATE command, R:BASE locks only the rows affected by each UPDATE. When ROWLOCKS is set off, R:BASE sets table locks during each UPDATE, regardless of how many rows are affected.

6.15.6.81 RULES

Operating Condition

Syntax: SET RULES ON/OFF

Default: ON

SET RULES determines whether R:BASE checks data against all existing rules during data entry and modification when you use the EDIT, EDIT USING, ENTER, INSERT, LOAD, or UPDATE commands, or the import/export utility.

Set RULES off to direct R:BASE to ignore all rules when rules are not defined for a table, you are archiving data, or you are transferring data into another database. This speeds up processing because R:BASE normally checks the SYS_RULES table even if no rules are defined for a table. R:BASE does, however, check each entry against the data type of the column regardless of the RULES setting. If the database is protected by a database owner's user identifier, R:BASE does not accept the SET RULES command until you enter the owner's user identifier.

6.15.6.82 SCRATCH

Operating Condition

Syntax: SET SCRATCH ON / OFF / TMP / <path>

Default: TMP

SET SCRATCH sets the drive and directory location for temporary files created when sorting data.

- SET SCRATCH ON to store temporary sort files on the database drive and directory.
- SET SCRATCH OFF to store temporary files on the current drive and directory.
- SET SCRATCH TMP to store temporary files in the Windows TEMP directory.
- SET SCRATCH <path> provides the path to the location where temporary files are stored.

You can use the SCRATCH command in the configuration file so that the setting is made prior to launching R:BASE.

About SCRATCH TMP

By default, R:BASE configuration files include the TMP for SCRATCH setting.

This default will allow R:BASE or Oterro sessions to use the user's TMP environment settings for SCRATCH files on startup and eliminate all issues related to setting the SCRATCH directory and related files.

To take advantage of this setting, use the option "SCRATCH TMP" in the appropriate R:BASE and OTERRO configuration files or use the "SET SCRATCH TMP" command in your application startup files.

6.15.6.83 SELMARGIN

Operating Condition

Syntax: SET SELMARGIN value

Range: 0 to the width of your screen

Default: 0

Use SELMARGIN to set the left margin for displaying the results of a SELECT command. The default for SELMARGIN is 0 (zero), which sets the margin to column 2. Use SELMARGIN when an ASCII file requires a predefined position or when data has a required starting point.

Note: Setting SELMARGIN to 0 (zero) or 2 begins the left margin at column two.

6.15.6.84 SEMI

Operating Condition

Syntax: SET SEMI ON/OFF

Default: OFF

Use SEMI to set the semicolon (;) key as the command terminator instead of the [Enter] key.

When SEMI is set on, you can enter multiple command lines without a continuation symbol. Also, the semicolon runs commands created for other SQL databases, such as SQL Server.

Note: When SEMI is set on, all commands, including EXIT, must be followed with a semicolon.

The following command line sets SEMI off:

```
SET SEMI OFF;
```

6.15.6.85 SEMI (Special Character)

Special Character

Sets the character for the command separator.

Syntax: SET SEMI=NULL
SET SEMI=char
(Use NULL to disable the special character.)

Default: ;

6.15.6.86 SERVER

Operating Condition

Syntax: SET SERVER ON/OFF

Default: ON

SET SERVER sets messages from a foreign data source on or off. When SERVER is set on, messages from the foreign data source are displayed.

6.15.6.87 SHORTNAME

Operating Condition

Syntax: SET SHORTNAME ON/OFF

Default: OFF

Alters the display format of the directory contents, where the file names are listed.

With SHORTNAME set to ON, the DIR command lists the file name, extension, size in bytes, and the date and time files were last modified, only listing the contents in the traditional format.

Example 1:

The following command could be used to review the database files placed in a temp folder. The display uses SHORTNAME set to OFF.

```
R>DIR *.RX?

 Volume in drive C is Acer
 Volume Serial Number is 4060-5572

 Directory of C:\Temp\

17.11.11    04:56 PM             48,458 RBTIDATA.RX1
17.11.11    04:56 PM          3,112,960 RBTIDATA.RX2
17.11.11    04:56 PM            770,048 RBTIDATA.RX3
17.11.11    04:56 PM             12,288 RBTIDATA.RX4
25.08.11    01:38 PM             73,146 RRBYW17.RX1
25.08.11    01:38 PM          1,310,720 RRBYW17.RX2
25.08.11    01:38 PM            180,224 RRBYW17.RX3
25.08.11    01:38 PM          3,964,928 RRBYW17.RX4
           8 File(s)          9,472,772 bytes
           0 Dir(s)      41,285,623,808 bytes free
```

Example 2:

The following command could be used to review the database files placed in a temp folder. The display uses SHORTNAME set to ON.

```
R>DIR *.RX?

 Volume in drive C is Acer
 Directory of C:\Temp\

RBTIDATA RX1      48458 17.11.11   04:56p RBTIDATA.RX1
RBTIDATA RX2    3112960 17.11.11   04:56p RBTIDATA.RX2
RBTIDATA RX3     770048 17.11.11   04:56p RBTIDATA.RX3
RBTIDATA RX4      12288 17.11.11   04:56p RBTIDATA.RX4
RRBYW17  RX1      73146 25.08.11   01:38p RRBYW17.RX1
RRBYW17  RX2    1310720 25.08.11   01:38p RRBYW17.RX2
RRBYW17  RX3     180224 25.08.11   01:38p RRBYW17.RX3
RRBYW17  RX4    3964928 25.08.11   01:38p RRBYW17.RX4
           8 File(s)        9472772 bytes
           8 Dir(s)     41285361664 bytes free
```

6.15.6.88 SINGLE

Special Character

Sets the character for the single wildcard for R:BASE commands and clauses.

Syntax: SET SINGLE=NULL
        SET SINGLE=char
        (Use NULL to disable the special character.)

Default: _

6.15.6.89 SORT

Operating Condition

Syntax: SET SORT ON/OFF

Default: OFF

SET SORT sets the sort optimizer on or off. When set on, R:BASE sorts the minimal amount of data for large tables and recombines the sorted data with the unsorted rows using the minimum amount of disk space. Set SORT on when a sort fails.

SORT is only to be used when displaying a column, or columns, which are not indexed. Otherwise, no results will be displayed.

6.15.6.90 SORTMENU

Operating Condition

Syntax: SET SORTMENU ON/OFF

Default: ON

SORTMENU causes all data dictionary menus to be in alphabetical ascending order, including all pop-up menus that display tables, forms, views, labels, and reports. Menus with column names and values remain unsorted in their original order.

6.15.6.91 STATICDB

Operating Condition

Syntax: SET STATICDB ON/OFF

Default: OFF

SET STATICDB activates a read-only schema mode. A user who first connects to a database with STATICDB set to on engages that database to operate in a read-only schema mode whereby any user must have their STATICDB setting on in order to connect to that database.

The effect of having STATICDB set on is that no schema changes can occur during that connection session.

Schema Reading Mode with SET STATICDB
In a multi-user environment, R:BASE must be aware of any schema modifications made by a user. Being aware of schema modifications, which is necessary for database integrity, entails constantly re-reading schema information.

Because many data-processing operations do not make frequent schema changes to the database, such a re-reading unnecessarily slows down processing.

The command SET STATICDB ON instructs R:BASE to prevent any schema modifications, therefore, R:BASE does not read schema information. By default, STATICDB is set off. SET STATICDB OFF forces a re-read of schema information before running any command.

When connecting to a database with STATICDB set on, R:BASE displays the "Database Schema is Read-Only." message. When STATICDB is set on, any changes made to add new tables or views results in a temporary table/view. The table/view will be dropped when disconnecting from the database.

Also, the following R:BASE commands are not allowed when STATICDB is set on.

ALTER TABLE*	DROP*
COMMENT ON*	CREATE INDEX

* Access is not allowed to tables created prior to database connection.

No ALTER, DROP, CREATE, SCONNECT, SATTACH, SDETACH, ATTACH, and DETACH commands will have any effect. You can issue CREATE, ATTACH, and SATTACH commands but they will only create temporary tables or views. Similarly, PROJECT will create temporary tables with or without the TEMP keyword. Because the database schema is protected from changes, R:BASE doesn't need to worry that a user will change the schema in the middle of a series of commands.

The UNLOAD ALL command does not act upon temporary tables. You can, however, unload individual temporary tables with the UNLOAD command.

If a user has STATICDB set off, that user cannot connect to a database being used by a user who has STATICDB set on. All users accessing the same database must have a matching STATICDB setting.

You can find out what the current STATICDB setting is with the SHOW STATICDB command. You can also capture the current STATICDB setting as a value with the CVAL function:

```
SET VAR vcval = (CVAL('STATICDB'))
```

6.15.6.92 TIME

Operating Condition

Syntax: **SET TIME SEQUENCE HHMMSS** (time sequence)
SET TIME FORMAT HH:MM:SS (time format)

Default: **SET TIME SEQUENCE HHMMSS** (time sequence)
SET TIME FORMAT HH:MM:SS (time format)

SET TIME sets the time entry sequence and output format, using up to 20 characters. Set TIME entry sequence and display format in separate commands. Use H to specify hours, M for minutes, S for seconds, and .SSS for thousandths. TIME can be specified of up to thousandths of a second. R:BASE stores the setting with the database.

The keyword SEQ (sequence) sets the entry sequence such as HHMMSS while the keyword FOR (format) sets the display format. For example, the format HH:MM:SS can display 14:30:20. R:BASE displays midnight (24:00) as 0:00. You can also include AP to display time on a 12-hour clock. In the previous example, if you change the format to HH:MM:SS AP, R:BASE displays 2:30:20 PM. If the format contains spaces or commas, enclose the format in quotes. Enter the hours, minutes, and seconds in the order SEQ specifies.

Example: Valid Time Formats using two thirty and twenty seconds, p.m.

Time Format	Displays
HH:MM:SS	14:30:20
HH:MM:SS AP	2:30:20 PM
HH/MM/SS	14/30/20
HH-MM-SS AP	2-30-20 PM

TIME can affect time functions. For best results, first set TIME to the default HH:MM:SS and then use the time functions.

6.15.6.93 TIMEOUT

Operating Condition

Syntax: SET TIMEOUT value

Range: 0 to 1440

Default: 0

Use TIMEOUT to shut down an inactive R:BASE workstation and exit to Windows after a set amount of time passes. A countdown only begins when R:BASE is waiting for a keystroke, not while R:BASE is processing commands or while you are entering data. This is a useful feature for automatically disconnecting idle R:BASE sessions for scheduled database maintenance.

The default for TIMEOUT is 0 (zero), which does not activate a countdown. TIMEOUT is set in minutes (not seconds), and all workstations must set TIMEOUT separately.

When a TIMEOUT occurs, a command file can be run; however, the command file cannot expect a keystroke. If you want to run a command file when a TIMEOUT occurs, you need to store the name of the file in a variable called **RBTI_TIMEOUT**.

For example:

```
SET VARIABLE RBTI_TIMEOUT TEXT = 'c:\CustDB\cleanup.rmd'
```

The following command line will exit a user to Windows after the user's workstation is inactive for one hour:

```
SET TIMEOUT 60
```

Notes:

- The RBTI_TIMEOUT command file must end with a RETURN command.
- The TIMEOUT command will close ANY and ALL open forms, designers, and editors WITHOUT saving the changes which have been made since the last save. It is the responsibility of the developer and end-user to implement proper coding and/or behavior to eliminate unexpected shutdowns without saving the changes. The TIMEOUT command will disconnect from the currently opened database (if applicable) before terminating the R:BASE session.

6.15.6.94 TOLERANCE

Operating Condition

Syntax: SET TOLERANCE value

Default: 0

SET TOLERANCE sets the tolerance for comparisons between numbers with REAL and DOUBLE data types. The default tolerance of 0 means that the match must be exact to six digits of accuracy for REAL numbers and to 15 digits of accuracy for DOUBLE numbers. R:BASE stores the setting with the database.

The following command specifies a tolerance of .1 when testing column values. If the tolerance is .1 and the WHERE clause specifies *colname* = 100, then values between 99.9 and 100.1 are valid. If you set the tolerance to one, the values between 99 and 101 are valid.

```
SET TOLERANCE .1
```

6.15.6.95 TRACE

Operating Condition

Syntax: SET TRACE ON/OFF

Default: OFF

SET TRACE ON will execute TRACE (Interactive Command File Debugger) inside a command file to trace a block of code as defined.

Example 01:

TRACE filename.ext (typical command line option)

Example 02: (in a command file)

your code here ....
SET TRACE ON (this will start the trace within a command file)
Your code here ...
SET TRACE OFF (this will stop the trace within a command file)
Your remaining code here ...

Notes:

- Once you turn OFF the TRACE in an R:BASE session, you will need to turn it back ON.
- (CVAL('TRACE')) will return the current status of TRACE (Values: ON or OFF)
- SHOW TRACE will display the current status of TRACE (Values: ON or OFF)
- Newly created configuration file will also include the option for **TRACE ON**
- If you want no one to TRACE your code, setting the TRACE option to OFF at the beginning of your code or startup file will disable the TRACE command.

6.15.6.96 TRANSACT

Operating Condition

Syntax: SET TRANSACT ON/OFF
SET TRANS ON/OFF

Default: OFF

Mode: Transaction Processing

SET TRANSACT toggles transaction processing on and off. When transaction processing is set on and AUTOCOMMIT is set off, all commands entered after one COMMIT or ROLLBACK command until the next comprise a transaction. The commands in a transaction are executed as they are entered, but changes to the data and database structure are not made permanent until you enter COMMIT (or exit the database). You can undo all changes in the transaction by entering ROLLBACK.

When transaction processing is on and AUTOCOMMIT is also on, each command that is executed successfully is treated as a transaction and made permanent. ROLLBACK has no effect when AUTOCOMMIT is on.

Only the first user to connect to a closed database can enter the TRANSACT setting for that database. Enter the command before connecting to the database. If anyone else already has the database open, R:BASE displays a message telling you that your TRANSACT setting must match that of the open database before you can connect. Transaction processing is either on for all users or off for all users in a given database.

6.15.6.97 UINOTIF

Operating Condition

Syntax: SET UINOTIF ON/OFF

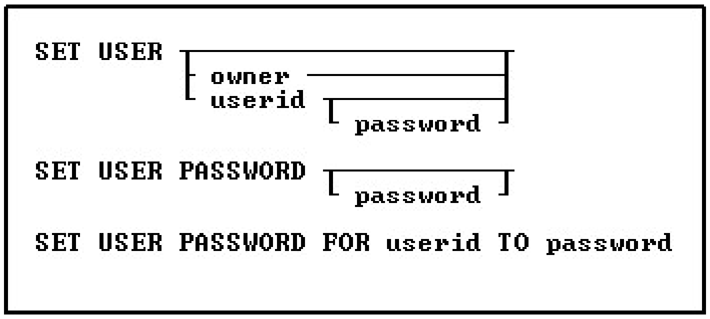
Default: ON

Controls the automatic user interface updates at the Database Explorer to refresh the list of tables, views, stored procedures, forms, reports, and labels.

Setting UINOTIF to OFF within an application will improve the performance.

6.15.6.98 USER

Use the SET USER command to create users, change the password for a user, or run R:BASE with a user identifier and password, if one has been set up.



Options

FOR userid
Specifies a user identifier. For a value with spaces, the userid must be enclosed in quotes.

owner
Specifies the database owner name.

password
Creates a new password. Enter NONE to remove an existing password.

PASSWORD
Specifies or changes the password for the current user identifier.

TO password
Creates a new password. Enter NONE to remove an existing password.

userid
Specifies a user identifier. For a value with spaces, the userid must be enclosed in quotes.

About the SET USER Command

Passwords are specific to user identifiers and databases and are not required by R:BASE; however, once a password is set up, R:BASE prompts for the user's password every time the user connects to the database or issues a user identifier. User identifiers have a maximum length of 36 characters. Passwords have a minimum length of three characters and maximum length of 36 characters.

To run R:BASE with your user identifier then connect to the database, enter the following command line:

```
SET USER <Userid>
```

You can also enter the following command line to run R:BASE with your user identifier:

```
SET USER
```

R:BASE displays a dialog box and prompts you for your user identifier.

When a password has been set up for a user identifier, R:BASE prompts for the password after the correct user identifier has been entered.

Note: When a user enters a user identifier or password in a dialog box, the user identifier is not displayed on screen.
To add or change a password, connect to the database with the user identifier and enter the following command line:

```
SET USER PASSWORD
```

R:BASE prompts for the user's identifier, then prompts for the password. A user can cancel a password by entering NONE.

If the database owner is the current user, the database owner can assign him/herself a password using the SET USER PASSWORD command; however, if the database owner forgets the assigned password, the password cannot be found or changed.

As the database owner, to change a user's password, connect to the database and enter the following command line:

```
SET USER PASSWORD FOR <Userid> TO <Password>
```

Enter your current password when R:BASE prompts you for it, then when R:BASE prompts you for a new password, enter NONE.

Note: A user's password is revoked when the database owner revokes all the user's privileges.

6.15.6.99 UTF8

Operating Condition

Syntax: SET UTF8 ON/OFF

Default: OFF

Controls the ability to use Unicode characters for string functions in applications and environments which will use higher character sets.

6.15.6.100 VERIFY

Operating Condition

Syntax: SET VERIFY COLUMN/ROW

Default: COLUMN

Mode: Multi-user

SET VERIFY, used in the multi-user environment, specifies the level of concurrency control as a row or a column within a row.

SET VERIFY allows you to specify whether R:BASE concurrency control will apply to individual columns within a row or to all columns in the row. When the level of concurrency control is set to COLUMN, R:BASE checks only the columns you change while you are editing. When the level of concurrency control is set to ROW, if you change data in any column, R:BASE checks every column in the row.

R:BASE concurrency control operates automatically when you are using a form in multi-user environments. Concurrency control includes autorefresh and verification. When you refresh or try to save a row, R:BASE checks whether data has been changed by another user and alerts you if it has changed. This prevents simultaneous changes that could corrupt the integrity of your data. The SET VERIFY command affects the operation of both autorefresh and verification when you are using a form.

When concurrency control is set to COLUMN, R:BASE looks for conflicts, those instances when two users have both modified the same column. When R:BASE detects a conflict in at least one field:

- R:BASE displays all of the other user's changes in the appropriate fields.
- Where there is no conflict, changes you made continue to be displayed.
- R:BASE displays a message informing you that data has changed.

When concurrency control is set to ROW, R:BASE looks for a change to any column in the whole row, whether it is a conflict or not. When R:BASE detects a change:

- R:BASE displays all of the other user's changes in the appropriate fields.
- Where there is no conflict, changes you made to the data are discarded.
- R:BASE displays a message informing you that data has changed.

Whether concurrency control is set to COLUMN or ROW, you can review the changes and then continue editing the data in the form. After autorefresh, R:BASE prompts you to press any key to continue editing. After verification, you can either move on to your next task or edit the data again. If you choose to move on when the level of concurrency control is set to COLUMN, you will be discarding any changes you made that are still displayed. R:BASE prompts you to press [Esc] if you want to move on, or to press [Enter] if you want to edit the displayed data.

When you edit data in a form, concurrency control is always enforced.

When you enter data in a form, concurrency control is enforced only when you are entering values in fields defined with a same-table look-up, or when you return to a row in a region that you had entered previously.

The first command line in the following example sets the level of concurrency control to check for changes in the entire row. The second command line starts an editing session using the form named *custform*.

```
SET VERIFY ROW
EDIT USING custform
```

6.15.6.101 WAIT

Operating Condition

Syntax: SET WAIT value

Range: 0 to 16383 seconds

Default: 4

Mode: Multi-user

SET WAIT sets the minimum number of seconds to retry the last requested resource (a table or database) before halting execution. Rather than aborting execution, SET WAIT allows you to set a length of time for R:BASE to keep trying to access a resource. A message is displayed showing the approximate percentage of wait time remaining.

If you do not run a SET WAIT command, R:BASE automatically retries the locked resource for approximately four seconds before issuing a retry message.

For commands that wait for resources, the precise period of the wait is at least as long as the time specified. On some computers, processing requirements may extend the length of the wait to longer than one second for each second designated.

When you enter a command from the R> Prompt and the waiting period expires, R:BASE displays a message prompting you to retry or ignore the command. When the command runs as part of a command file, however, and the waiting period expires, R:BASE ignores the command and goes on to the next command.

The following command tells R:BASE to retry the last requested resource for approximately 20 seconds.

```
SET WAIT 20
```

You can also adjust the interval in which R:BASE tries the command during the SET WAIT period.

For more information, see INTERVAL.

6.15.6.102 WALKMENU

Operating Condition

Syntax: SET WALKMENU ON/OFF

Default: OFF

Allows shortcut access to menus

SET WALKMENU is a menu shortcut function allowing the user to access menu selections by typing the beginning characters (up to when a match is made) of their names. Pressing any navigational keys (such as [Home] or [Page Up]) clears the buffer containing the keystrokes entered by the user while traversing the menu list. Any keystrokes not resulting in a match are not stored in the buffer, causing a beep.

6.15.6.103 WHILEOPT

Operating Condition

Syntax: SET WHILEOPT ON/OFF

Default: ON

SET WHILEOPT improves the optimization and processing of WHILE ...ENDWHILE loops within applications by pre-compiling variables used within the WHILE loop. Follow these guidelines:

- Don't clear your WHILE variable(s).
- Don't define variables within your WHILE loop, only outside the loop; values can change within the loop.
- Adhere to the syntax rules for the SWITCH statement by making sure that the argument for the SWITCH statement is an expression.
- If you issue multiple SET VARIABLE commands on a single command line, those variables will not be optimized. If you want to increase the speed for that loop, put those SET VARIABLE commands on separate lines.

The WHILEOPT setting must be changed in a command file. The setting cannot be saved to the configuration file.

6.15.6.104 WIDTH

Operating Condition

Syntax: SET WIDTH value

Range: 40 to 5000 characters

Default: 79

SET WIDTH controls the width of a data line that R:BASE directs to the printer, screen, or file when using the BACKUP, COMPUTE, CROSSTAB, DISPLAY, SELECT, TYPE, UNLOAD, or WRITE commands. Do not set the width to a number greater than the number of characters your printer can fit on a line; a typical page and computer screen display 80 characters. WIDTH does not affect report generation; each report defines the width of a data line.

Note: The 5000-character maximum does not apply to the TYPE or DISPLAY commands, which continue to have 256-character width limits.

6.15.6.105 WINAUTH

Operating Condition

Syntax: SET WINAUTH ON/OFF

Default: OFF

SET WINAUTH controls if R:BASE will automatically authenticate database connections through Integrated Windows Authentication (IWA) with Windows Active Directory. When WINAUTH is set ON, the security features of the Windows clients and servers allows access to an R:BASE database when the Windows user name and password matches a defined R:BASE database user name and password. The current

Windows user information on the client computer is supplied to R:BASE and does not prompt user for additional user name and password.

In order to use this feature, R:BASE database administrators must create an owner and user privileges for the database files to take full advantage of IWA support.

In addition to creating the database user privileges, the operating condition WINAUTH must be set to ON when prior to users connecting to the database. WINAUTH can be set within the R:BASE configuration file, or defined within an application startup file.

Requirements:

- WINAUTH must be set ON before connecting to the database.
- Database must contain an owner and user privileges that match Windows network accounts.

Notes:

- R:BASE will accept user names that contain spaces.
- Passwords are case-sensitive.
- If the authentication exchange initially fails to identify the user, R:BASE will prompt the user for a user name and password.
- A new CVAL Function (CVAL('WINAUTH')) has been established to check if WINAUTH is on.
- When an R:BASE user successfully connects to the database via IWA, the current R:BASE USER identifier is set to the current Windows user name.

6.15.6.100 WINBEEP

Operating Condition

Syntax: SET WINBEEP ON/OFF

Default: OFF

Allows R:BASE to access a subset of the Sound Events in Windows. WINBEEP command will use the current system sound schema.

When set to OFF R:BASE will use a standard Windows sound for all errors and the BEEP command. When set to ON certain sounds, as set in the Windows Control Panel, will be used instead depending on which sound type is used. Below is a table of Types and their corresponding sound event. You must set the actual sounds used in the Windows Sounds Control Panel Applet. You will also need to ensure that your speakers are Un-Muted and working properly. In either case, WINBEEP ON or WINBEEP OFF, if you have the "No Sounds" scheme selected in your Sounds Control Panel you will not hear any beeps from R:BASE for Windows.

Sound Type	**Sound Event**
0	SYSTEM
1	SYSTEMSTART
2	SYSTEMEXIT
3	SYSTEMHAND
4	SYSTEMASTERISK
5	SYSTEMQUESTION
6	SYSTEMEXCLAMATION
7	SYSTEMWELCOME
8	SYSTEMDEFAULT

Example

```
SET CAPTION ' '
SET VAR VRows INTEGER = 0
SET VAR VMsg TEXT = NULL
SELECT COUNT(*) INTO VRows INDIC IVRows FROM TableName
IF VRows = 0 THEN
   CLS
   SET WINBEEP ON 2
   BEEP
   PAUSE 2 USING 'No Record(s) on File!' AT CENTER CENTER
   SET WINBEEP OFF
   GOTO Done
ELSE
   CLS
   SET WINBEEP ON 1
   BEEP
   SET VAR VMsg = ((CTXT(.VRows)) & 'Record(s) on File!')
   PAUSE 2 USING .VMsg AT CENTER CENTER
   SET WINBEEP OFF
   GOTO Done
ENDIF

LABEL Done
   CLS
   CLEAR ALL VAR
```

6.15.6.107 WRAP

Operating Condition

Syntax: SET WRAP ON/OFF

Default: ON

Text in fields that have NOTE and TEXT data types will wrap in forms and FILLIN windows, variables, and reports.

6.15.6.108 WRITECHK

Operating Condition

Syntax: SET WRITECHK ON/OFF

Default: OFF

SET WRITECHK ON tells R:BASE to verify every write to disk.

6.15.6.109 ZERO

Operating Condition

Syntax: SET ZERO ON/OFF

Default: OFF

SET ZERO allows a null to be treated as a zero in a mathematical expression involving INTEGER, NUMERIC, REAL, DOUBLE, CURRENCY, DATE, DATETIME, or TIME data types. With ZERO set on, R:BASE returns a negative number when you subtract an integer from a null. With ZERO set off, the same computation results in a null. R:BASE stores the setting with the database.

6.15.6.110 ZOOMEDIT

Operating Condition (**R:BASE for DOS only**)

Syntax: SET ZOOMEDIT ON/OFF

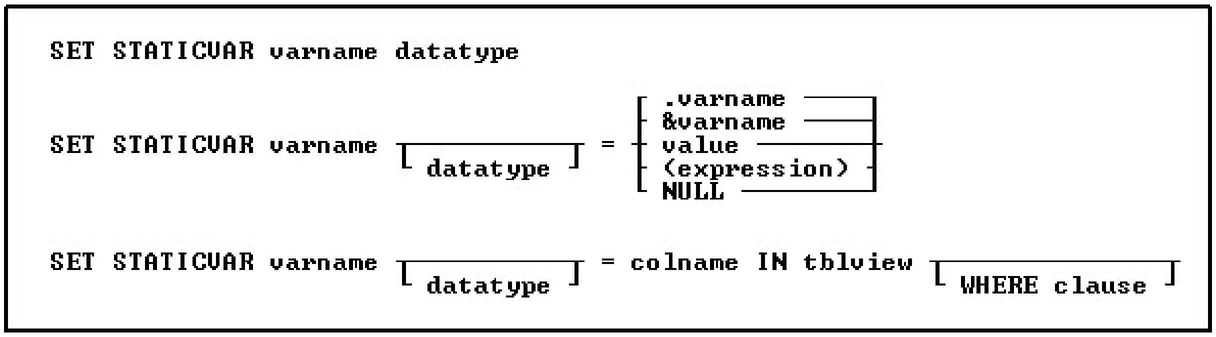
Default: OFF

Toggles the field expansion method.

Set ZOOMEDIT to on if you want to open RBEdit, the R:BASE text editor, when you zoom in on a NOTE filed in a form. When ZOOMEDIT is off, a dialog box that wraps text is opened instead.

6.15.7 SET STATICVAR

Use the SET STATICVAR command to define or redefine a static variable value and/or data type.



Options

colname
Specifies a column name. The column name is limited to 128 characters. In a command, you can enter *#c*, where *#c* is the column number in the table's column order. In an SQL command, a column name can be preceded by a table or correlation name and a period (*tblname.colname*).

datatype
Specifies an R:BASE data type for the static variable. See Data Types.

(expression)
Determines a value using a text or arithmetic formula. The expression can include constant values, functions, or system variables such as *#date*, *#time*, and *#pi*.

IN tblview
Specifies a table or view.

NULL
Sets the static variable equal to NULL.

value
Sets the static variable equal to a specified value. A value is a constant amount, text string, date, or time, or the value assigned to *varname*.

varname
Specifies a static variable name, which must be unique among both the static variable names within the database and the global variable names for the R:BASE session. The static variable name is limited to 128 characters.

&varname
Sets the first variable equal to the exact contents of a second variable; the ampersand tells R:BASE to evaluate the contents of the variable first.

For example, if *varname* is the text string (2+3), then &*varname* is the value 5.

.varname
Sets the first variable equal to the exact contents of a second variable.

For example, if *varname* is the text string (2+3), then .*varname* is (2+3).

WHERE clause
Limits rows of data. For more information, see the WHERE Clause.

About the SET STATICVAR Command

The SET STATICVAR command defines static variables, which are part of a connected database. Once defined, the variables are created every time the database is connected. When a database is disconnected, the static variables associated with the database are cleared. Static variable can be used just like normal variables with SET VARIABLE commands and in expressions. The basic syntax is just like the SET VARIABLE command except you use the word STATICVAR instead of VARIABLE (or VAR).

The UNLOAD command will create SET STATICVAR commands for the static variables defined for the connected database.

To clear one or more static variables use the CLEAR command. CLEAR ALL VAR will clear all variables (global and static), but immediately recreates the static variables, just like R:BASE does with system variables.

Important: It is recommended to define each static variable as a separate entry in the database. Defining multiple static variables with a single SET STATICVAR command should be avoided, as the multiple variables are defined as a group and are retained as a single database entry. If the variable group are always used as a set, then there will not be a problem. If the intent is to store each static variable as a separate item, each variable must be defined with a separate SET STATICVAR command.

The benefits of static variables include:

- Database unloads will no longer trip over missing global variables
- Users would not have to remember to define required variables used in views
- Distributed applications could contain static variables specific to each customer

Static variable have the following restrictions:

- The variable name is not an R:BASE reserved word.
- The variable name begins with a letter, contains only letters, numbers, and the following special characters: #, $, _ , and %.
- The variable definitions are limited to 512 characters.

It is good programming practice to always define the data type for the variable before assigning it a value. When defining an variable as a text string, enclose the text string in single quote marks (or the current QUOTES character); otherwise, it might be interpreted as an arithmetic expression.

Assigning a Data Type to a Static Variable

The *datatype* option refers to one of the valid R:BASE data types. You can define a static variable to have a NOTE data type, but R:BASE treats it as TEXT for most uses. You can also specify the precision and scale for NUMERIC data types.

The *datatype* option creates a static variable, determines its data type, and sets its value to null. Use this option to define a static variable's data type before assigning a value to the variable.

For an existing static variable, you can use the *datatype* option to change the data type, but it is recommended to use one of the conversion functions. If you change the data type, the new data type must be compatible with the current variable value; if the variable is not compatible, R:BASE displays an error message and leaves the value and data type unchanged. If you change a variable with a TEXT data type to a non-compatible data type, R:BASE changes the value to null.

Assigning a Value to a Static Variable

The *value* option is a data value or constant, such as 10, TOM, 3.1416, or $17.23. If the static variable already exists, any new value must have a data type that is compatible with that variable. If the static variable does not exist, R:BASE defines the variable's data type. You can also define the variable's data type in this command before assigning it a value.

Setting the Value of a Static Variable to Another Variable

When you set a static variable to the value of another variable, the second variable must be a dot variable (.) or an ampersand (&) variable.

When you precede a variable with a dot (.), R:BASE uses the value stored in the variable as if it were a constant.

When you precede a variable with an ampersand (&), R:BASE first evaluates the value contained in the ampersand variable. For example, consider the following uses of the command:

```
SET STATICVAR vM TEXT = 'Multi'
SET STATICVAR vP TEXT = 'Purpose'
SET STATICVAR vMP TEXT = '(vM + vP)'
SET STATICVAR vMPValue = .vMP
SET STATICVAR vMPCompute = &vMP
```

When the third command line runs, the variable *vMP* will contain (vM + vP). When the forth command line runs, variable *vMPValue* will also contain (vM + vP) because the dot tells R:BASE to set the value as an exact match to the contents of variable *vMP*. When the fifth command line runs, variable *vMPCompute* will contain MultiPurpose (the concatenation of Multi and Purpose) because the ampersand tells R:BASE to compute the contents of variable *vMP*.

As shown in the example above, an ampersand variable can contain one command or part of one command, such as an expression. The first variable is set to the computed value of the ampersand variable. Below is an example:

```
1. SET STATICVAR v1 TEXT
2. SET STATICVAR v2 INTEGER
3. SET STATICVAR v1 = '((50 + 100)/ 2)'
4. SET STATICVAR v2 = &v1
```

1. Sets the data type for the variable *v1* to TEXT.
2. Sets the data type for the variable *v2* to INTEGER.
3. Sets variable *v1* to a text value that is a valid arithmetic expression.
4. Sets variable *v2* to &*v1*.

R:BASE computes the expression contained in *v1* and assigns the calculated value to *v2*. When R:BASE sees a variable name preceded by ampersand, it treats the contents of the variable as if it was entered from the keyboard.

Setting a Static Variable to an Expression

An (*expression*) can be either an arithmetic operation that combines two or more items in an arithmetic computation, or a string expression that concatenates two or more text items, or uses a TEXT function. The items can be values or the values contained in variables.

If you do not predefine the data type of a static variable, the original data type of each item determines the data type of the result. For example, if you add a variable that has an INTEGER data type to a variable that has a REAL data type, the resulting variable has a REAL data type unless you define the result to be an INTEGER data type.

If any item in an arithmetic expression is null, the result will be null unless you specify SET ZERO ON.

Assigning Column Values in a Table or View

If you specify a table or view in a SET STATICVAR command, you can include an optional WHERE clause to indicate which row to use. If you do not include the WHERE clause, R:BASE uses the value for the column in the first row.

You must have SELECT privileges on the table to use this form of SET STATICVAR.

Examples

Defines the vMessage static variable to have a TEXT data type.
```
SET STATICVAR vMessage TEXT
```

Defines the vReal static variable to have a REAL data type, and assigns it the value 100.9.
```
SET STATICVAR vReal REAL = 100.9
```

Defines the vNumer static variable to have a NUMERIC data type having a precision of 9 and scale of 3.
```
SET STATICVAR vNumer NUMERIC (9,3)
```

Assigns the integer value 14322 to the vNum static variable.
```
SET STATICVAR vNum = 14322
```

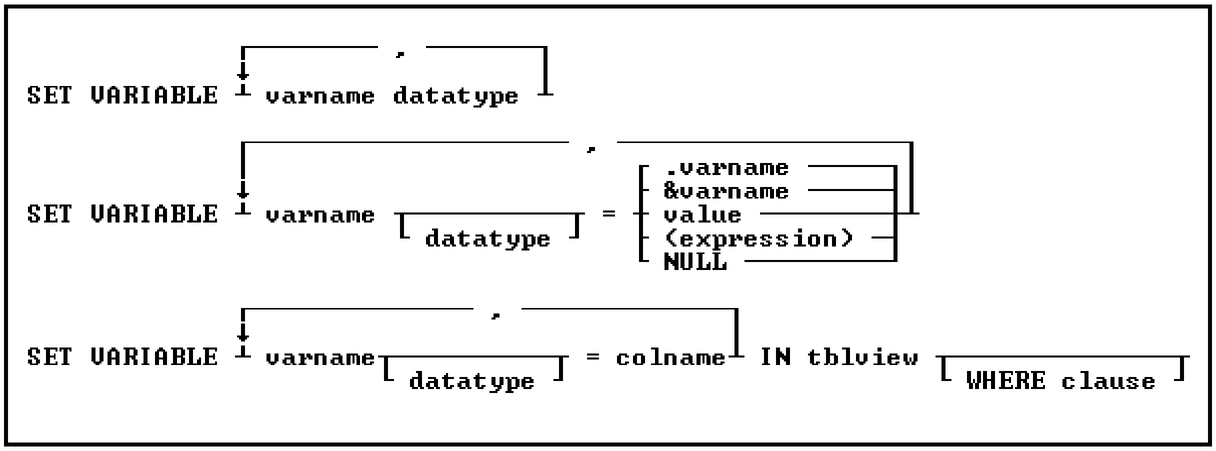
Assigns the value of the above vNum variable to the vTwo static variable.
```
SET STATICVAR vTwo =.vNum
```

Assigns the value 03/25/2022 to the vLtrDate static variable.
```
SET STATICVAR vLtrDate = ('12/25/2021' + 90)
```

Assigns the vCompanyName static variable from the row in the *LicenseInformation* table where the *LicenseNumber* value is equal to BM-68215.
```
SET STATICVAR vCompanyName TEXT = CompanyName IN LicenseInformation WHERE
LicenseNumber = 'BM-68215'
```

6.15.8 SET VARIABLE

Use the SET VARIABLE command to define or redefine a variable value and/or data type.



Options

,
Indicates that this part of the command is repeatable.

colname
Specifies a column name. The column name is limited to 128 characters.

In a command, you can enter *#c*, where *#c* is the column number shown when the columns are listed with the LIST TABLES command. In an SQL command, a column name can be preceded by a table or correlation name and a period (*tblname.colname*).

datatype
Specifies an R:BASE data type for the variable. See Data Types.

(expression)
Determines a value using a text or arithmetic formula. The expression can include other columns from the table, constant values, functions, or system variables such as *#date*, *#time*, and *#pi*.

IN tblview
Specifies a table or view.

NULL
Sets the variable equal to NULL.

value
Sets the variable equal to a specified value. A value is a constant amount, text string, date, or time, or the value assigned to *varname*.

varname
Specifies a variable name, which must be unique among the variable names within the database. The variable name is limited to 128 characters.

&varname
Sets the first variable equal to the exact contents of a second variable; the ampersand tells R:BASE to evaluate the contents of the variable first.

For example, if *varname* is the text string (2+3), then &*varname* is the value 5.

.varname
Sets the first variable equal to the exact contents of a second variable.

For example, if *varname* is the text string (2+3), then .*varname* is (2+3).

WHERE clause
Limits rows of data. For more information, see the WHERE Clause.

About the SET VARIABLE Command

Variables identify a changeable value. R:BASE provides three kinds of variables: global, error, and system. The SET VARIABLE command defines global variables, which are temporary variables that exist within R:BASE, but are not part of any database. Global variables remain in memory until you clear them or exit from R:BASE. R:BASE sets error and system variables internally.

Global variables have several uses: they can provide a temporary value in a command, hold the result of a calculation, act as a counter, or capture keyboard entries for use with menus or screens. The most common method of defining variables is to assign the variable value with the SET VARIABLE command.

Variable names have the following restrictions:

- The variable name is not an R:BASE reserved word.
- The variable name begins with a letter, contains only letters, numbers, and the following special characters: #, $, _ , and %.

It is good programming practice to always define the data type for the variable before assigning it a value, unless you are setting a variable to a column value or using the variable in the CHOOSE command.

When defining an variable as a text string, enclose the text string in single quote marks (or the current QUOTES character); otherwise, it might be interpreted as an arithmetic expression.

Assigning a Data Type to a Variable

The *datatype* option refers to one of the valid R:BASE data types. You can define a variable to have a NOTE data type, but R:BASE treats it as TEXT for most uses. You can also specify the precision and scale for NUMERIC data types.

The *datatype* option creates a variable, determines its data type, and sets its value to null. Use this option to define a variable's data type before assigning a value to the variable. To set multiple variables in the same command, separate the variables by a comma or the current delimiter.

For an existing variable, you can use the *datatype* option to change the data type, but it is recommended to use one of the conversion functions. If you change the data type, the new data type must be compatible with the current variable value; if the variable is not compatible, R:BASE displays an error message and leaves the value and data type unchanged. If you change a variable with a TEXT data type to a non-compatible data type, R:BASE changes the value to null.

Assigning a Value to a Variable

The *value* option is a data value or constant, such as 10, TOM, 3.1416, or $17.23. If the variable already exists, any new value must have a data type that is compatible with that variable. If the variable does not exist, R:BASE defines the variable's data type.

You can also define the variable's data type in this command before assigning it a value.

Setting the Value of a Variable to Another Variable

When you set the variable to the value of another variable, the second variable must be a dot variable (.) or an ampersand (&) variable.

When you precede a variable with a dot (.), R:BASE uses the value stored in the variable as if it were a constant.

When you precede a variable with an ampersand (&), R:BASE first evaluates the value contained in the ampersand variable. For example, consider the following uses of the command:

```
SET VARIABLE vM TEXT = 'Multi'
SET VARIABLE vP TEXT = 'Purpose'
SET VARIABLE vMP TEXT = '(vM + vP)'
SET VARIABLE vMPValue = .vMP
SET VARIABLE vMPCompute = &vMP
```

When the third command line runs, the variable *vMP* will contain (vM + vP). When the forth command line runs, variable *vMPValue* will also contain (vM + vP) because the dot tells R:BASE to set the value as an exact match to the contents of variable *vMP*. When the fifth command line runs, variable *vMPCompute* will contain MultiPurpose (the concatenation of Multi and Purpose) because the ampersand tells R:BASE to compute the contents of variable *vMP*.

As shown in the example above, an ampersand variable can contain one command or part of one command, such as an expression. The first variable is set to the computed value of the ampersand variable. Below is an example:

```
1.  SET VARIABLE v1 TEXT
2.  SET VARIABLE v2 INTEGER
3.  SET VARIABLE v1 = '((50 + 100)/ 2)'
4.  SET VARIABLE v2 = &v1
```

- Sets the data types for variables *v1* and *v2* to TEXT and INTEGER, respectively.
- Sets variable *v1* to a text value that is a valid arithmetic expression.
- Sets variable *v2* to &*v1*.

R:BASE computes the expression contained in *v1* and assigns the calculated value to *v2*. When R:BASE sees a variable name preceded by ampersand, it treats the contents of the variable as if it was entered from the keyboard.

Setting a Variable to an Expression

An (*expression*) can be either an arithmetic operation that combines two or more items in an arithmetic computation, or a string expression that concatenates two or more text items, or uses a TEXT function. The items can be values or the values contained in variables.

If you do not predefine the data type of a variable, the original data type of each item determines the data type of the result. For example, if you add a variable that has an INTEGER data type to a variable that has a REAL data type, the resulting variable has a REAL data type unless you define the result to be an INTEGER data type.

If any item in an arithmetic expression is null, the result will be null unless you specify SET ZERO ON.

Assigning Column Values in a Table or View

If you specify a table or view in a SET VARIABLE command, you can include an optional WHERE clause to indicate which row to use. If you do not include the WHERE clause, R:BASE uses the value for the column in the first row.

You must have SELECT privileges on the table to use this form of SET VARIABLE.

In instances where your building a dynamic SET VARIABLE command based on previous options made, you must use an ampersand variable in place of a column or table name, for example:

```
CHOOSE vtab FROM #TABLES
CHOOSE vcol FROM #COLUMNS IN &vtab
SET VARIABLE vnewpr = &vcol IN &vtab
```

Enter the table and column names into the *vtab* and *vcol* variables first. You can do this by using the CHOOSE #TABLES and CHOOSE #COLUMNS commands, as shown in the above example. The CHOOSE command displays a menu of tables or columns from which to choose. By using ampersand variables to hold the table and column names, you can use the same SET VARIABLE command to get values from different columns in a table or from different tables. Each time SET VARIABLE requests a column, it retrieves information from the first row in the table stored in *&vtab*.

NOTE: You can define multiple variables with a single SET VARIABLE command when you set the value of the variables to the value of columns in a table. However, when capturing column data into variables, it is better to use the SELECT command; specifically, SELECT INTO. SELECT INTO is the SQL compliant command when capturing table data into variables.

Examples

The following table provides examples of the SET VARIABLE command.

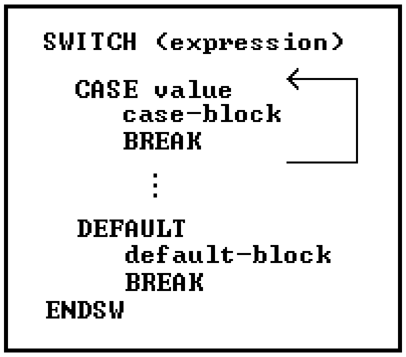
SET VARIABLE Examples

Examples	**Description**
SET VARIABLE *vtext* TEXT	Defines the *vtext* variable to have a TEXT data type.
SET VARIABLE *vreal* REAL = 100.9	Defines *vreal* variable to have a REAL data type, and assigns it the value 100.9.
SET VARIABLE *vnumer* NUMERIC (9,3)	Defines the *vnumer* variable to have a NUMERIC data type having a precision of 9 and scale of 3.
SET VARIABLE *vnum* = 14322	Assigns the integer value 14322 to the *vnum* variable.
SET VARIABLE *VTWO* =.VNUM	Assigns the value of the *vnum* variable to the *vtwo* variable.
SET VARIABLE *V3* = &*V4*	Assigns the computed value of *v4* to the *v3* variable.
SET VARIABLE *vltdate* = ('12/25/93' + 90)	Assigns the value 03/25/94 to the *vltdate* variable.
SET VARIABLE *vfulln* = (.*VFIRSTN* & .*VLASTN*)	Assigns to the *vfulln* variable the value of the full name formed by concatenating the values in the *vfirstn* and *vlastn* variables The ampersand inserts a space between the two values.
SET VARIABLE *v1* = *col1*, *v2* = *col2*, *v3*= *col3* IN *tbl1* WHERE *col1* = 'Smith' **OR SQL compliant variation:**	Assigns Smith to the variable *v1*; the value of column *col2* to *v2*; and the value of column *col3* in *tbl1*, from the row where *col1* contains Smith, to variable *v3*.

SELECT *col1, col2, col3* INTO *v1* INDI *iv1, v2* INDI *iv2, v3* INDI *iv3* FROM *tbl1* WHERE *col1* = 'Smith' *See* SELECT INTO	

6.15.9 SWITCH/ENDSW

Use the SWITCH...ENDSW command in a program to define a block of possible actions to take depending on the value of an expression. The SWITCH and CASE statements help control complex conditional and branching operations.


```
SWITCH (expression)
  CASE value
     case-block
     BREAK
       :
  DEFAULT
     default-block
     BREAK
ENDSW
```

Options

BREAK
Ends SWITCH processing; use this option within each CASE comparison and in the DEFAULT block.

case-block
Contains one or more commands to execute if the CASE value matches the SWITCH expression.

CASE value
Compares the SWITCH value to another value. If the values match, the commands following CASE are executed; otherwise, the next CASE comparison is checked.

DEFAULT
Provides commands to execute if no CASE comparisons are true.

default-block
Contains one or more commands to execute if no CASE comparisons are true.

(expression)
Determines a value using a text or arithmetic formula. The expression can include other columns from the table, constant values, functions, or system variables such as *#date*, *#time*, and *#pi*.

About the SWITCH...ENDSW Command Structure

The SWITCH statement is a control statement that handles multiple selections by passing control to one of the CASE statements within its body whose value matches the initial expression. The switch statement transfers control to a statement within its body. The syntax diagram shows the entire SWITCH...ENDSW, including the SWITCH value, CASE blocks, and the DEFAULT block.

```
SWITCH ([expression])
   CASE [value]
      [case-block]
      BREAK
DEFAULT
      [default-block]
      BREAK
ENDSW
```

The SWITCH statement allows a developer to control the order in which the code is executed because it is a conditional statement. In addition to handling multiple selections by passing control to one of the CASE statements within its body, the SWITCH provides an efficient mechanism for controlling, tracing, and debugging output at run time using external settings. Practically, you may define a common command file template/Custom Form Action and then use the R:BASE percent variables (%) to pass all required parameters for SWITCH and CASE values. A nested IF...ENDIF can be rewritten as one SWITCH statement.

The SWITCH Expression

SWITCH defines the expression to be compared. You can have multiple comparisons, so ENDSW defines the end of the comparisons. The SWITCH expression result must be either an INTEGER or a TEXT data type. The SWITCH expression can be a calculation, constant value, or variable. Any length of text can be compared, but only the first 30 characters are checked in each CASE block.

CASE Blocks

The SWITCH statement can include any number of CASE instances, but no two case statements can have the same value. A CASE block consists of three parts: the CASE comparison, the commands following each comparison, and the BREAK statement.

CASE comparisons must be the same data type as the SWITCH expression result - either INTEGER or TEXT. A CASE value cannot be an expression, but must be a constant value or a variable. You can have multiple CASE comparisons to run a single set of commands. For an example of how to use multiple comparisons, see "Examples" below.

The commands following a CASE comparison can include any R:BASE command, including a nested SWITCH…ENDSW structure. You can nest as many SWITCH…ENDSW structures as memory allows.

Use a BREAK statement as the last command in a CASE block to exit from the SWITCH…ENDSW structure. The BREAK command stops R:BASE from checking any additional CASE comparisons. Otherwise, R:BASE will execute every CASE statement even if only one condition is met.

The DEFAULT Block

Use the DEFAULT block to provide a set of commands to be executed if none of the CASE comparisons matches the expression. You can have only one DEFAULT block for each SWITCH…ENDSW structure. The DEFAULT block should be located in the last statement block in a SWITCH...ENDSW structure. If a CASE block follows a DEFAULT block, R:BASE generates a warning.

Example

The following SWITCH...ENDSW structure uses a date entered in a DIALOG command in the expression. The TDWK function calculates day of the week as text from the date stored in *vday*.

```
DIALOG 'Enter a date:' vday vendkey 1
SWITCH (TDWK(.vday))
   CASE 'Saturday'
   CASE 'Sunday'
      WRITE 'This is a weekend day.'
      SHOW VARIABLE vday
      BREAK
   DEFAULT
      WRITE 'This is a weekday.'
      SHOW VARIABLE vday
      BREAK
ENDSW
```

If you entered 12/17/94 when prompted for the date, the first CASE comparison would check whether the day of the week is the word *Saturday*. Because the word is *Saturday*, R:BASE would display the message below. The BREAK command prevents R:BASE from processing the rest of the commands in the SWITCH...ENDSW structure.

```
This is a weekend day.
12/17/94
```

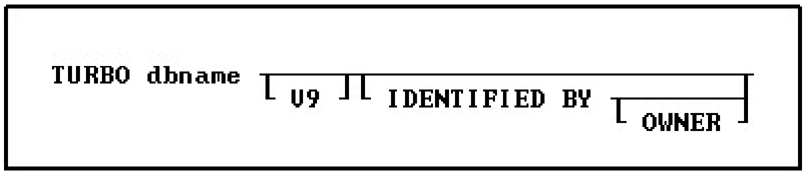
If the date entered is not *Saturday* or *Sunday* - for example, 12/22/94 - the information in the DEFAULT block would display the following.

```
This is a weekday.
12/22/94
```

6.16 T

6.16.1 TURBO

Use the TURBO command to convert your database files to R:BASE 11 database files.



Options

dbname
Specifies a database name

V9
Specifies that you are converting a R:BASE Turbo V-8 database to R:BASE 11.

IDENTIFIED BY
Specifies the user identifier. If left blank, R:BASE prompts you for the user identifier. R:BASE does not display it as you enter the text.

OWNER
Optional; specifies the database owner name. If omitted and an OWNER name exists, you will be prompted.

Notes:

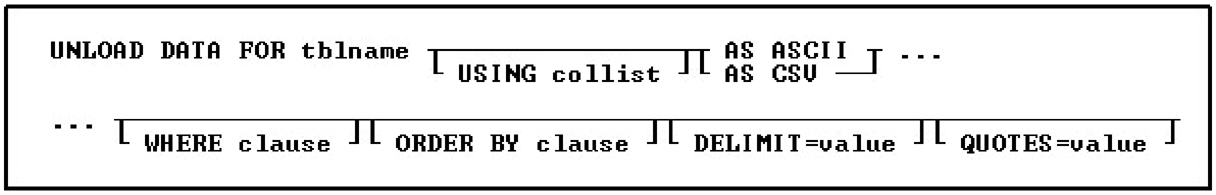
- If you are converting a RB1-RB4 database file structure to RX1-RX4 for the first time, you do not need to specify the "V9" parameter.
- The TURBO command will convert database files as far back as version 6.5.

For more on database conversions, please refer to the Database Conversion Guide PDF document located within the R:BASE program directory).

6.17 U

6.17.1 UNLOAD

Use the UNLOAD command to copy the data of a database table to a specified output device.



```
UNLOAD DATA FOR tblname [USING collist] [AS ASCII | AS CSV] ...

... [WHERE clause] [ORDER BY clause] [DELIMIT=value] [QUOTES=value]
```

Options

AS ASCII
Unloads data in ASCII-delimited format. Use only with the UNLOAD DATA command.

AS CSV
Unloads data in a minimally quoted comma separated format. Each field will be separated by the current DELIMIT character (usually the comma). Fields that contain the current DELIMIT character will be enclosed in the current QUOTES character.

DATA
Unloads the data.

DELIMIT=value
Specifies a custom delimiter value for the ASCII or CSV unload formats. The value can be a character or the CHAR function.

ORDER BY clause
Sorts rows of data. For more information, see ORDER BY. Unloads the table/view definition structure. The output contains only the SQL command necessary to create the table/view.

QUOTES=value
Specifies a custom quote character value for the ASCII or CSV unload formats. The value can be a character or the CHAR function.

tblname
Specifies the table name to unload the data.

USING collist
Specifies the column(s) to use with the command, where text may also be inserted.

WHERE clause
Limits rows of data. For more information, see WHERE.

About the UNLOAD Command

Use UNLOAD to transfer tables or views from one database to another, or to back up a database. You can also use UNLOAD to free up space while using a temporary table.

The UNLOAD command creates a file with a .LOB extension for binary large objects, and the originating file that you specify for the data. Your originating file can NOT have a .LOB file extension.

Transferring Tables and Views

UNLOAD does not change the data or structure of the original database. UNLOAD DATA with the AS ASCII or AS CSV option unloads computed column values as well as non-computed columns.

If the UNLOAD AS CSV syntax has been used, you can use the LOAD AS CSV command to restore the data.

Backing up a Database

R:BASE unloads data in ASCII delimited format: values are separated by the current delimiter and all text strings are enclosed in quotation marks. UNLOAD creates a file containing commands that set special characters, such as commas and quotation marks. The setting of the SET WIDTH condition effects the width of data lines in the unloaded file

If the database has columns defined as binary or text large objects, then UNLOAD creates two files, one file containing the R:BASE commands, and a second file (with a .LOB extension) containing the large object data. Both files are needed to transfer the information back into an R:BASE database. Your originating file can NOT have a .LOB file extension.

Unloading Temporary Tables

Use the UNLOAD *tblname* command to backup individual temporary tables created when STATICDB is set on-which activates a read-only schema mode. When UNLOAD is used to backup temporary tables, it generates a SET STATICDB OFF command to be executed prior to the CREATE SCHEMA command.

Examples

Example 01:

The following command lines unload only the data from the *product* table to a file named MYFILE.DBS. The data is in ASCII delimited format. The OUTPUT SCREEN command redirects the output back to the screen and closes the file.

```
OUTPUT myfile.dbs
UNLOAD DATA FOR product AS ASCII
OUTPUT SCREEN
```

Example 02:

In the example below the a file will be created that contains Comma Separated Values with no headings and no page breaks.

```
SET HEADINGS OFF
SET LINES 0
SET WIDTH 200
OUTPUT myfile.csv
UNLOAD DATA FOR Employees AS CSV
OUTPUT SCREEN
```

The commands above might create the file below. Notice that Jane Dough has Quotes surrounding her address. This is because the text contains an embedded comma.

```
Robert,Smith,123 Main St,Denver,CO,Support
Jane,Dough,'98 Folk St, Apt 1',Pittsburgh,PA,Sales
Matt,Follows,14 Arrowhead Ln,Portsmouth,RI,Services
```

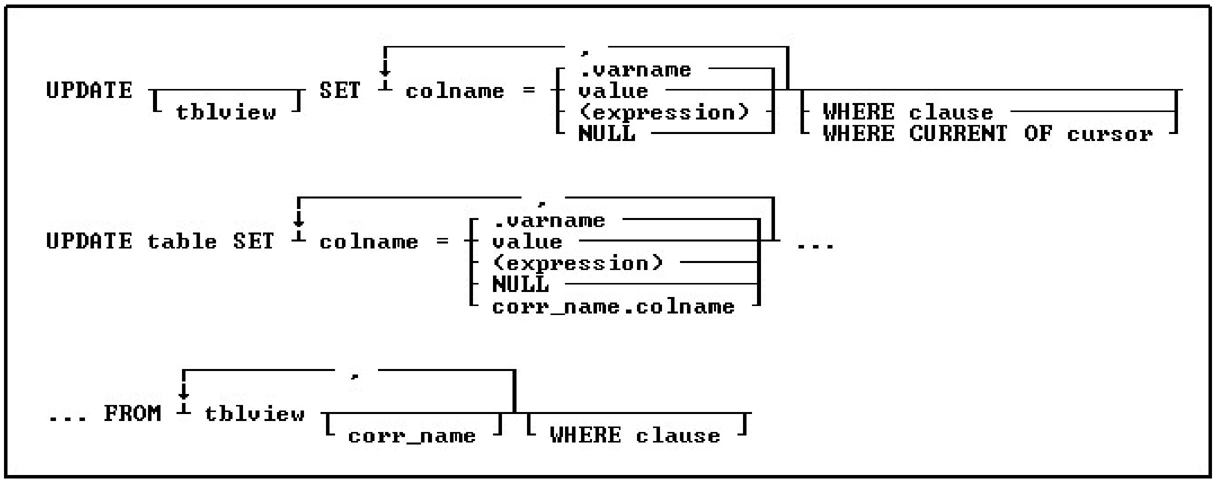
Example 03:
The following unloads the first name, last name, and the preceded text "Contact Name" within the ASCII output

```
UNLOAD DATA FOR Contact USING 'Contact Name:',ContFName,ContLName AS ASCII
```

Copyright © 1982-2025 R:BASE Technologies, Inc.

6.17.2 UPDATE

Use the UPDATE command to change the data in one or more columns in a table or a view.



Options

,
Indicates that this part of the command is repeatable.

(expression)
Determines a value using a text or arithmetic formula. The expression can include other columns from the table, constant values, functions, or system variables such as *#date*, *#time*, and *#pi*.

FROM tbllist
Specifies a list of tables from which data can be retrieved and updated.

NULL
Sets the values in the column equal to null.

SET colname
Specifies the column to update.

table
Specifies a table.

tblview
Specifies a table or view. If no table or view name is included, columns will be updated in all tables containing the specified columns, according to the conditions of the WHERE clause.

value
Specifies a value to enter in the specified column.

.varname
Specifies a global variable that provides a value for a column.

WHERE clause
Limits rows of data. For more information, see WHERE.

WHERE CURRENT OF cursor
Specifies a cursor that refers to a specific row to be affected by the UPDATE command. With this option, you must specify *tblview*.

About the UPDATE Command

The UPDATE command is useful for adjusting values in columns that require uniform changes.

The UPDATE command only modifies data in columns in one table or view. You can also update a table by referencing values from another table. You can modify a column's value by doing the following:

- Entering a new value for the column as a constant or variable
- Entering an expression that calculates a value for the column
- Entering a null value

Notes:

- Only users that have been granted rights to update the table(s) or column(s) can run the UPDATE command.
- R:BASE complies with defined rules, even for columns not affected by the update. If an update breaks a rule, the update is not processed.
- You cannot use UPDATE with computed or autonumbered columns. To change a computed column value, change the values in the columns to which the computed column refers.
- The UPDATE command will not update data in a multi-table View (a View based on multiple tables), as the data is not editable.
- A View with a GROUP BY parameter is also not editable.

Updating Column Values

You can update a column with a specific value. The value you use must meet the requirements of the column's data type, for example, a numeric column cannot be loaded with a text value.

Use the current delimiter character (the default is a comma) to separate each column and its new value from the next column and value.

Use the following guidelines when modifying data with UPDATE:

- Do not embed commas within entries for CURRENCY, DATE, DATETIME, DOUBLE, INTEGER, NUMERIC, or REAL data types. R:BASE automatically inserts commas and the current currency symbol.
- When values for CURRENCY, DOUBLE, NUMERIC, or REAL or data types are decimal fractions, you must enter the decimal point. When values are whole numbers, R:BASE adds a decimal point for you at the end of the number. R:BASE adds zeros for subunits in whole currency values. For example, using the default currency format, R:BASE loads an entry of 1000 as $1,000.00.
- When values for NOTE or TEXT data types contain commas, you can either enclose the entries within quotes, or use SET DELIMIT to change the default delimiter (comma) to another character.
- When values for NOTE or TEXT data types contain single quotes ('), and you are using the default QUOTES character ('), use two single quotes ('') in the text string. For example, 'Walter Finnegan''s order.'
- When values for NOTE or TEXT data types exceed the maximum length of a column, R:BASE truncates the value and adds it to the table. A message is displayed that tells you which row has been truncated.

Using an Expression or Variable

Enclose expressions in parentheses. If you use global variables in an expression, dot the variable (*.varname*). If expressions contain values that have a TEXT data type, enclose the values within quotes. The default QUOTES character is the single quote (').

If you attempt to use a null value in an expression or computed column, the result of the expression is null. However, if you set ZERO to on, R:BASE treats null values as zeros and processes expressions as if the null value were zero.

Using the WHERE Clause

If an UPDATE command includes a table or view name, you do not need to specify a WHERE or WHERE CURRENT OF clause. All rows will be updated. If you use a WHERE CURRENT OF clause, you must include a table or view name in the command.

If you omit a table or view name, you must use a WHERE clause with the UPDATE command so that you do not change values in more rows than you intended to change. The WHERE clause pinpoints the rows you want to change. If any columns exist in more than one table, all occurrences are changed if the column value meets the WHERE clause conditions. Test the WHERE clause by using the SELECT command before using the clause with UPDATE command. By using a WHERE clause with a SELECT command, you can view the rows you want to change before changing them.

R:BASE takes significantly less time to process a WHERE clause if one of the columns specified in the clause is an indexed column.

Using UPDATE with Transaction Processing

If more than one person at a time executes an UPDATE command and transaction processing is on, R:BASE might not execute the command concurrently. If you hold an UPDATE lock, you can read, modify, or delete any row in a table. R:BASE blocks any additional requests for UPDATE until other SELECT or UPDATE locks are cleared.

Examples

The following command changes values in the *company* and *custphone* columns of the *customer* table for the row where *custid* equals 100.

```
UPDATE customer SET company = 'Quality Computers', +
custphone = '617-341-3762' WHERE custid = 100
```

The following command changes the *invoicetotal* column in the *transmaster* table to the value of the expression (*invoicetotal * .9*) for rows where *transid* is greater than 5000.

```
UPDATE transmaster SET invoicetotal = ( invoicetotal * .9) +
WHERE transid > 5000
```

The following command changes the *listprice* column to the value of the expression (*1.1 * listprice*) for every row in the *prodlocation* table containing an entry in the *listprice* column.

```
UPDATE prodlocation SET listprice = (1.1 * listprice) +
WHERE listprice IS NOT NULL
```

The following command adds to the set of conditions in the above command. The command below extracts all of the selling prices from the *transdetail* table and requires that *listprice* be changed only if it matches a current selling price in the table.

```
UPDATE product SET listprice = (1.1 * listprice) +
WHERE listprice IS NOT NULL AND model = 'CX3000' +
AND listprice IN (SELECT price FROM transdetail +
WHERE model = 'CX3000')
```

The following command changes the *onhand* column in the *prodlocation* table (specified by cursor *curs1*) to the value of the expression (*onhand - 100*). The changes are made only in the row currently referenced by the cursor.

```
UPDATE prodlocation SET onhand = (onhand - 100) +
WHERE CURRENT OF curs1
```

The following example shows interactive data updating in an application file. The value of *var1* is used in the expression that is assigned to the *onhand*column of the *prodlocation* table. The UPDATE command changes values in *onhand* to the value of the expression (*onhand - .var1*) for all rows containing model numbers that begin with the letter C. The wildcard character % indicates one or more additional characters.

```
SET VARIABLE var1 TEXT
DIALOG 'Enter quantity by which to reduce inventory: '  var1 vend 1
SET VARIABLE var1 INTEGER
UPDATE prodlocation SET onhand = (onhand - .var1) +
WHERE model LIKE 'C%'
```

The following command changes the last names of two employees. This command omits the table name, thereby causing a global change to all tables that meet the WHERE clause criteria.

```
UPDATE SET emplname TO 'Smith-Simpson' WHERE +
(empfname = 'Mary' AND emplname = 'Simpson') OR +
(empfname = 'John' AND emplname = 'Smith')
```

The following example corrects a problem that can occur with an incorrect date sequence setting. For example, assume that you had the date sequence set to a four-digit year when you entered transactions, and you entered dates with a two-digit year (3/1/93). The dates would be stored as 3/1/0093. And, if you wanted the date to be in the 20th century, you could use the UPDATE command to modify the existing dates to 20th century dates by adding 1900 years to each date, with the ADDYR function.

The SET DATE command makes sure that you are using a four-digit year. The UPDATE command changes all *transdate* values to 20th century dates, where the current value of the column is less than 1/1/1900. The last SET DATE command returns to a two-digit date sequence and format.

```
SET DATE MM/DD/YYYY
UPDATE transmaster SET transdate = (ADDYR(transdate,1900)) +
WHERE transdate < 1/1/1900
SET DATE MM/DD/YY
```

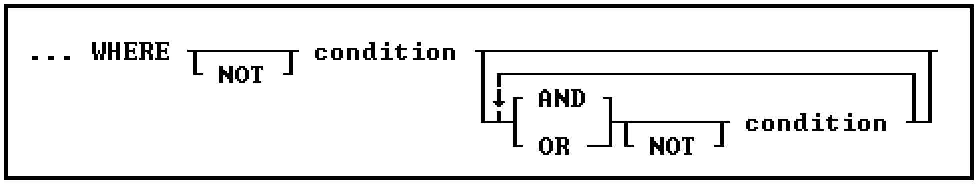
Assume that you wanted to update the *inventory* table with the sum of the units sold from the *orders* table. Because there are many rows in the *orders* table for each part number, you cannot do this directly with the UPDATE command. The CREATE VIEW command creates a view containing the sum of the units sold from the *orders* table. The UPDATE command updates the *inventory* table by extracting the *totalsold* value from the view named *orders_view* for each part number.

```
CREATE VIEW orders_view (partid,totalsold) AS SELECT +
partid, sum(sold) FROM orders GROUP BY partid

UPDATE inventory SET onhand = (T1.onhand - T2.totalsold) +
FROM inventory T1, orders_view T2 +
WHERE T1.partid = T2.partid
```

6.18 W

6.18.1 WHERE

Use a WHERE clause in commands to qualify or restrict the rows affected by a command.



Options

AND
Indicates the following condition must be met along with the preceding condition.

condition
Identifies requirements to be in the WHERE syntax.

NOT

Reverses the meaning of a connecting operator. AND NOT, for example, indicates that the first condition must be met and the following condition must not be met.

OR
Indicates the following condition can be met instead of the preceding condition.

About the WHERE Clause

In most commands, a WHERE clause follows the syntax diagram above.

The two main elements in any WHERE clause are conditions and connecting operators.

We now support "COUNT = LAST" in two different ways. If the entire WHERE clause is "WHERE COUNT = LAST" then R:BASE works like it always has to quickly fetch the last row of the table. The NEW functionality is to have other conditions in the WHERE clause and you want the last row of whatever qualifies.

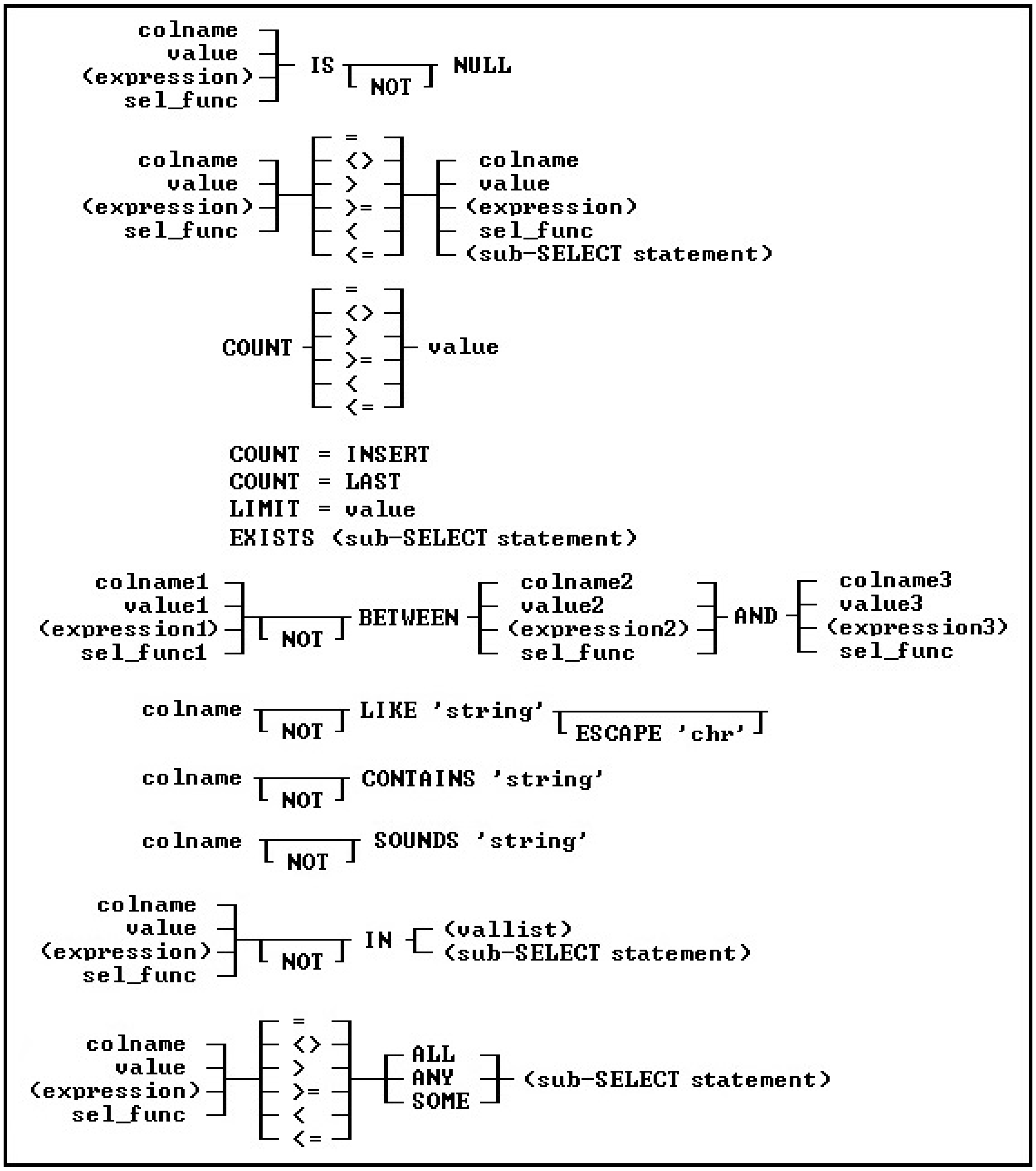
To make it work this way specify the other conditions and then add "AND COUNT = LAST".

Here is an example:

```
SELECT * FROM Customer WHERE CustID > 100 AND COUNT = LAST
```

WHERE Clause Conditions

The following syntax diagram and table show the basic formats for WHERE clause conditions, which can be used alone or together.



Basic WHERE Clause Conditions

Condition Syntax	Description
colname op DEFAULT	True if a column value compares correctly with the DEFAULT value for the column. *Op* can be =, <>, >=, >, <=, or <.
colname = USER	True if a column value equals the current user identifier.
item1 IS NULL	True if *item1* has a null value. *Item1* can be a column name, value, or expression. A null value cannot be used in a comparison with an operator.
item1 op item2	True if the relationship between two items is true as defined by an operator. *Item1* can be a column name, value, or expression; *item2* can be a column name, value, expression, or sub-SELECT statement.

COUNT=INSERT	Refers to the last row inserted in a table by the current user, even if it has been modified by another user. The COUNT=INSERT condition can be used with a single-table view, but not with a multi-table view. If there is not a newly inserted row in the table, then COUNT=INSERT performs the same action as COUNT=LAST, and fetches the current end row of the table.
COUNT=LAST	Refers to the last row in a table. The COUNT=LAST condition can be used with a single-table view, but not with a multi-table view.
COUNT *op value*	Refers to a number of rows defined by *op* and *value*.
LIMIT=*value*	Specifies a number of rows affected by a command. A LIMIT condition should be the last condition in a WHERE clause.
EXISTS (sub-SELECT statement)	True if sub-SELECT statement returns one or more rows.
item1 BETWEEN *item2* AND *item3*	True if the value of *item1* is greater than or equal to the value of *item2*, and if the value of *item1* is less than or equal to the value of *item3*.
colname LIKE '*string* '	True if a column value equals the text string. With LIKE, a string can also be a DATE, TIME, or DATETIME value. The text string can contain R:BASE wildcard characters.
colname LIKE '*string* ' ESCAPE '*chr* '	True if a column value equals a text string. If you want to use a wildcard character as a text character in the string, specify the ESCAPE character 1*chr*. In the string, use *chr* in front of the wildcard character.
colname CONTAINS '*string* '	True if a column value contains the text string.
colname SOUNDS '*string* '	True if the soundex value of a column matches the soundex value of the text string.
item1 IN (*vallist*)	True if *item1* is in the value list.
item1 IN (sub-SELECT statement)	True if *item1* is in the rows selected by a sub-SELECT.
item1 op ALL (sub-SELECT statement)	True if the relationship between *item1* and every row returned by a sub-SELECT statement matches an operator.
item1 op ANY(sub-SELECT statement)	True if the relationship between *item1* and at least one value returned by a sub-SELECT statement matches an operator.
item1 op SOME (sub-SELECT statement)	ANY and SOME are equivalent.

Notes:

- Placing NOT before most text operators (such as NULL or BETWEEN) reverses their meaning.
- When a SELECT statement is part of a WHERE clause, it is called a sub-SELECT clause. A sub-SELECT clause can contain only one column name (not a column list or *), expression, or function. The INTO and ORDER BY clauses in a sub-SELECT are ignored.

You can only use the current wildcard characters to compare a column to a text value when using the LIKE comparison. The default wildcard characters are the percent sign (%), which is used for one or more characters, and the underscore (_), which is used for a single character.

If you compare a column with a value, you can either enter the value or specify a global variable. If you specify a variable, R:BASE compares the column with the current value of the variable.

To significantly reduce processing time for a WHERE clause, use INDEX processing. To use indexes, the following conditions must be met:

- A condition in the WHERE clause compares an indexed column.
- If the WHERE clause contains more than one condition, R:BASE selects the condition that places the greatest restriction on the WHERE clause.
- Conditions are not joined by the OR operator.
- The comparison value is not an expression.

Connecting Operators

When you use more than one condition in a WHERE clause, the conditions are connected using the connecting operators AND, OR, AND NOT, and OR NOT.

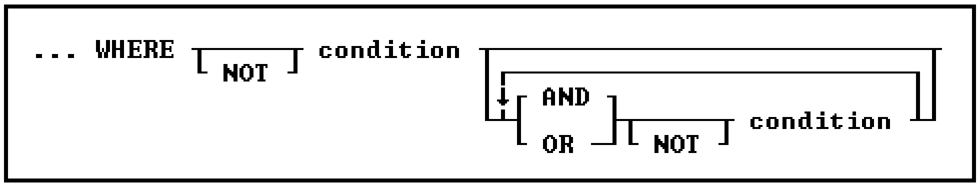
The connecting operator AND requires that both conditions it separates must be satisfied. The connecting operator OR requires that either condition it separates must be satisfied.

The connecting operator AND NOT requires that the preceding condition must be satisfied, and the following condition must not be satisfied. The connecting operator OR NOT requires that either the preceding condition must be satisfied, or any condition except the following condition must be satisfied.

In WHERE clauses with multiple conditions, conditions that are connected by AND or AND NOT are evaluated before those connected by OR or OR NOT. However, you can control the order in which conditions are evaluated by either placing parentheses around conditions or using the SET AND condition. If you set AND off, conditions are always evaluated from left to right.

WHERE Builder

When launching the WHERE Clause Builder, the following window will appear:



Examples

The following WHERE clause chooses sales amounts that are less than the value of a variable containing the daily average.

```
... WHERE amount < .dailyave
```

The following WHERE clause specifies the seventh row.

```
... WHERE COUNT = 7
```

The following WHERE clause specifies each row from the *employee*table that contains both the first name *June* and the last name *Wilson*.

```
SELECT * FROM employee WHERE empfname = 'june' AND emplname = 'wilson'
```

The following WHERE clause selects dates in the *actdate* column that are greater than dates in the *begdate* column or are less than dates in the *enddate* column.

```
... WHERE actdate BETWEEN begdate AND enddate
```

The next three WHERE clauses use the following data:

```
empfname emplname
-------- --------
    Mary Jones
    John Smith
   Agnes Smith
    John Brown
```

In both of the following clauses, R:BASE first evaluates the conditions connected by AND, selecting John Smith. Then R:BASE adds any Marys to the list because the connecting operator is OR. The final result includes John Smith and Mary Jones.

```
...WHERE empfname = 'Mary' OR empfname = 'John' +
 AND emplname = 'Smith'
```

```
...WHERE empfname = 'Mary' OR (empfname = 'John' +
 AND emplname = 'Smith')
```

By moving the parentheses around the conditions connected by OR, you can select only John Smith. In the following WHERE clause, the first name can be either Mary or John, but the last name must be Smith.

```
...WHERE (empfname = 'Mary' OR empfname = 'John') AND +
 emplname = 'Smith'
```

The following example illustrates a sub-SELECT in a WHERE clause. Assume you wanted a list of all sales representatives that had transactions greater than $100,000, and the information for such a list was contained in two tables, *employee* and *transmaster*. The relevant columns in these tables are:

```
employee transmaster
empid  emplname empid    netamount
----- --------- ----- ------------
  102 Wilson      133   $32,400.00
  129 Hernandez   160    $9,500.00
  133 Coffin      129    $6,400.00
  165 Williams    102  $176,000.00
  166 Chou        160  $194,750.00
  167 Watson      129   $34,125.00
  160 Smith       131  $152,250.00
  131 Simpson     102   $87,500.00
  102                   $22,500.00
  102                   $40,500.00
  131                  $108,750.00
  131                   $80,500.00
  129                   $56,250.00
  102                   $57,500.00
  160                  $140,300.00
  129                   $95,500.00
  129                  $155,500.00
  133                   $88,000.00
  131                  $130,500.00
  102                    $3,060.00
  165                    $3,060.00
  167                    $3,830.00
  133                   $12,740.00
  165                   $26,310.00
```

To display a list of employees in the *transmaster* table with a transaction larger than $100,000, enter the following command:

```
SELECT empid, emplname FROM employee WHERE empid IN +
   (SELECT empid FROM transmaster WHERE netamount > 100000)
```

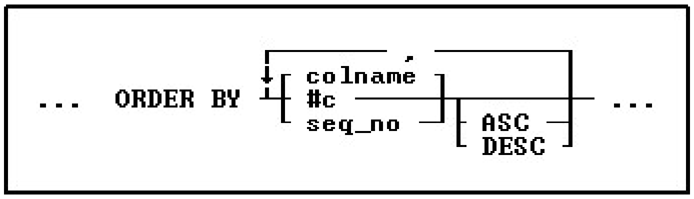
R:BASE displays the following list:

```
    empid emplname
--------- ----------------
      102 Wilson
      129 Hernandez
      131 Simpson
      160 Smith
```

Note: You can use a sub-SELECT in any command that allows a full WHERE clause.

6.18.1.1 ORDER BY

Use the ORDER BY clause with an R:BASE command to specify the order in which rows of data from a table are displayed.



Options

,
Indicates that this part of the command is repeatable.

ASC
DESC
Specifies whether to sort a column in ascending or descending order.

#c
Takes the place of a column name and refers to the column numbers displayed with the LIST TABLE command.

colname
Sorts by any column name or combination of column names.

seq_no
Refers to the items listed in the SELECT command that is using the ORDER BY command, ordered from left to right. An item can be a column name, expression, or SELECT function.

About the ORDER BY Command

The syntax for the ORDER BY clause is the same for all commands. ORDER BY must refer to only one table or view.

You can significantly reduce the time R:BASE takes to process an ORDER BY clause when the column or columns listed in the ORDER BY clause are included in an index with the same column sort order as that specified in the ORDER BY clause.

Using the SET SORT Command
The ORDER BY command uses the R:BASE automatic sort optimizer. If you are sorting extremely large tables, and if your disk space is limited, the automatic sort optimizer might be unable to sort the data. Instead, use the SET SORT ON command because it uses the least disk space necessary to sort data; however, the SET SORT ON command is slower than the automatic sort.

Examples

The following command displays data from the *custid*, *company*, and *custcity* columns from the *customer*table.

```
SELECT custid, company, custcity FROM customer
```

The ORDER BY clause in the command below arranges the *custid*values in descending order.

```
SELECT custid, company, custcity FROM customer +
ORDER BY custid DESC
```

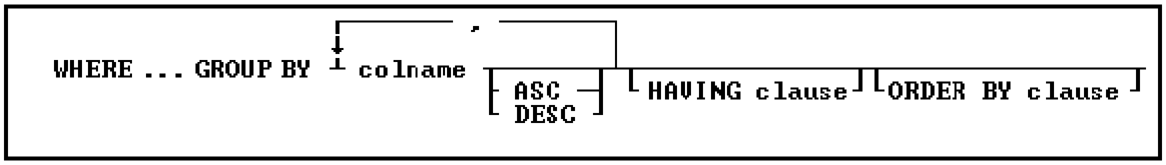
You can substitute a column's sequence number for a column named in the ORDER BY clause. You must use a sequence number when referring to an expression, function, constant, or when a UNION operator is used. The following command is equivalent to the command example above.

```
SELECT custid, company, custcity FROM customer ORDER +
BY 1 DESC
```

R:BASE for DOS only: You can also specify the maximum and minimum memory allocated with the SET SORT command using the MAX and MIN functions. You can show the current memory allocation settings with SHOW SORT using the MAX, MIN, and LAST functions-LAST shows the amount of memory you need to perform the last sort.

6.18.1.2 GROUP BY

This clause determines which rows of data to include.



Options

,
Indicates that this part of the command is repeatable.

ASC
DESC
Specifies whether to sort a column in ascending or descending order.

colname
Specifies a column name. The column name is limited to 128 characters.

In a command, you can enter *#c*, where *#c* is the column number shown when the columns are listed with the LIST TABLES command. In an SQL command, a column name can be preceded by a table or correlation name and a period (*tblname.colname*).

GROUP BY
Returns a groups of rows as a summary resulting in only unique rows. This option is generally used with SELECT commands.

HAVING clause
Limits the rows affected by the GROUP BY clause.

ORDER BY clause
Sorts rows of data.

About the GROUP BY command

This optional clause groups rows according to the values in one or more columns and sorts the results. GROUP BY consolidates the information from several rows into one row. This results in a table with one row for each value in the named column or columns and one or more values per column.

The columns listed in the GROUP BY clause are related to those listed in the command clause. Any column named in the GROUP BY clause can also be named in the command clause, but any column not named in the GROUP BY clause can be used only in the command clause if the column is used in a SELECT command.

Examples

The SELECT command clause can contain the columns named in the GROUP BY clause, and SELECT functions that refer only to columns not named in the GROUP BY clause. Because the GROUP BY clause processes information resulting from a WHERE clause, you can add a GROUP BY clause to see the sales each employee has made:

```
SELECT empid FROM transmaster WHERE netamount < $100,000 +
GROUP BY empid
```

The following intermediate result table contains columns not named in the command clause because the command clause has not been processed yet (not all the columns fit in the display, however). The first part of the processing is to group the rows by *empid.* Because seven different employees are included, the intermediate result table includes seven rows.

Intermediate Result Table-GROUP BY empid

transid	custid	empid	netamount
4975, 4980, 5000, 5060, 5045	101, 101, 101, 101, 100	102	$87,500, $22,500, $40,500, $57,500, $3,060
4790, 4865, 5050, 5070	104, 102, 104, 104	129	$6,400, $34,125, $56,250, $95,500
5015	103	131	$80,500
4760, 5080, 5048	100, 100, 103	133	$32,400, $88,000, $12,740
4780	105	160	$9,500
5046, 5049	101, 102	165	$3,060, $26,310
5047	102	167	$3,830

You can include more than one column in a GROUP BY clause. If you group the rows in the above example by *custid* as well as *empid*, the command looks like this:

```
SELECT empid, custid FROM transmaster +
WHERE netamount < $100,000 GROUP BY empid, custid
```

In the following table, rows are now grouped by both *empid* and *custid*, resulting in eleven groups.

Intermediate Result Table-GROUP BY empid and custid

transid	custid	empid	netamount
5045	100	102	$3,060
4975, 4980, 5000, 5060	101	102	$87,500, $22,500, $40,500, $57,500
4865	102	129	$34,125
4790, 5050, 5070	104	129	$64,000, $56,250, $95,500
5015	103	131	$80,500
4760, 5080	100	133	$32,400, $88,000
5048	103	133	$12,740
4780	105	160	$9,500
5046	101	165	$3,060
5049	102	165	$26,310
5047	102	167	$3,830

If one or more of the columns named in the GROUP BY clause contain null values, R:BASE forms a separate group for null values. Review the result of this SELECT command for the *employee* table:

```
SELECT empid, emplname, hiredate, emptitle FROM employee
```

```
empid        emplname     hiredate     emptitle
102          Wilson       03/18/90     Manager
129          Hernandez    08/28/91     Manager
131          Smith        04/14/92     -0-
133          Coffin       11/26/93     Representative
160          Simpson      01/09/94     -0-
165          Williams     07/05/92     Representative
167          Watson       07/10/92     Representative
```

```
166             Chou            07/10/93        Sales Clerk
```

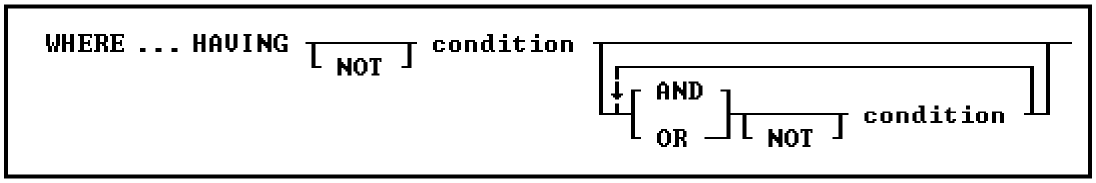
If you group these rows by the *emptitle* column, which contains null values, you get the following intermediate result table:

Intermediate Result Table-GROUP BY emptitle

empid	emplname	hiredate	emptitle
102, 129	Wilson, Hernandez	03/18/90, 08/28/91	Manager
133, 165, 167	Coffin, Williams, Watson	11/26/93, 07/05/92, 07/10/92	Representative
166	Chou	07/10/93	Sales Clerk
131, 160	Smith, Simpson	04/14/94, 01/09/94	-0-

6.18.1.3 HAVING

This clause determines which rows of data to include based on the results of a prior GROUP BY clause.



Options

AND
OR
AND indicates two conditions must both be true.
OR indicates either condition must be true.

condition
Specifies a combination of one or more expressions and/or operations that would evaluate to either true or false. See the "HAVING Conditions" below.

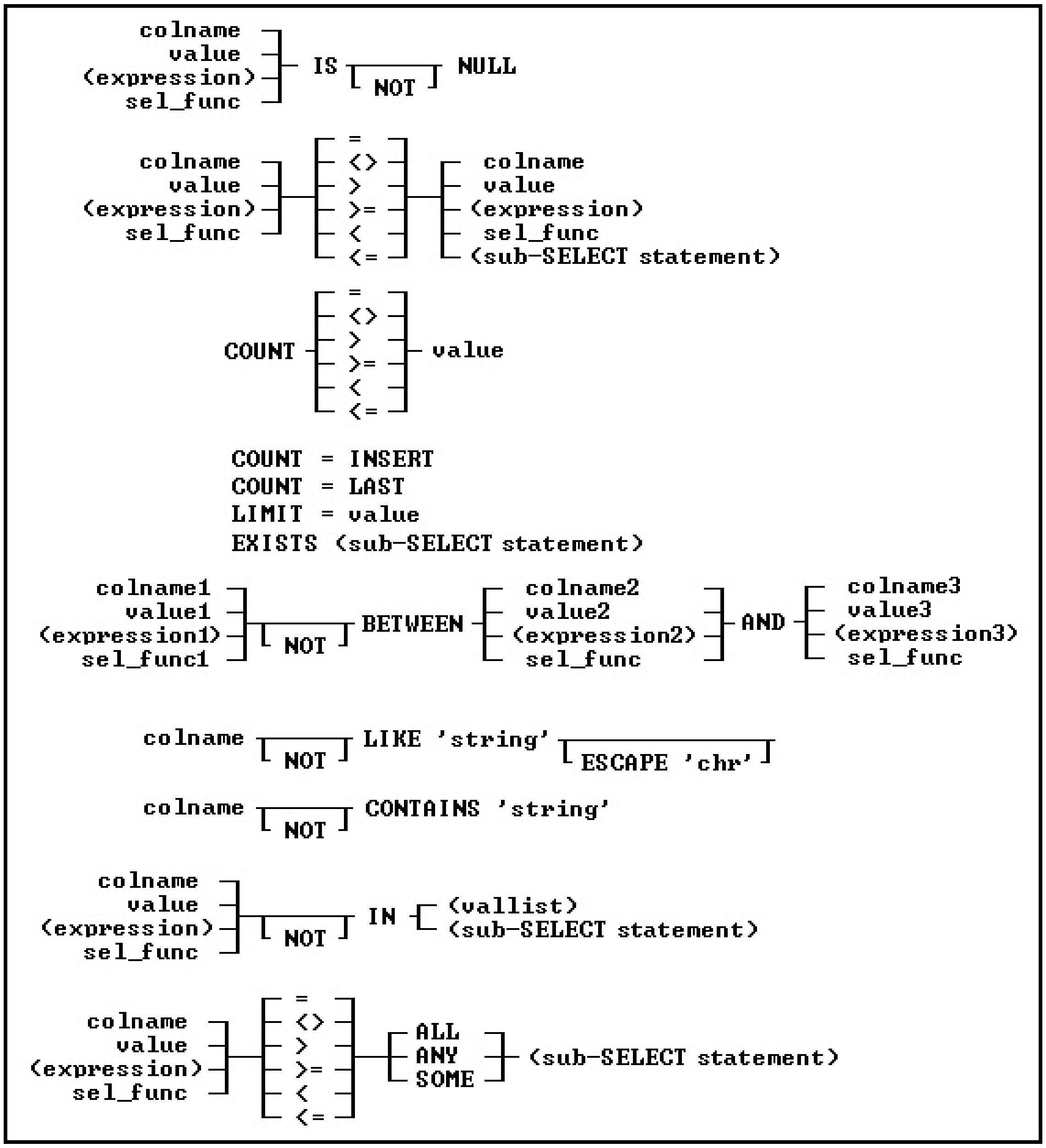
NOT
Reverses the meaning of an operator or indicates that a condition is not true.

About the HAVING command

The optional HAVING clause selects rows that meet one or more conditions from among the results of the GROUP BY clause. HAVING works the same as a WHERE clause with the following exceptions:

- A WHERE clause modifies the intermediate results of a FROM clause; a HAVING clause modifies the intermediate results of a GROUP BY clause.
- A HAVING clause can include SELECT functions.

HAVING Conditions:



Examples

To display sales information for only those employees who have made more than one sale to the same customer, add a HAVING clause such as the following to one of the examples shown previously in GROUP BY. When used in a HAVING clause, SELECT functions compute results based on the values grouped in the specified column. In this HAVING clause, COUNT returns the number of values grouped in the *transid* column.

```
SELECT empid, custid FROM transmaster +
WHERE netamount < $100,000 +
GROUP BY empid, custid HAVING COUNT(transid) > 1
```

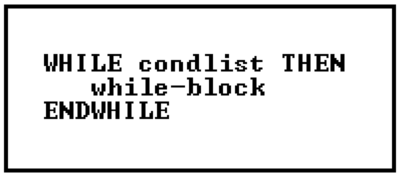
Intermediate Result Table-HAVING COUNT(transid) > 1

transid	custid	empid	netamount

```
4975, 4980, 5000, 5060      101    102    $87,500, $22,500, $40,500, $57,500

4790, 5050, 5070            104    129    $6,400, $56,250, $95,500
5080                        100    133    $32,400, $88,000
```

6.18.2 WHILE/ENDWHILE

Use the WHILE...ENDWHILE structure in a program to continuously run a set of commands based on a specified condition.



```
WHILE condlist THEN
   while-block
ENDWHILE
```

Options

condlist
Specifies a list of conditions that identify the requirements to be met.

while-block
Specifies commands to be executed if the WHILE condition is true.

About the WHILE...ENDWHILE Command

A WHILE ... ENDWHILE structure consists of conditions, commands, and an ENDWHILE statement. As long as WHILE conditions are true, R:BASE runs the commands repeatedly.

WHILE Conditions

The basic WHILE conditions are the same as those used in an IF...ENDIF structure and are as follows:

Condition	**Description**
varname IS NULL	The value of the variable is null.
varname IS NOT NULL	The value of the variable is not null.
varname CONTAINS *'string'*	The variable has a TEXT data type and contains a '*string*' as a substring in the variable value.
varname NOT CONTAINS *'string'*	The variable has a TEXT data type and a *'string'* is not contained as a substring in the variable value.
varname LIKE *'string'*	The variable equals a '*string*.' A *'string'* can contain wildcards.
varname NOT LIKE *'string'*	The variable does not equal the *'string*'. A '*string'* can contain wildcards.
varname BETWEEN *value1* AND *value2*	The value of the variable is greater than or equal to *value1* and less than or equal to *value2*. The variable and the values must be the same data type.
varname NOT BETWEEN *value1* AND *value2*	The value of the variable is less than *value1* or greater than *value2*. The variable and the values must be the same data type.
item1 op *item2*	*Item1* has the specified relationship to *item2. Item1* can be a column name, value, or expression; *item2* can be a column name, value, or expression.

A variable can be substituted for the first variable in each of the condition formats shown above, and for either item when using an operator comparison. The condition should not use dotted variables, unless the current value of that variable is to be evaluated. When comparing items with an operator, (e.g. item1 < item2, item1 >= item2, etc.) the condition may be enclosed in parentheses, where R:BASE will evaluate the expression each time through the loop.

You can only use wildcard characters with the LIKE and NOT LIKE operators. For example, *varname* LIKE '*string%.*'

You can combine conditions from the WHILE condition list by using the connecting operators AND, OR, AND NOT, and OR NOT. Be careful when using these conditions in a condition list. Conditions connected by AND are evaluated first, then conditions connected by OR are evaluated. However, you can use parentheses to set the evaluation order.

WHILE Loop Commands

All WHILE loop commands are retained in memory, so a WHILE loop runs more quickly than a GOTO or LABEL structure. A computer must have enough available memory to read all of the commands in a WHILE loop, or the program terminates abnormally.

R:BASE optimizes commands in a WHILE loop so that it runs more quickly. Use the following guidelines when constructing WHILE loops so they run more quickly.

- Do not clear variables in the WHILE loop. Rather, set those variables to null.
- Do not define variables within the WHILE loop. Only define variables outside of the loop because the values can change within the loop.
- If you issue multiple SET VARIABLE commands on a single command line, then those variables will not be optimized. If you want to increase the speed for that loop, you should put the SET VARIABLE commands on separate lines.

To turn off WHILE loop optimization, set WHILEOPT off.

The ENDWHILE Statement

ENDWHILE indicates the end of the loop. Place an ENDWHILE statement at the end of each WHILE loop. Each time R:BASE reaches the ENDWHILE statement, R:BASE returns to the WHILE command at the top of the loop and checks to see whether the conditions are still true or false. If true, R:BASE again runs the commands between the WHILE and the ENDWHILE. If false, R:BASE runs the command line immediately following the ENDWHILE.

Exiting from a WHILE Loop

To exit from a WHILE loop before the WHILE condition becomes false, use an IF...ENDIF structure to check other conditions, then use BREAK to exit from the WHILE loop. The BREAK command causes the WHILE loop to terminate when the conditions specified in the IF statement become true.

Never use GOTO to exit from a WHILE loop; use BREAK instead. BREAK clears the WHILE loop. When you do not use BREAK or the naturally occurring exit (that is, when the WHILE loop conditions are no longer true) to exit from a WHILE loop, R:BASE continues to read commands into memory. If you have a large command or procedure file, you can run out of memory and your program terminates abnormally.

Skip to the next WHILE Occurrence

Use the CONTINUE command to move to the next occurrence of the WHILE loop and run the code.

In the following example, when the code is run, processing returns to line 3 after it completes the CONTINUE command on line 6. The while-block commands in line 8 are not run.

```
SET VARIABLE v1=0
SET VARIABLE V2=1
WHILE v1 = 0 THEN
    *(while-block commands)
  IF v2 <> 0 THEN
    CONTINUE
  ENDIF
    *(while-block commands)
ENDWHILE
```

Example

In the following example, R:BASE runs the commands in the WHILE block and evaluates the *v2* condition in the IF statement. If *v2* is not equal to zero, R:BASE runs the BREAK command and terminates the WHILE loop. R:BASE then runs the commands immediately following the ENDWHILE statement. As long as the WHILE condition (*v1*) is true and the IF condition (*v2*) remains false, the WHILE loop continues processing.

```
SET VARIABLE v1 = 0
WHILE v1 = 0 THEN
   *(while-block commands)
   IF v2 <> 0 THEN
      BREAK
   ENDIF
   *(while-block commands)
ENDWHILE
*(next command outside the while-block
```

Part VII

7 R:BASE Database Functions

7.1 Function Categories

Arithmetic and Mathematical Functions

ABS	AVG	BRND	COUNT
DIM	EXP	LAVG	LISTOF
LMAX	LMIN	LOG	LOG10
LSTDEV	LSUM	LVARIANCE	MAX
MIN	MOD	PSTDEV	PVARIANCE
RANDOM	RNDDOWN	RNDUP	ROUND
SIGN	SQRT	STDEV	SUM
VARIANCE			

Conversion Functions

AINT	ANINT	CHAR	CTXT
DWRD	FLOAT	HTML	ICHAR
IHASH	INT	MAKEUTF8	NINT
SOUNDEX			

Database Utility Functions

CVAL	IINFO	TINFO

Date and Time Functions

ADDDAY	ADDFRC	ADDHR	ADDMIN
ADDMON	ADDSEC	ADDYR	DATETIME
DEXTRACT	DNW	DWE	GETDATE
IDAY	IDIM	IDOY	IDWK
IFRC	IHR	ILY	IMIN
IMON	ISEC	IWOY	IYR
IYR4	JDATE	RDATE	RTIME
TDWK	TEXTRACT	TMON	

Encryption Functions

DECRYPT	ENCRYPT

Financial Functions

FV1	FV2	PMT1	PMT2
PV1	PV2	RATE1	RATE2
RATE3	TERM1	TERM2	TERM3

Keyboard and Environment Functions

ENVVAL	CVAL	CHKCUR	CHKKEY
CHKFILE	CHKFUNC	CHKTABLE	CHKVAR
CVTYPE	DELFUNC	DLCALL	DLFREE
DLLOAD	FILENAME	FINDFILE	IFWINDOW
ISTAT	GETKEY	LASTKEY	

Logical Functions

IFCASEEQ	IFEQ	IFEXISTS	IFF
IFGE	IFGT	IFLE	IFLT
IFNE	IFNULL		

String Manipulation Functions

CTR	FORMAT	ICAP	ICAP1
ICAP2	ICAP3	ISALPHA	ISDIGIT
ISLOWER	ISSPACE	ISTR	ISUPPER

ITEMCNT	LJS	LTRIM	LUC
REVERSE	RJS	RTRIM	SFIL
SGET	SKEEP	SKEEPI	SLEN
SLOC	SLOCI	SLOCP	SMOVE
SPUT	SRPL	SSTRIP	SSTRIPI
SSUB	SSUBCD	STRIM	TRANSLATE
TRIM	ULC		

Trigonometric Functions

ACOS	ASIN	ATAN	ATAN2
COS	COSH	SIN	SINH
TAN	TANH		

7.2 A

7.2.1 ABS

(ABS(*arg*))

Returns the absolute or positive value of *arg* (a value with a DOUBLE, REAL, NUMERIC, or INTEGER data type).

In the following example, the value of *vabs* is *2*.

```
SET VAR vnum = -2
SET VAR vabs = (ABS(.vnum))
```

7.2.2 ACOS

(ACOS(*arg*))

Computes the arccosine of *arg* where *arg* is in the range -1 to 1. The result is an angle in radians between 0 and pi (where pi = 3.14159265358979).

In the following example, the value of *vacos* is *2.094395*, the arccosine of -0.5.

```
SET VAR vacos = (ACOS(-0.5))
```

7.2.3 ADDDAY

(ADDDAY(*date,int*))

Adds the specified number of days to a date or datetime value. Functions that result in an invalid date, for example, February 30, will return NULL.

7.2.4 ADDFRC

(ADDFRC(*time,int*))

Adds the specified number of milliseconds to a time or datetime value.

7.2.5 ADDHR

(ADDHR(*time,int*))

Adds the specified number of hours to a time or datetime value.

Copyright © 1982-2025 R:BASE Technologies, Inc.

7.2.6 ADDMIN

(ADDMIN(*time,int*))

Adds the specified number of minutes to a time or datetime value.

7.2.7 ADDMON

(ADDMON(*date,int*))

Adds the specified number of months to a date or datetime value.

7.2.8 ADDSEC

(ADDSEC(*time,int*))

Adds the specified number of seconds to a time or datetime value.

7.2.9 ADDYR

(ADDYR(*date,int*))

Adds the specified number of years to a date or datetime value.

7.2.10 AINT

(AINT(*arg*))

Truncates the decimal fraction, returning a whole number in the original REAL, NUMERIC, or DOUBLE data type.

In the following example, the value of *vaint* is *1*.

```
SET VAR vaint = (AINT(1.8))
```

7.2.11 ANINT

(ANINT(*arg*))

Rounds the decimal fraction to the nearest integer, returning a whole number in the original REAL, NUMERIC, or DOUBLE data type.

In the following example, the value of *vanint1* is *3.0* and the value of *vanint2* is *4.0*.

```
SET VAR vanint1 = (ANINT(2.6))
SET VAR vanint2 = (ANINT(4.45))
```

7.2.12 ASIN

(ASIN(*arg*))

Computes the arcsine of *arg* where *arg* is in the range -1 to 1. The result is an angle in radians between -pi/2 and pi/2.

In the following example, the value of *vasin* is *-0.5236* (-pi/6 radians)

```
SET VAR vasin = (ASIN(-0.5))
```

7.2.13 ATAN

(ATAN(*arg*))

Computes the arctangent of *arg* in radians where *arg* is any amount. The result is an angle in radians between -pi/2 and pi/2.

In the following example, the value of *vatan* is *0.7854* (pi/4 radians).

```
SET VAR vatan = (ATAN(1))
```

7.2.14 ATAN2

(ATAN2(*x,y*))

Computes the arctangent of *x/y*. The result is the angle in radians between -pi/2 and pi/2.

In the following example, the value of *vatan2* is *0.7854*.

```
SET VAR vatan2 = (ATAN2(1,1))
```

7.3 B

7.3.1 BRND

(BRND(*arg1,arg2,arg3*))

Rounds REAL, DOUBLE, or CURRENCY data to a specific number of decimal places and allows specification of the number of significant digits to return. *Arg1* is the value to be rounded. *Arg2* is the number of significant digits to return, and *arg3* is the precision The precision is specified as a decimal number, for example, .01 rounds to two decimal places.

In the following example, the value of *vresult* is 1234.57.

```
SET VAR vresult = (BRND(1234.5678342,8,.01))
```

7.4 C

7.4.1 CHAR

(CHAR(*integer*))

Converts an ASCII integer value to its corresponding character. This is not the same as the CHAR data type.

In the following example, the value of *vchar1* is *A* and the value of *vchar2* is *a*.

```
SET VAR vchar1 = (CHAR(65))
SET VAR vchar2 = (CHAR(97))
```

See also:

ASCII Character Chart

7.4.2 CHKCUR

(CHKCUR('*cursorname*'))

Checks to see if a cursor is declared and is not dropped. The function returns an integer value of 1 if the name exists and is not dropped, and 0 if it is not declared or dropped.

Copyright © 1982-2025 R:BASE Technologies, Inc.

7.4.3 CHKFILE

(CHKFILE('*filespec*'))

Checks to see if a file or folder name exists. If no path is specified, the function checks for the file or folder name in the current directory. Otherwise, the function checks for the file or folder name in the specified location.

The function returns a 1 if the file or folder name is found, and 0 if it is not found. Wildcards in the filename will produce unpredicatable results.

7.4.4 CHKFUNC

(CHKFUNC('function_name'))

Checks to see if a DLL function exists or not. If the DLL function exists, a 1 is returned. If the DLL function does not exist, a 0 is returned.

Example:

```
SET VAR v1 = (CHKFUNC('FunctionName'))
```

7.4.5 CHKKEY

(CHKKEY(0))

Returns an integer value of *1* if there are keystrokes available in the type-ahead buffer. Returns *0* if no keystrokes are available. Use CHKKEY before GETKEY to determine if a key is available.

CHKKEY does nothing with the zero that you enter in parentheses; CHKKEY returns a value without receiving one.

7.4.6 CHKTABLE

(CHKTABLE('*tblview*'))

Checks to see if a table/view exists. The function returns a value based upon the permanent or temporary nature of a table or view, and if a table is attached as a server or dBASE table.

Return Value	Status
0	table/view does not exist
1	permanent table
2	temporary table
3	server table
4	dBASE table
5	permanent view
6	temporary view

Examples:

Example 01:
```
SET VAR vCheckTable INTEGER = (CHKTABLE('Contact'))
SHOW VAR vCheckTable
1
```

Example 02:
```
SET VAR vCheckTable INTEGER = (CHKTABLE('QuarterlySummary'))
SHOW VAR vCheckTable
5
```

7.4.7 CHKVAR

(CHKVAR('var*name*'))

Checks to see if a variable name exists. The function returns an integer value of 1 if the variable name exists or declared, and 0 if it is not found or declared.

7.4.8 COS

(COS(*angle*))

Returns the trigonometric cosine of *angle*. The result is between -1 and 1.

In the following example, the value of *vcos* is *0.5002*.

```
SET VAR vcos = (COS(1.047))
```

7.4.9 COSH

(COSH(*angle*))

Returns the hyperbolic cosine of *angle*.

In the following example, the value of *vcosh* is *1.6000*.

```
SET VAR vcosh = (COSH(1.047))
```

7.4.10 CTR

(CTR(*text,width*))

Centers *text* in *width* characters, returning a text string.

In the following example, the value of *vctr* is   *ABCD*   .

The text string is centered in a 10-character field.

```
SET VAR vctr = (CTR('ABCD',10))
```

7.4.11 CTXT

(CTXT(*arg*))

Converts an internal value, returning a text string.

In the following example, the value of *vctxt1* is the value *37.6* and is a REAL data type. The value of *vctxt2* is the value *37.6* and is a TEXT data type. If you attempt to use *vctxt2* in a mathematical function, it will fail.

```
SET VAR vctxt2 = (CTXT(.vctxt1))
```

7.4.12 CVAL

(CVAL(*'showkeyword'*))

Returns the current value or setting of *'showkeyword'*. You must either enclose the SHOW keyword in quotation marks or use a dot variable that has a TEXT data type to which you have assigned the SHOW keyword. You can use all SHOW keywords with CVAL, as well as DATABASE, DBPATH, CURRDIR.

The following keywords can be used for (CVAL('keyword')):

- AND
- ANSI
- AUTOCOMMIT
- AUTODROP
- AUTOSKIP
- BELL
- BLANK
- BUILD
- CASE
- CLEAR
- CLIPBOARDTEXT
- COLOR
- COMPUTER
- CONNECTIONS
- CURRDIR
- CURRDRV
- CURRENCY
- CURRENTPRINTER
- DATABASE
- DATE
- DATE CENTURY
- DATE FORMAT
- DATE SEQUENCE
- DATE YEAR
- DBCOMMENT
- DBPATH
- DEBUG
- DELIMIT
- DRIVES
- ECHO
- EDITOR
- EOFCHAR
- EQNULL
- ERROR
- ERROR DETAIL
- ERROR VARIABLE
- ESCAPE
- EXPLODE
- FASTFK
- FASTLOCK
- FEEDBACK
- FILES
- FIXED
- HEADINGS
- IDQUOTES
- INSERT
- INTENSITY
- INTERVAL
- LAST BLOCK TABLE
- LAST ERROR
- LAYOUT
- LINEEND
- LINES
- LOOKUP
- MANOPT
- MANY
- MAXTRANS
- MDI
- MESSAGES
- MIRROR
- MULTI
- NAME
- NETUSER
- NOTE_PAD
- NULL

- OLDLINE
- ONELINE
- PAGEMODE
- PASSTHROUGH
- PLATFORM
- PLUS
- POSFIXED
- PORTS
- PRINTERS
- PRN_Status
- PRN_Orientation
- PRN_Size
- PRN_Source
- PRN_Quality
- PRN_Copies
- PRN_ColorMode
- PRN_DuplexMode
- PRN_Collation
- QUALCOLS
- QUOTES
- REFRESH
- REVERSE
- ROWLOCKS
- RULES
- SCRATCH
- SCREENSIZE
- SELMARGIN
- SEMI
- SERVER
- SINGLE
- SORT
- SORTMENU
- STATICDB
- TIME
- TIME FORMAT
- TIME SEQUENCE
- TIMEOUT
- TOLERANCE
- TRACE
- TRANSACT
- USER
- USERAPP
- USERDOMAIN
- USERID
- VERIFY
- VERSION
- VERSION BITS
- VERSION BUILD
- VERSION SYSTEM
- WAIT
- WALKMENU
- WHILEOPT
- WIDTH
- WINBEEP
- WINDOWSPRINTER
- WRAP
- WRITECHK
- ZERO
- ZOOMEDIT

Examples:

In the following example, the value of *vcval* is *OFF* if the value of MULTI is set to off.

```
SET VAR vcval = (CVAL('MULTI'))
```

In the following example, the user keyword is loaded into a variable and then the variable is used in the CVAL function. It returns the current user identifier.

```
SET VAR vword text = 'USER'
SET VAR vuser = (CVAL(.vword))
```

7.4.12.1 AND

(CVAL('AND'))

Returns the status of the AND command parameter (ON/OFF).

7.4.12.2 ANSI

(CVAL('ANSI'))

Returns the status of the ANSI command parameter (ON/OFF).

7.4.12.3 AUTOCOMMIT

(CVAL('AUTOCOMMIT'))

Returns the status of the AUTOCOMMIT command parameter (ON/OFF).

7.4.12.4 AUTODROP

(CVAL('AUTODROP'))

Returns the status of the AUTODROP display control (ON/OFF).

7.4.12.5 AUTOSKIP

(CVAL('AUTOSKIP'))

Returns the status of the AUTOSKIP command parameter (ON/OFF).

7.4.12.6 BELL

(CVAL('BELL'))

Returns the status of the BELL command parameter (ON/OFF).

7.4.12.7 BLANK

(CVAL('BLANK'))

Returns an empty variable value.

BLANK is similar to a null value, but is not based on your database NULL setting.

7.4.12.8 BUILD

(CVAL('BUILD'))

The BUILD parameter of the SHOW command can be used to determine the exact build number of the R:BASE Front-End GUI. This can also be used with CVAL to store the build number information in a variable.

See also:
(CVAL('VERSION')) to determine the version as well as build number of R:BASE Engine
(CVAL('VERSION BUILD')) to determine only the build number of the R:BASE Engine

7.4.12.9 CASE

(CVAL('CASE'))

Returns the status of the CASE command parameter (ON/OFF).

7.4.12.10 CLEAR

(CVAL('CLEAR'))

Returns the status of the CLEAR command parameter (ON/OFF).

7.4.12.11 CLIPBOARDTEXT

(CVAL('CLIPBOARDTEXT'))

Returns the contents of the Windows Clipboard.

7.4.12.12 COLOR

(CVAL('COLOR'))

Returns the value for the foreground and background COLOR of data displays.

This CVAL parameter is specific to R:BASE for DOS.

7.4.12.13 COMPUTER

(CVAL('COMPUTER'))

Returns the name of your computer.

Example:

```
SET VAR vComp = (CVAL('COMPUTER'))
```

7.4.12.14 CONNECTIONS

(CVAL('CONNECTIONS'))

Returns the number of users currently connected to the current database or 0 if not connected. In the event of a non-graceful disconnect R:BASE will not be able to decrement the connections count. This can occur if the network session terminates unexpectedly, or if the users operating system crashes. If this happens the count will be innacurate until all users disconnect from the database. This works because the last session of R:BASE will be able to tell that NO users are connected and will reset the count to zero. Unfortunatly, due to file system limitations, R:BASE is only able to tell if a database is open by any users or not at all, and is not able to tell, except by this count, how many users are connected.

7.4.12.15 CURRDIR

(CVAL('CURRDIR'))

Returns the current directory. This is the same information returned by using the CD command at the R> Prompt.

7.4.12.16 CURRDRV

(CVAL('CURRDRV'))

Returns the current drive. The drive is in X: format with the drive letter followed by a colon.

7.4.12.17 CURRENCY

(CVAL('CURRENCY'))

The CURRENCY data type holds monetary values of up to 23 digits represented in the currency format, established using SET CURRENCY. Amounts are in the range ±$99,999,999,999,999.99. Commas or the current delimiter can be used. If no decimal point is included, .00 is assumed.

Data is stored as two long integer values, reserving four bytes of internal storage.

7.4.12.18 CURRENTPRINTER

(CVAL('CURRENTPRINTER'))

This parameter returns the current printer for the R:BASE session.

7.4.12.19 DATABASE

(CVAL('DATABASE'))

Returns a text string containing the current connected database or NULL if the user is not connected to a database. This can be used to ensure that the user is connected before trying to execute some code. The example below shows how this might work.

```
SET VAR vDB = (CVAL('database'))
IF vDB IS NULL THEN
    CONN MyDB
ENDIF

SET VAR vDB = (CVAL('database'))
IF vDB IS NULL THEN
    PAUSE 2 USING 'MyDB is currently unavailable'
ENDIF
```

The check is repeated to ensure that the attempt to connect to the database was successful before continuing with the command file.

7.4.12.20 DATE

(CVAL('DATE'))

Returns the DATE format of the currently connected database.

7.4.12.21 DATE CENTURY

(CVAL('DATE CENTURY'))

Returns the default DATE CENTURY of the currently connected database.

7.4.12.22 DATE FORMAT

(CVAL('DATE FORMAT'))

Returns the default DATE FORMAT of the currently connected database.

7.4.12.23 DATE SEQUENCE

(CVAL('DATE SEQUENCE'))

Returns the default DATE SEQUENCE of the currently connected database.

7.4.12.24 DATE YEAR

(CVAL('DATE YEAR'))

Returns the default DATE YEAR of the currently connected database.

7.4.12.25 DBCOMMENT

(CVAL('DBCOMMENT'))

DBCOMMENT returns the comment for the current connected database.

7.4.12.26 DBPATH

(CVAL('DBPATH'))

DBPATH returns the full path to the current connected database.

7.4.12.27 DEBUG

(CVAL('DEBUG'))

Returns the status of the DEBUG command parameter(ON/OFF).

7.4.12.28 DELIMIT

(CVAL('DELIMIT'))

Returns the value of the DELIMIT character setting.

7.4.12.29 DRIVES

(CVAL('DRIVES'))

Returns the list of all currently available drives

Example:

```
SET VAR vDrives = (CVAL('Drives'))
```

vDrives will include the list of ALL drives!

Result: aCDeF

In that list of drives, drives with CAPITAL letters, such as C,D or F would be the hard disk drive, and drives with lower case letters would be removable drives, such as a or e.

a = floppy disk drive
C = Hard Disk
D = Hard Disk/CD-ROM/DVD
e = zip disk
F = Hard Disk/CD-ROM/DVD or even mapped network drive

All hard drives, including CD-ROM/DVD and network mapped drives would be CAPITAL letters.

All removable drives, including floppy disk drives and zip drives would be lower case letters.

7.4.12.30 ECHO

(CVAL('ECHO'))

Returns the status of the ECHO command parameter (ON/OFF).

7.4.12.31 EDITOR

(CVAL('EDITOR'))

Returns the current text editor used by R:BASE.

The default editor is RBEDIT.

7.4.12.32 EOFCHAR

(CVAL('EOFCHAR'))

Returns the status of the EOFCHAR command parameter (ON/OFF).

7.4.12.33 EQNULL

(CVAL('EQNULL'))

Returns the status of the EQNULL command parameter (ON/OFF).

7.4.12.34 ERROR

(CVAL('ERROR'))

Returns the status of the ERROR MESSAGES display control (ON/OFF).

7.4.12.35 ERROR DETAIL

(CVAL('ERROR DETAIL'))

When an -ERROR- occurs now you can track additional information including the name of the file being run and the byte offset within the file. If the thing being run is a procedure it saves information on the select used to fetch the procedure too. We have implemented a new method of simple "stack" to keep the last three errors. To see the information tracked use this new CVAL option.

```
SET VAR vError = (CVAL('ERROR DETAIL'))
```

Each time you call this particular CVAL function the stack pointer decrements so successive calls allow you to see all three errors. Call it a fourth time and you will see the first one again, etc.

7.4.12.36 ERROR VARIABLE

(CVAL('ERROR VARIABLE'))

Returns the value of the current ERROR VARIABLE.

7.4.12.37 ESCAPE

(CVAL('ESCAPE'))

Returns the status of the ESCAPE command parameter (ON/OFF).

7.4.12.38 EXPLODE

(CVAL('EXPLODE'))

Returns the status of the EXPLODE command parameter (ON/OFF).

Used with R:BASE for DOS only. When EXPLODE is set on, dialog boxes are displayed in full size instantly. When EXPLODE is set off, dialog boxes are displayed in an expanding fashion from the center.

7.4.12.39 FASTFK

(CVAL('FASTFK'))

Returns the status of the FASTFK command parameter (ON/OFF).

7.4.12.40 FASTLOCK

(CVAL('FASTLOCK'))

Returns the status of the FASTLOCK command parameter (ON/OFF).

7.4.12.41 FEEDBACK

(CVAL('FEEDBACK'))

Returns the status of the FEEDBACK command parameter (ON/OFF).

7.4.12.42 FILES

(CVAL('FILES'))

Returns the value of the FILES operating condition.

7.4.12.43 FIXED

(CVAL('FIXED'))

Returns the status of the FIXED command parameter (ON/OFF).

7.4.12.44 HEADINGS

(CVAL('HEADINGS'))

Returns the status of the HEADINGS display control (ON/OFF).

7.4.12.45 IDQUOTES

(CVAL('IDQUOTES'))

Returns the value of the IDQUOTES character setting.

7.4.12.46 INSERT

(CVAL('INSERT'))

Returns the status of INSERT display control (ON/OFF).

7.4.12.47 INTENSITY

(CVAL('INTENSITY'))

Returns the status of INTENSITY display control (ON/OFF).

Used with R:BASE 6.5++ for Windows only.

7.4.12.48 INTERVAL

(CVAL('INTERVAL'))

Returns value of the INTERVAL command parameter (ON/OFF).

7.4.12.49 LAST BLOCK TABLE

(CVAL('LastBlockTable'))

Tells you which table holds the last block in File 2.

If the last block table contains significant dead space, packing it should free up space at the end of File 2.

7.4.12.50 LAST ERROR

(CVAL('LAST ERROR'))

This is an alternative to the current ERROR VARIABLE system of -ERROR- trapping. Something more along the lines of an error variable which must be EXPLICITLY CLEARED.

Why? The current system does not allow error trapping in nested code segments or large blocks of code. Furthermore, since the -ERROR variable is automatically cleared after each command, it requires error trapping logic immediately after each command.

7.4.12.51 LAYOUT

(CVAL('LAYOUT'))

Returns the status of the LAYOUT display control (ON/OFF).

7.4.12.52 LINEEND

(CVAL('LINEEND'))

Returns the value of the LINEEND character setting.

7.4.12.53 LINES

(CVAL('LINES'))

Returns value of the LINES display control.

7.4.12.54 LOOKUP

(CVAL('LOOKUP'))

Returns the value of the LOOKUP command parameter (ON/OFF).

7.4.12.55 MANOPT

(CVAL('MANOPT'))

Returns value of the MANOPT command parameter (ON/OFF).

7.4.12.56 MANY

(CVAL('MANY'))

Returns the value of the MANY character setting.

7.4.12.57 MAXTRANS

(CVAL('MAXTRANS'))

Returns value of the MAXTRANS operating condition.

7.4.12.58 MDI

(CVAL('MDI'))

Returns the value of the MDI operating condition (ON/OFF).

7.4.12.59 MESSAGES

(CVAL('MESSAGES'))

Returns the status of the MESSAGES display control (ON/OFF).

7.4.12.60 MIRROR

(CVAL('MIRROR'))

Returns the value of the MIRROR operating condition.

7.4.12.61 MULTI

(CVAL('MULTI'))

Returns the value of the MULTI operating condition (ON/OFF).

7.4.12.62 NAME

(CVAL('NAME'))

Returns the currently logged-in R:BASE user NAME.

7.4.12.63 NETUSER

(CVAL('NETUSER'))

Returns the currently logged-in network user name.

7.4.12.64 NOTE_PAD

(CVAL('NOTE_PAD'))

Returns value of the NOTE_PAD operating condition.

7.4.12.65 NULL

(CVAL('NULL'))

Returns the value of the NULL character setting.

7.4.12.66 OFFMESS

(CVAL('OFFMESS'))

Return a text string, separated by a comma, with a list of all turned off -ERROR- message numbers in a current R:BASE session.

7.4.12.67 OLDLINE

(CVAL('OLDLINE'))

Returns the value of the OLDLINE operating condition (ON/OFF).

Used with R:BASE 6.5++ for Windows only. Allows the ability to have presentation objects, such as lines, cross multiple sections in Reports.

7.4.12.68 ONELINE

(CVAL('ONELINE'))

Returns the value of the ONELINE operating condition (ON/OFF).

Used with R:BASE for DOS only. When set to ON NOTE and TEXT fields will never wrap to the next line in Reports and SELECTS. Instead they will be truncated at the end of the column.

7.4.12.69 PAGEMODE

(CVAL('PAGEMODE'))

Returns the value of the PAGEMODE operating condition (ON/OFF).

7.4.12.70 PASSTHROUGH

(CVAL('PASSTHROUGH'))

Returns the value of the PASSTHROUGH operating condition (ON/OFF).

7.4.12.71 PLATFORM

(CVAL('PLATFORM'))

Returns 16 or 32 and WIN or DOS based on what R:BASE can determine about the Operating System. This may be complicated by using the Windows options which hide Windows from DOS programs.

7.4.12.72 PLUS

(CVAL('PLUS'))

Returns the value of the PLUS character setting.

7.4.12.73 POSFIXED

(CVAL('POSFIXED'))

Returns the value of the POSFIXED operating condition.

7.4.12.74 PORTS

(CVAL('PORTS'))

Returns the list of all available ports, separated by comma, on that workstation.

Example:

```
SET VAR vAvailablePorts = (CVAL('PORTS'))
SHOW VARIABLE vAvailablePorts
```

Will return the text string with a list of all ports that are available on that workstation. Each item in the list will be separated by comma (or the database character settings for comma).

7.4.12.75 PRINTERS

(CVAL('PRINTERS'))

Returns the list of all installed printers.

7.4.12.76 PRN_STATUS

(CVAL('PRN_STATUS'))

Captures the printer status.

7.4.12.77 PRN_ORIENTATION

(CVAL('PRN_ORIENTATION'))

Captures the printer orientation.

7.4.12.78 PRN_SIZE

(CVAL('PRN_SIZE'))

Captures the printer paper size.

7.4.12.79 PRN_SOURCE

(CVAL('PRN_SOURCE'))

Captures the printer paper source.

7.4.12.80 PRN_QUALITY

(CVAL('PRN_QUALITY'))

Captures the printer print quality (DPI).

7.4.12.81 PRN_COPIES

(CVAL('PRN_COPIES'))

Captures the printer copy count.

7.4.12.82 PRN_COLORMODE

(CVAL('PRN_COLORMODE'))

Captures the printer color mode.

7.4.12.83 PRN_DUPLEXMODE

(CVAL('PRN_DUPLEXMODE'))

Captures the printer duplex mode.

7.4.12.84 PRN_COLLATION

(CVAL('PRN_COLLATION'))

Captures the printer collation.

7.4.12.85 QUALCOLS

(CVAL('QUALCOLS'))

Returns the value of the QUALCOLS operating condition.

7.4.12.86 QUALKEYS

(CVAL('QUALKEYS'))

Returns the columns assigned as a QualKeys for the current database.

7.4.12.87 QUALKEY TABLES

(CVAL('QUALKEY TABLES'))

Returns the tables assigned with QualKey columns for the current database.

7.4.12.88 QUOTES

(CVAL('QUOTES'))

Returns the value of the QUOTES character setting.

7.4.12.89 REFRESH

(CVAL('REFRESH'))

Returns the value of the REFRESH operating condition.

7.4.12.90 REVERSE

(CVAL('REVERSE'))

Returns the value of the REVERSE operating condition.

7.4.12.91 ROWLOCKS

(CVAL('ROWLOCKS'))

Returns the value of the ROWLOCKS operating condition.

7.4.12.92 RULES

(CVAL('RULES'))

Returns the value of the RULES operating condition.

7.4.12.93 SCRATCH

(CVAL('SCRATCH'))

Returns the value of the SCRATCH operating condition.

7.4.12.94 SCREENSIZE

(CVAL('SCREENSIZE'))

Returns the Screen Area in Pixels

7.4.12.95 SELMARGIN

(CVAL('SELMARGIN'))

Returns the value of the SELMARGIN operating condition.

7.4.12.96 SEMI

(CVAL('SEMI'))

Returns the value of the SEMI operating condition.

7.4.12.97 SERVER

(CVAL('SERVER'))

Returns the value of the SERVER operating condition.

7.4.12.98 SINGLE

(CVAL('SINGLE'))

Returns the value of the SINGLE operating condition.

7.4.12.99 SORT

(CVAL('SORT'))

Returns the value of the SORT operating condition.

Copyright © 1982-2025 R:BASE Technologies, Inc.

7.4.12.100 SORTMENU

(CVAL('SORTMENU'))

Returns the value of the SORTMENU operating condition.

7.4.12.101 STATICDB

(CVAL('STATICDB'))

Returns the value of the STATICDB operating condition.

7.4.12.102 TIME

(CVAL('TIME'))

Returns the TIME format of the currently connected database.

7.4.12.103 TIME FORMAT

(CVAL('TIME FORMAT'))

Returns the default TIME FORMAT of the currently connected database.

7.4.12.104 TIME SEQUENCE

(CVAL('TIME SEQUENCE'))

Returns the default TIME SEQUENCE of the currently connected database.

7.4.12.105 TIMEOUT

(CVAL('TIMEOUT'))

Returns the value of the TIMEOUT operating condition.

7.4.12.106 TOLERANCE

(CVAL('TOLERANCE'))

Returns the value of the TOLERANCE operating condition.

7.4.12.107 TRACE

(CVAL('TRACE'))

Returns the status of the TRACE command parameter (ON/OFF).

7.4.12.108 TRANSACT

(CVAL('TRANSACT'))

Returns the value of the TRANSACT operating condition.

7.4.12.109 USER

(CVAL('USER'))

Returns the R:BASE USER identifier.

7.4.12.110 USERAPP

(CVAL('USERAPP'))

Returns the file extensions to be displayed in the Object Manager.

Used with R:BASE 6.5++ for Windows only. You can specify which files display in the Object Manager after you click the Apps tab. Your choices are saved in the RBASE.CFG file. You can specify a maximum of three extensions.

7.4.12.111 USERDOMAIN

(CVAL('USERDOMAIN'))

Returns the Name of Logged-In Network Domain.

7.4.12.112 USERID

(CVAL('USERID'))

Returns the Windows User ID.

7.4.12.113 VERIFY

(CVAL('VERIFY'))

Returns the value of the VERIFY operating condition.

7.4.12.114 VERSION

(CVAL('VERSION'))

The VERSION parameter of the SHOW command can be used to determine the exact version as well as the build number of the R:BASE Engine. This can also be used with CVAL to store the version as well as build number information in a variable.

See also:
(CVAL('BUILD')) to determine the exact build number of the R:BASE Front-End GUI
(CVAL('VERSION BUILD')) to determine only the build number of the R:BASE Engine

7.4.12.115 VERSION BITS

(CVAL('VERSION BITS')

Returns 16 or 32. At this time there are no versions of R:BASE that we can recommend running on 16 bit systems and as such this will always return 32.

7.4.12.116 VERSION BUILD

(CVAL('VERSION BUILD'))

Returns only the current build number of the R:BASE Engine. This can be useful in determining which features are available to you. This can also be used with CVAL to store the build number build number of the R:BASE Engine in a variable.

See also:
(CVAL('BUILD')) to determine the exact build number of the R:BASE Front-End GUI
(CVAL('VERSION')) to determine the version as well as build number of R:BASE Engine

7.4.12.117 VERSION SYSTEM

(CVAL('VERSION SYSTEM'))

Returns the value WIN or DOS.

7.4.12.118 WAIT

(CVAL('WAIT'))

Returns the value of the WAIT operating condition.

7.4.12.119 WALKMENU

(CVAL('WALKMENU'))

Returns the value of the WALKMENU operating condition.

7.4.12.120 WHILEOPT

(CVAL('WHILEOPT'))

Returns the value of the WHILEOPT operating condition (ON/OFF).

7.4.12.121 WIDTH

(CVAL('WIDTH'))

Returns the value of the WIDTH operating condition.

7.4.12.122 WINBEEP

(CVAL('WINBEEP'))

Returns the value of the WINBEEP operating condition.

7.4.12.123 WINDOWSPRINTER

(CVAL('WINDOWSPRINTER'))

This parameter returns the windows default printer.

7.4.12.124 WRAP

(CVAL('WRAP'))

Returns the value of the WRAP operating condition (ON/OFF).

7.4.12.125 WRITECHK

(CVAL('WRITECHK'))

Returns the value of the WRITECHK operating condition (ON/OFF).

7.4.12.126 ZERO

(CVAL('ZERO'))

Returns the value of the ZERO operating condition (ON/OFF).

7.4.12.127 ZOOMEDIT

(CVAL('ZOOMEDIT'))

Returns the value of the ZOOMEDIT operating condition (ON/OFF).

7.4.13 CVTYPE

(CVTYPE('*colvarname*',*flag*))

Returns the data type for a given column or variable name.

To return the data type for a given column, a zero "0" flag must be used. To return the data type for a given variable, the one "1" flag must be used.

After connecting to ConComp or RRBYW14, try the following two examples at the R> Prompt:

Example 01.

```
SET VAR vCustIDType TEXT = (CVTYPE('CustID',0))
SET VAR vEmpCity TEXT = (CVTYPE('EmpCity',0))

SHOW VARIABLES

vCustIDType = INTEGER TEXT
vEmpCity = TEXT,16 TEXT
```

Example 02.

```
SET VAR vCustIDType TEXT = (CVTYPE('CustID',0))
SET VAR vVarInquiry TEXT = (CVTYPE('vCustIDType',1))

SHOW VARIABLES

vCustIDType = INTEGER TEXT
vVarInquiry = TEXT,7 TEXT
```

NOTES:

- When using the zero flag to return column data types, you must be connected to a database.
- When returning a TEXT data type, the length is included and is separated with a comma. When returning a NUMERIC data type, the precision and scale are separated with commas.

7.5 D

7.5.1 DATETIME

(DATETIME(*date,time*))

Concatenates date and time variables or constants.

The following expression concatenates the #DATE and #TIME values into a variable (*vdatetime*) that has a DATETIME data type.

```
SET VAR vdatetime=(DATETIME(.#DATE,.#TIME))
```

7.5.2 DECRYPT

(DECRYPT(*'string','password'*))

Returns the original string that was encrypted using the ENCRYPT function. A password must be specified to return the original string.

```
SET VAR vDecryptNumber TEXT =
(DECRYPT('1CCF2D7B795D4317475B5144154A','laser'))
SHOW VAR vDecryptNumber
7247330053
```

If the password specified with DECRYPT is not correct, the result will be a NULL valued string.

7.5.3 DELFUNC

(DELFUNC('function_name'))

Deletes a declared DLL function. If the DLL function is successfully deleted, a 1 is returned. If the DLL function is not deleted, a 0 is returned.

Example:

```
SET VAR v1 = (DELFUNC('FunctionName'))
```

7.5.4 DEXTRACT

(DEXTRACT(*datetime*))

Returns the date portion of a value that has a DATETIME data type.

In the following example, the value of *vdextract* is *06/12/93*.

```
SET VAR vdextract = (DEXTRACT('06/12/93 12:15:30.123'))
```

7.5.5 DIM

(DIM(*arg1*,*arg2*))

Returns the positive difference between *arg1* and *arg2*. *Arg1* must be a value with a DOUBLE, REAL, NUMERIC, or INTEGER data type. *Arg2* must be any value with a DOUBLE, REAL, NUMERIC, or INTEGER data type other than 0. The result is always positive or zero. If the result is less than or equal to zero, DIM returns *0*.

In the following example, the value of *vdim1* is *0*; *vdim2* is *2*; *vdim3* is *2*; *vdim4* is *0*.

```
SET VAR vdim1 = (DIM(2,4))
SET VAR vdim2 = (DIM(4,2))
SET VAR vdim3 = (DIM(-2,-4))
SET VAR vdim4 = (DIM(-4,-2))
```

7.5.6 DLCALL

The DLCALL Function calls any Windows dynamic link library (DLL) and loads it into memory for use with R:BASE.

Syntax for External DLL:

(DLCALL('libraryname.ext', 'FunctionOrProcedureName', [arg],[arg],[.....]))

Syntax for Windows API:

(DLCALL('libraryname', 'FunctionOrProcedureName', [arg],[arg],[.....]))

External DLLs must have the file name listed with the extension, such as 'MyLibrary.dll'.

Windows APIs require only the name of the Library, without the extension, such as: 'Kernel32' or 'User32'.

The DLLs must be created as Standard Windows 32-bit DLLs. Any number of functions or procedures can be used in a single DLL. Functions or Procedures to be used with DLCALL must be Exported in the DLL. No special code is necessary in the DLL for it to be used by R:BASE.

DLL Location:

DLLs can be located in the Legal Windows Search Path, and if elsewhere, then specify the FullPathName in DLLoad.

Search Path Used by Windows to Locate a DLL

With both implicit and explicit linking, Windows first searches for "known DLLs", such as Kernel32.dll and User32.dll. Windows then searches for the DLLs in the following sequence:

1. The directory where the executable module for the current process is located.
2. The current directory.
3. The Windows system directory. The GetSystemDirectory function retrieves the path of this directory.
4. The Windows directory. The GetWindowsDirectory function retrieves the path of this directory.
5. The directories listed in the PATH environment variable.

When or If DLLOAD is Used:

DLLOAD may be called "anytime" during the Session to Load the Library into Memory OR as a subsequent call to determine if the Library is Loaded.

In any event, DLLOAD is called internally when the Library is first referenced by DLCALL, and THEN the Library remains in memory until either the R:BASE session is ended OR DLFREE is called.

Data Type Rules:

Data types in the functions must be of the same storage size as the corresponding R:BASE data types.

Example: 32-bit Win32 Integer = 4 bytes of storage vs 32-bit R:BASE Integer = 4 bytes of storage

Declaration Logic:

Calls to a function OR procedure from ANY DLL must be declared at Least ONCE in the session in which they will be referenced.

Two Calling Conventions are supported, using the STDCALL and CDECL Keyword in the Declaration.

STDCALL

Function Declaration using STDCALL:

```
STDCALL FUNCTION 'functionName' (PTR VARCHAR (SIZE), ....... ) : VARCHAR (SIZE)
STDCALL FUNCTION 'functionName' ALIAS 'functionAliasName' (PTR TEXT
(SIZE), ....... ) : TEXT (SIZE)
```

Procedure Declaration using STDCALL:

```
STDCALL VOID FunctionOrProcedureThatHasNoReturnValue (PTR TEXT (SIZE))
```

Windows API Declaration using STDCALL:

```
STDCALL FUNCTION 'GetCurrentDirectoryA' ALIAS 'GetCurrentDir' (PTR TEXT, INTEGER )
: INTEGER
```

CDECL

Function Declaration using CDECL:

```
CDECL FUNCTION 'functionName' (INTEGER) : INTEGER
CDECL FUNCTION 'functionName' ALIAS 'functionAliasName' (PTR DOUBLE) : DOUBLE
```

Procedure Declaration using CDECL

```
CDECL VOID FunctionOrProcedureThatHasNoReturnValue (PTR TEXT (SIZE))
```

Parameters

'SIZE' applies to TEXT and VARCHAR data types.

Important Notes:

A. Parameters in the Declaration are in the REVERSE order from the Actual Function or Procedure in the DLL.
Parameters can be 0 to n Parameters of any legal R:BASE Datatype.

B. Function Names ARE CASE SENSITIVE in the DECLARATION ONLY.
The Case must match the casing used in the DLL.
Case is INSENSITIVE when used in DLCALL.

Remarks:

- For best results, TEXT and VARCHAR data should be passed with the PTR (Pointer) Attribute and ANY VARCHAR data type Larger than 32K as a Parameter, MUST be passed as PTR.

- VARCHAR data type as a Return Value restricted to 32K, but Any SIZE up to 256MB can be passed as a Pointer to the R:BASE Variable. Modification of the Data Passed as Pointer must be on the data pointed to.

- It is important that the SIZE parameter on TEXT and VARCHAR be Specified to avoid creating excess buffer space that has to be created from the Declaration.

For Example:

If the Function is Declared like this:
```
STDCALL function 'somefunction' ( ptr varchar (nn)) : integer
```
Then *nn* <= 256Mb.

If the Function is Declared like this:
```
STDCALL function 'somefunction' ( ptr varchar ) : integer
```
Then because SIZE is omitted, the buffer for VARCHAR will have default SIZE = 256MB.
*** AVOID THIS UNLESS THAT IS THE ACTUAL SIZE OF THE DATA TO BE PASSED!**

If the Function is Declared like this:
```
STDCALL function 'somefunction' ( integer ) : varchar(nn)
```
Then because SIZE is Specified, nn bytes <= 32K will be returned.

If the Function is Declared like this:
```
STDCALL function 'somefunction' ( integer ) : varchar
```
Then because SIZE is omitted, the buffer for VARCHAR will have default SIZE = 32K.

When SIZE is Specified, the data passed as parameter or as return value, if Greater than the SIZE, will be truncated to SIZE.

Example of a DLL created in Delphi exporting three functions:

```
// Begin Dll Code
library DemoLib;

uses
  SysUtils, Classes;

{$R *.res}

function MultInt (NumIN : Integer) : Integer; stdcall;
begin
  Result := (NumIN * 2);
end;

function MultDbl (NumDbl : Double) : Double; stdcall;
begin
  Result := (NumDbl * 2);
end;

procedure LCaseByREF(DataIN : PChar); stdcall;
begin
  ansiStrLower(DataIN);
end;

function LCaseByVAL (DataIN : PChar) : PChar; Stdcall;
begin
  Result := ansiStrLower(DataIN);
end;

Exports MultInt, LCaseByREF, LCaseByVAL;

begin

end.
// End Dll Code
```

Example Usage From Within R:BASE:

```
-- BEGIN Demo.rmd
-- Declare the functions to be used from the DLL
STDCALL function 'MultInt' ( Integer ) : Integer
STDCALL VOID 'LCaseByREF' (ptr TEXT (30))
STDCALL function 'LCaseByVAL (ptr TEXT (60)) : TEXT (60)

--Set somme variables for use
Set VAR vTEXT TEXT = 'RBASE TECHNOLOGIES'
SET VAR vINT INTEGER = 128
SET VAR v1 INTEGER = 0

-- OPTIONALLY CALL DLLOAD
```

```
SET VAR v1 = (DLLOAD('DemoLib.dll'))
IF v1 = 0 THEN
  PAUSE 2 USING 'DemoLib.dll NOT LOADED.. EXITING'
  RETURN
ENDIF

SET VAR V1 = (dlcall('demolib.dll', 'changecase', vtext))

{ The Value for v1 will be null because ChangeCase is a procedure and
doesn't
  RETURN A RESULT, but the value of vTEXT which is passed as a POINTER has
been
  changed to 'rbase technologies'}

SET VAR v1 = (DLCALL('demolib.dll', 'MultInt', vINT))

--The value for v1 will be 256 the value returned from the function.

-- running the following against RRBYW14
SELECT (DLCALL('demolib.dll','lcasebyval', Company))=60 FROM +
Customer WHERE LIMIT = 2

{Yields the following output:
 (DLCALL('demolib.dll','lcasebyval', Company)
 ------------------------------------------------------------
 computer warehouse - ii
 microtech university - i
}

SELECT ((ICAP2((DLCALL('demolib.dll','lcasebyval', Company))))) = 60 +
FROM Customer WHERE LIMIT = 2

{Yields the following otput:
 ((ICAP (DLCALL('demolib.dll','lcasebyval',
 ------------------------------------------------------------
 Computer Warehouse - Ii
 Microtech University - I
}

-- Optionally CALL DLFREE
SET VAR v1 = (DLFREE('DemoLib.dll'))

-- END Demo.rmd
```

Example of a DLL created in C++ Exporting three functions:

```
// BEGIN C++ DLL
// loaddll.cpp : Defines the entry point for the DLL application.
//
#include <windows.h>
#include <stdio.h>
```

```
BOOL APIENTRY DllMain( HANDLE hModule,
                       DWORD  ul_reason_for_call,
                       LPVOID lpReserved
                                         )
{
    return TRUE;
}
#ifdef __cplusplus    // If used by C++ code,
extern "C" {          // we need to export the C interface
#endif

__declspec(dllexport) int cfunc1(int i){
        return i;
}
__declspec(dllexport) double cfunc2(double *inp){
        double rtn = *inp;
        rtn++;
        return rtn;
}
__declspec(dllexport) char * cfunc3(char *inp){
        strcat(inp," + ");
        return inp;
}

#ifdef __cplusplus
}
#endif

// END C++ DLL
```

See Also:

CHKFUNC
DELFUNC
DLLOAD
DLFREE

Special thanks to:
Mike Byerley (Fort Wayne, Indiana), an Authorized R:BASE Developer, for his contribution to the introduction, implementation and testing of the DLCALL Function in R:BASE.

7.5.7 DLFREE

(DLFREE('libraryname.ext'))

Checks to see if a given library file can be freed.

The function returns an integer value of 1 if the library is freed and 0 if a given library is not freed.

Example:

SET VARIABLE vFreePlugin = (DLFREE('RCharts95.RBM'))

See Also:

CHKFUNC
DELFUNC
DLLCALL
DLLOAD

7.5.8 DLLOAD

(DLLOAD('libraryname.ext'))

Checks to see if a given library file is loaded.

The function returns an integer value of 1 if the library is loaded and 0 if a given library is not loaded.

Example:

```
SET VARIABLE vLoadPlugin = (DLLOAD('RCharts95.RBM'))
```

See Also:

CHKFUNC
DELFUNC
DLLCALL
DLFREE

7.5.9 DNW

(DNW(*date*))

Returns the date value for the next working (business) day.

This fuction recognizes Monday through Friday as business days and does not recognize Saturday or Sunday. Holidays are not considered.

7.5.10 DWE

(DWE(*date*))

Returns the date value for the next weekend day.

This fuction recognizes Saturday or Sunday as weekend days and does not recognize Monday through Friday. Holidays are not considered.

7.5.11 DWRD

(DWRD(*value*))

Converts a currency value to its word representation. For example, (DWRD($100.50)) returns "one hundred dollars and fifty cents."

7.6 E

7.6.1 ENCRYPT

(ENCRYPT(*'string','password'*))

Returns an encrypted string of HEX characters, which is twice as long as the original string, plus 8 more characters. A password must be specified to use with the DECRYPT function in order to return the original string.

```
SET VAR vEncrptNumber TEXT = (ENCRYPT('7247330053','laser'))
SHOW VAR vEncrptNumber
1CCF2D7B795D4317475B5144154A
```

Once a value is encrypted, users cannot tell what the value is by selecting the raw encrypted value. If a form retrieves the encrypted value, it can then call the DECRYPT function the value to get the original value.

7.6.2 ENVVAL

(ENVVAL(*'environmentvar'*))

Returns the current value of the specified DOS environment variable. You must either enclose the name of the environment variable in quotation marks or use a text variable to which you have assigned the environment variable.

First, assume you have the command below in your AUTOEXEC.BAT file.

```
SET workstat=10
```

Now, in R:BASE, you can use ENVVAL to find the value of that environment variable, as shown in the example below. The value of *vworkstat* will be the text value *10*.

```
SET VAR vworkstat = (ENVVAL('workstat'))
```

7.6.3 EXP

(EXP(*arg*))

Raises *e* to the *arg* power (where *e* = 2.71828182845905 and *arg* is any value with a DOUBLE, REAL, NUMERIC, or INTEGER data type). By raising *e* to an exponent, EXP performs the inverse operation of LOG.

In the following example, the value of *vexp* is *2.71828182845905*.

```
SET VAR vexp = (EXP(1))
```

7.7 F

7.7.1 FILENAME

(FILENAME(0))

Generates a unique filename with a .$$$ extension, and creates the file in the current directory.

7.7.2 FINDFILE

(FINDFILE('*filename*'))

Returns the location of a file. The function looks first in the current directory and then searches the DOS path for the file. If the file is found, the full path name is returned, if it isn't found, a NULL is returned. Wildcards in the filename will produce unpredictable results.

7.7.3 FLOAT

(FLOAT(*arg*))

Converts a number with a TEXT, INTEGER, or CURRENCY data type to a value with a REAL or DOUBLE data type. This is not the same as the FLOAT data type.

In the following example, the value of *vfloat1* is *2.*, a number that has a REAL data type, and the value of *vfloat2* is also *2.*, a number that has a DOUBLE data type. If a variable is not assigned a data type, it becomes DOUBLE.

```
SET VAR vfloat1 REAL = (FLOAT(2))
SET VAR vfloat2 = (FLOAT(2))
```

7.7.4 FORMAT

(FORMAT (*value*,'*picture-format*'))

Prints picture formats to a variable, rather than only to the screen. You can use FORMAT anywhere that you can use a function. The result of the FORMAT function is always text.

In the syntax for this function, *value* is the value you want to be displayed in a particular format; it can be a column, variable, or a constant value. *'Picture-format'* is the picture format you establish.

The FORMAT function can be useful in several ways:

- Aligning decimals
- Punctuating long numbers
- Formatting currency
- Formatting text

The characters you can use to format your data are listed below.

For All Data

[<]	Data is left justified.
[>]	Data is right justified.
[^]	Data is centered.

For Numbers

[-]	Places a minus sign to the right of a negative number.
[DB]	Places DB to the right of a negative number.
[( )]	Encloses a negative number in parentheses.
[CR]	Places CR to the right of a positive number.
9	Fills unused space with blanks.
0	Fills unused space with zeros.
*	Fills unused space with asterisks.

For Text

_	Letters are uppercase; other characters are blank.
|	Letters are lowercase; other characters are blank.
%	Letters are uppercase; other characters are unchanged.
?	Letters are lowercase; other characters are unchanged.

7.7.4.1 Aligning Decimals

The following example shows how you can use the FORMAT function to align decimal points in a column:

```
SELECT (FORMAT(bonuspct,'99.000')) FROM salesbonus
```

The following example shows the effect of the FORMAT function on the above SELECT statement:

Using FORMAT	Without FORMAT
0.003	0.003
0.002	0.002
0.000	0
0.001	0.001

7.7.4.2 Formatting Currency

The following example shows how you can use the FORMAT function to only display whole dollars:

```
SELECT (FORMAT(netamount,'[>]$999,999')) FROM salesbonus
```

This SELECT statement displays data as right justified whole dollars, as shown below:

Using FORMAT	Without FORMAT
$176,000	$176,000.00
$87,500	$87,000.00

7.7.4.3 Formatting Text

You must include a format character for each text character. The following example shows how you can use the FORMAT function to display text in uppercase:

```
SELECT (FORMAT(empfname,'________')) FROM employee
```

Using FORMAT	Without FORMAT
JUNE	June
ERNEST	Ernest
PETER	Peter

7.7.4.4 Punctuating Long Numbers

The following example shows how you can use the FORMAT function to include a comma after the thousand's place:

```
SELECT (FORMAT(transid,'999,999')) FROM transmaster
```

The following shows the effect of the FORMAT function on the above SELECT statement:

Using FORMAT	Without FORMAT
104	104
2,002	2002
39,765	39765

7.7.5 FV1

(FV1(*pmt,int,per*))

Returns the future value of a series of equal payments of the amount, *pmt*, periodic interest rate, *int*, and compounding periods, *per*.

In the following example, FV1 returns the ending balance in your savings account if you deposit $300 each month for four years with an annual interest rate of 6.25%. The value of *vfv1* (the balance) is *$16,311.90*.

```
SET VAR vfv1 = (FV1(300,(.0625/12),(4 *12)))
```

7.7.6 FV2

(FV2(*pv*,*int*,*per*))

Returns the future value of an amount, *pv*, invested at a periodic interest rate, *int*, for compounding periods, *per*.

In the following example, FV2 returns your ending balance for a savings account where you have deposited $1,800 with an annual interest rate of 5.5% compounded monthly (simple interest divided by compounding periods) for seven years (12 periods times seven years). The value of *vfv2* (the balance) is *$2,642.98*.

```
SET VAR vfv2 = (FV2(1800,(.055/12),(7 *12)))
```

7.8 G

7.8.1 GETDATE

(GETDATE('Calendar Caption'))

Example:

In a command file, EEP or Custom Button

```
SET VAR vGetDate = (GETDATE('Select Date'))
```

Will bring up the Windows GUI Calendar with today's date circled in red. Either you could click on OK to accept the circled date or click on any other date on the calendar using the current month or scrolling months!

The GETDATE function will return a valid date in column or variable.

7.8.2 GETKEY

(GETKEY(0))

Gets the text value, in brackets, of the first key available in the type-ahead buffer. If a key is not available, GETKEY waits for the next keystroke. Since R:BASE normally checks the buffer for the [Ctrl] + [Break] keys, you must set ESCAPE to off for GETKEY to work properly. Use CHKKEY before GETKEY to determine if a key is available. GETKEY does nothing with the zero that you enter in parentheses; GETKEY returns a value without receiving one.

If you are executing a process that takes several minutes to complete, you can use the CHKKEY and GETKEY functions to tell R:BASE what to do next, even while the process is executing.

7.8.3 GETVAL

(GETVAL('arg1', 'arg2'))

Gets a value based on the argument data provided. The followng GETVAL arguments are available:

- CheckMessageStatus
- GetDriveReady
- GetIPAddress
- GetLock

- GetMACAddr
- GetVolumeID
- PlayAndExit
- PlayAndWait

7.8.3.1 CheckMessageStatus

(GETVAL('CheckMessageStatus','####'))

Example:

```
    SET VAR vStatus TEXT = NULL
    SET VAR vStatus = (GETVAL('CheckMessageStatus','2038'))
```

Variable vStatus will return the text value of the current message status (ON/OFF) for error message 2038.

7.8.3.2 GetDriveReady

(GETVAL('GetDriveReady','driveletter'))

Example:

```
    SET VAR vReady = (GETVAL('GetDriveReady','A'))
```

Result: False if not ready, True if ready.

7.8.3.3 GetIPAddress

(GETVAL('GetIPAddress','n'))

Where 'n' is the parameter to either retrieve the number of active network adapters or to retrieve the IP address of a given active network adapter. Use '0' to retrieve the number of active network adapters and 1-9 to retrieve the IP address(es) of active network adapter(s) of a given network workstation/server.

Example:

```
-- Start
-- GetIPAddress.RMD
CLS
CLEAR VARIABLE vActiveAdapters,vIPAddress1,vIPAddress2, +
vIPAddress3,vIPAddress4,vPauseMessage,vCaption
SET VAR vActiveAdapters TEXT = NULL
SET VAR vIPAddress1 TEXT = NULL
SET VAR vIPAddress2 TEXT = NULL
SET VAR vIPAddress3 TEXT = NULL
SET VAR vIPAddress4 TEXT = NULL
SET VAR vPauseMessage TEXT = NULL
SET VAR vCaption TEXT = 'Understanding New GETVAL Function'

-- To retrieve the number of active network adapters
SET VAR vActiveAdapters = (GETVAL('GetIPAddress','0'))

-- To retrieve the IP address of first active network adapter
```

```
SET VAR vIPAddress1 = (GETVAL('GetIPAddress','1'))

-- To retrieve the IP address of second active network adapter
SET VAR vIPAddress2 = (GETVAL('GetIPAddress','2'))

-- To retrieve the IP address of third active network adapter
SET VAR vIPAddress3 = (GETVAL('GetIPAddress','3'))

-- To retrieve the IP address of fourth active network adapter
SET VAR vIPAddress4 = (GETVAL('GetIPAddress','4'))

SET VAR vPauseMessage = +
('Number of Active Adapter(s):'+(CHAR(009))&.vActiveAdapters+ +
  (CHAR(009))+(CHAR(013))+ +
 'IP Address of Active Adapter 1:'+(CHAR(009))&.vIPAddress1+ +
  (CHAR(009))+(CHAR(013))+ +
 'IP Address of Active Adapter 2:'+(CHAR(009))&.vIPAddress2+ +
  (CHAR(009))+(CHAR(013))+ +
 'IP Address of Active Adapter 3:'+(CHAR(009))&.vIPAddress3+ +
  (CHAR(009))+(CHAR(013))+ +
 'IP Address of Active Adapter 4:'+(CHAR(009))&.vIPAddress4+ +
  (CHAR(009))+(CHAR(013)))

CLS
PAUSE 2 USING .vPauseMessage CAPTION .vCaption +
ICON APP +
Button 'Yes, this is the R:BASE you have always wanted!' +
OPTION BACK_COLOR WHITE +
|MESSAGE_COLOR WHITE +
|MESSAGE_FONT_COLOR GREEN +
|BUTTON_COLOR WHITE
CLS
CLEAR VARIABLE vActiveAdapters,vIPAddress1,vIPAddress2, +
vIPAddress3,vIPAddress4,vPauseMessage,vCaption
RETURN
-- end
```

7.8.3.4 GetLock

(GETVAL('GetLock','tableviewname'))

GetLock is the first required parameter and the table/view name is the name of the table or view. Use this function to programmatically find the LOCK status of a table or view. The returning value is ON or OFF, depending on whether a lock is in place upon the table or view.

Example:

```
SET VAR vCheclLock = (GETVAL('GetLock','Customer'))
```

vCheckLock will return the value of ON or OFF for the *Customer* table

7.8.3.5 GetMACAddr

Use (GETVAL('GetMACAddr','n')) to retrieve the number of active network adapter(s) on workstation/server as well as the physical MAC (Media Access Control) address(es) of active network adapter(s).

Media Access Control address is a physical hardware address that uniquely identifies each node of a network. In IEEE 802 networks, the Data Link Control (DLC) layer of the OSI Reference Model is divided into two sub-layers: the Logical Link Control (LLC) layer and the Media Access Control (MAC) layer. The MAC layer interfaces directly with the network media. Consequently, each different type of network media requires a different MAC layer.

```
(GETVAL('GetMACAddr','n'))
```

Where 'n' is the parameter to either retrieve the number of active network adapters or to retrieve the MAC address of given active network adapter. Use '0' to retrieve the number of active network adapters and 1-9 to retrieve the MAC address of active network adapter(s) on given network station/server.

Examples:

Example 01: To get the number of active network adapter(s)

```
SET VAR vActiveAdapters = (GETVAL('GetMACAddr','0'))
```

The variable vActiveAdapters will return the number of active network adapters on that work station/server.

Example 02: To retrieve the MAC address of first active adapter

```
SET VAR vMACAddress = (GETVAL('GetMACAddr','1'))
```

The variable vMACAddress will return the value of first active network adapter on that workstation.

Example 03: To retrieve the MAC address of second active adapter

```
SET VAR vMACAddress = (GETVAL('GetMACAddr','2'))
```

The variable vMACAddress will return the value of second active network adapter on that workstation.

Example 04: To retrieve the MAC address of third active adapter

```
SET VAR vMACAddress = (GETVAL('GetMACAddr','3'))
```

The variable vMACAddress will return the value of third active network adapter on that workstation.

Practically, this new (GETVAL('GetMACAddr','n')) function can be used to customize access to your R:BASE for Windows Applications. Consequently, you can use this function to customize the properties of any control and/or settings of form using the PROPERTY command based on unique MAC address of workstation.

Example 05: Disabling/Hiding Application Menus based on MAC Address

Assuming your application includes 6 main menu options, namely Customers, Contacts, Employees, Products, Sales, Quarterly Sales Reports, General Inquiry (Read Only) with Component IDs MM_Customers, MM_Contacts, MM_Employees, MM_Products, MM_Sales, MMQuarterlyReports, MM_GeneralInquiry accordingly.

In a secure work environment, suppose only one workstation or notebook can have access to all menus with MAC address 00-0D-43-2B-D6-B4 and everyone else can only access to General Inquiry (Read Only) menu.

If you would like to disable Customers, Contacts, Employees, Products, Sales and Quarterly Sales Report menus to all except the workstation or notebook with the MAC address 00-0D-43-2B-D6-B4, then here's an example to use as embedded Custom EEP in "On After Start EEP" option of form properties:

```
IF (GETVAL('GetMACAddr','1')) <> '00-0D-43-2B-D6-B4' THEN
```

```
    PROPERTY MM_Customers ENABLED 'FALSE'
    PROPERTY MM_Contacts ENABLED 'FALSE'
    PROPERTY MM_Employees ENABLED 'FALSE'
    PROPERTY MM_Products ENABLED 'FALSE'
    PROPERTY MM_Sales ENABLED 'FALSE'
    PROPERTY MM_QuarterlyReports ENABLED 'FALSE'
ENDIF
RETURN
```

If you would like to hide Customers, Contacts, Employees, Products, Sales and Quarterly Sales Report menus to all except the workstation or notebook with the MAC address 00-0D-43-2B-D6-B4, then here's an example to use as embedded Custom EEP in "On After Start EEP" option of form properties:

```
IF (GETVAL('GetMACAddr','1')) <> '00-0D-43-2B-D6-B4' THEN
    PROPERTY MM_Customers VISIBLE 'FALSE'
    PROPERTY MM_Contacts VISIBLE 'FALSE'
    PROPERTY MM_Employees VISIBLE 'FALSE'
    PROPERTY MM_Products VISIBLE 'FALSE'
    PROPERTY MM_Sales VISIBLE 'FALSE'
    PROPERTY MM_QuarterlyReports VISIBLE 'FALSE'
ENDIF
RETURN
```

7.8.3.6 GetVolumeID

(GETVAL('GetVolumeID','driveletter'))

Example:

```
SET VAR vVolumeID = (GETVAL('GetVolumeID','C'))
```

Will return the label name of drive C.

7.8.3.7 PlayAndExit

(GETVAL('PlayAndExit','filepath'))

Example:

```
SET VAR vPlay = (GETVAL('PlayAndExit','c:\windows\media\tada.wav'))
```

Will play sound/wave file and continue the next command in a command file or EEP.

7.8.3.8 PlayAndWait

(GETVAL('PlayAndWait','filepath'))

Example:

```
SET VAR vPlay = (GETVAL('PlayAndWait','c:\windows\media\tada.wav'))
```

Will play sound/wave file and waits for the sound to finish before continue to a next command. This could be a long time if the sound file is long. You would want this if, for instance, you're writing an answering machine aplication in R:BASE for Windows.

7.9 H

7.9.1 HTML

(HTML(*string*))

Converts a text value to valid HTML code.

7.10 I

7.10.1 ICAP

(ICAP(*arg*))

Converts *arg* to a string with an initial capital letter on only the first word.

In the following example, the value of *vicap* is *Mary is going to Murrysville*.

```
SET VAR vicap = (ICAP('mary is going to Murrysville'))
```

7.10.2 ICAP1

(ICAP1(*arg*))

Converts *arg* to a string with an initial upper case letter on the first word, and lower case on an initial capital letter on any following words. This is also knwon as Sentence Casing.

In the following example, the value of *vicap1* is *Mary went down the street.*

```
SET VAR vicap1 = (ICAP1('mary went down The Street.'))
```

7.10.3 ICAP2

(ICAP2(*arg*))

Converts *arg* to a string with an initial capital letter on each word. This is also known as Word Case.

In the following example, the value of *vicap2* is *John Smith*.

```
SET VAR vicap2 = (ICAP2('john smith'))
```

7.10.4 ICAP3

(ICAP3(*arg*))

Converts *arg* to a string with an initial upper case letter on the first word, and principal words. Non-capitalized letters/words include articles (a, an, the), conjunctions (e.g., and, but, or), and prepositions (e.g., on, in, with). This is also known as Title Casing.

In the following example, the value for vTitle is "Snow White and the Seven Dwarfs".

```
SET VAR vTitle TEXT = (ICAP3('SNOW WHITE AND THE SEVEN DWARFS'))
```

7.10.5 IFCASEEQ

(IFCASEEQ(*arg1,arg2,arg3,arg4*))

If the case and values for *arg1* and *arg2* are equal, IFCASEEQ returns the value of *arg3*. If the case and values for *arg1* and *arg2* are not equal, IFCASEEQ returns the value of *arg4*. The data types of *arg1* and *arg2* must match and the data types of *arg3* and *arg4* must match.

The function enables easier conformance for password verification, as a mixture of upper and lower case is a standard.

In the following example, the value for vCheckPW1 is NO.

```
SET VAR vCheckPW1 TEXT =
(IFCASEEQ('SuperStrong1147','SUPERSTRONG1147','Yes','No'))
```

In the following example, the value for vCheckPW2 is TRUE.

```
SET VAR vCheckPW2 TEXT = (IFCASEEQ('LastOne','LastOne','True','False'))
```

7.10.6 ICHAR

(ICHAR(*arg*))

Converts a single character, returning its corresponding ASCII integer value.

In the following example, the integer value of *vichar* is *65*.

```
SET VAR vichar = (ICHAR('A'))
```

7.10.7 IDAY

(IDAY(*arg*))

Where *arg* is a value that has either a DATE or DATETIME data type, IDAY returns the integer day of the month for a particular date.

In the following example, the value of *viday* is *12*.

```
SET VAR viday = (IDAY('06/12/93'))
```

7.10.8 IDIM

(IDIM(*arg*))

Where *arg* is a value that has a DATE data type, IDIM returns the number of days within that month.

In the following example, the value of *vidim* is *31*.

```
SET VAR vidim = (IDIM('03/15/2006'))
```

7.10.9 IDOY

(IDOY(*arg*))

Where *arg* is a value that has a DATE data type, IDOY returns the number day of the year.

In the following example, the value of *vidoy* is *74*.

```
SET VAR vidoy = (IDOY('03/15/2006'))
```

7.10.10 IDWK

(IDWK(*arg*))

Where *arg* is a value that has either a DATE or DATETIME data type, IDWK returns the day of the week where Monday is 1.

In the following example, the value of *vidwk* is *3*.

```
SET VAR vidwk = (IDWK('06/16/93'))
```

7.10.11 IFEQ

(IFEQ(*arg1*,*arg2*,*arg3*,*arg4*))

If *arg1* and *arg2* are equal, IFEQ returns the value of *arg3*. If *arg1* and *arg2* are not equal, IFEQ returns the value of *arg4*. The data types of *arg1* and *arg2* must match and the data types of *arg3* and *arg4* must match.

The following command changes the value of the *thiscol* column to *100* if the values of the variables *vaval* and *vbval* are equal. If *vaval* and *vbval* are not equal, the value of *thiscol* becomes *200* in rows where *thiscol* is greater than or equal to *100*.

```
UPDATE thistab SET thiscol=(IFEQ(.vaval, .vbval, 100, 200))+
WHERE thiscol >= 100
```

7.10.12 IFEXISTS

(IFEXISTS(*arg1*,*arg2*,*arg3*))

If *arg1* contains a value, IFEXISTS returns the value of *arg2*. If *arg1* is null, then the value of *arg3* is returned.

7.10.13 IFF

IFF((*condition*), *arg1*, *arg2*)

If the condition is met, then the value of arg1 is returned. If condition is not met, then the value of arg2 is returned.

The condition must follow the syntax specifications as set with IF...ENDIF command structure.

The condition must list a set of conditions that combine to form a statement that is either true or false. Conditions can be combined with the connecting operators AND, OR, AND NOT, and OR NOT. It is important to note that the condition needs to be a single item, which is why quotes are used in the examples below.

The data types of arg1 and arg2 must match.

Examples:

```
-- Example 01:
- -Comparison of Variables
SET VAR v1 INTEGER = 1
SET VAR v2 INTEGER = 2
SET VAR v3 = (IFF('.v1<.v2','First','Second'))
SHOW VAR v3
First

-- Example 02:
-- SELECT and Display Feedback on Data

CONNECT RRBYW19
R>SELECT NetAmount,(IFF('NetAmount>$100000','Good','Needs Improvement'))=30 AS Status
FROM SalesBonus

 NetAmount       Status
 --------------- ------------------------------
     $176,000.00 Good
      $87,500.00 Needs Improvement
      $22,500.00 Needs Improvement
      $40,500.00 Needs Improvement
      $76,800.00 Needs Improvement
      $36,625.00 Needs Improvement
      $56,250.00 Needs Improvement
     $152,250.00 Good
     $108,750.00 Good
      $78,750.00 Needs Improvement
      $27,000.00 Needs Improvement
       $9,500.00 Needs Improvement
     $210,625.00 Good
```

7.10.14 IFGE

(IFGE(*arg1,arg2,arg3,arg4*))

If *arg1* is greater than or equal to *arg2*, IFGE returns the value of *arg3*. If *arg1* is less than *arg2*, IFGE returns the value of *arg4*. The data types of *arg1* and *arg2* must match and the data types of *arg3* and *arg4* must match.

In the following example, the value of *vifge* is *4*, because the date 01/10/2013 is greater than the date 10/15/2012.

```
SET VAR vifge = (IFGE('01/10/2013','10/15/2012',4,5))
```

7.10.15 IFGT

(IFGT(*arg1,arg2,arg3,arg4*))

If *arg1* is greater than *arg2*, IFGT returns the value of *arg3*. If *arg1* is less than *arg2*, IFGT returns the value of *arg4*. The data types of *arg1* and *arg2* must match and the data types of *arg3* and *arg4* must match.

In the following example, the value of *vifgt* is *4*, because the date 1/1/96 is greater than the date 1/1/95.

```
SET VAR vifgt = (IFGT('1/1/96','1/1/95',4,5))
```

7.10.16 IFLE

(IFLE(*arg1,arg2,arg3,arg4*))

If *arg1* is less than or equal to *arg2*, IFLE returns the value of *arg3*. If *arg1* is greater than *arg2*, IFLE returns the value of *arg4*. The data types of *arg1* and *arg2* must match and the data types of *arg3* and *arg4* must match.

In the following example, the value of *vifle* is *100*, since *A* is less than *B*.

```
SET VAR A = 37
SET VAR B = 48
SET VAR vifle = (IFLE(.A,.B,100,100))
```

7.10.17 IFLT

(IFLT(*arg1,arg2,arg3,arg4*))

If *arg1* is less than *arg2*, IFLT returns the value of *arg3*. If *arg1* is greater than *arg2*, IFLT returns the value of *arg4*. The data types of *arg1* and *arg2* must match and the data types of *arg3* and *arg4* must match.

In the following example, the value of *viflt* is *4*, since *A* is less than *B*.

```
SET VAR A = 37
SET VAR B = 48
SET VAR viflt = (IFLT(.A,.B,4,5))
```

7.10.18 IFNE

(IFNE(*arg1,arg2,arg3,arg4*))

If *arg1* and *arg2* are not equal, IFNE returns the value of *arg3*. If *arg1* and *arg2* are equal, IFNE returns the value of *arg4*. The data types of *arg1* and *arg2* must match and the data types of *arg3* and *arg4* must match.

In the following example, the value of *vifne* is *7*, since v1 is not equal to v2.

```
SET VAR v1 = 25
SET VAR v2 = 30
SET VAR vifne = (IFNE(.v1,.v2,7,8))
```

7.10.19 IFNULL

(IFNULL(*arg1,arg2,arg3*))

If *arg1* is null, then the value of *arg2* is returned. If *arg1* is not null, then the value of *arg3* is returned.

7.10.20 IFRC

(IFRC(*arg*))

Where *arg* is a value that has either a TIME or DATETIME data type, IFRC returns the current thousandth of a second. For this function to work correctly, the TIME format must be set to include thousandths of a second.

In the following example, the value of *vifrc* is *123*.

```
SET TIME FOR HH:MM:SS.SSS
SET VAR vifrc = (IFRC('12:15:30.123'))
```

7.10.21 IFWINDOW

(IFWINDOW('*windowname*'))

Returns 1 if a form with the *windowname* is open, 0 if not. *Windowname* is the name given to the instance of an MDI form started with the "AS *alias*" option when using the ENTER, EDIT USING, or BROWSE USING commands.

7.10.22 IHASH

(IHASH(*arg,n*))

R:BASE includes a function that can be used to create an integer value from a text value. The function was designed to create effective integer keys from long text columns. The function, IHASH, converts the entire text value, or just a specified number of characters.

Using this method is more complex that just indexing the LASTNAME column. First you need to add a computed column to your table using the IHASH function on the LASTNAME column to convert its text to integer values. You can modify your table through the Data Designer or the ALTER TABLE command:

```
ALTER TABLE employee ADD Hash_Lname=(IHASH(lastname,0)) INTEGER
```

The IHASH function converts the entire name to integer when used with the parameter 0. A different parameter converts the specified number of characters from the name, starting at the first character. For example, the parameter 7 will convert the first 7 characters of the lastname to an integer value. Deciding on the number of characters to convert can be one of the hardest things about using this method. Consider the relationships expressed in the following chart:

	Convert FEW characters	Convert MORE characters
PROS	Less input required	Less duplicate values
CONS	Greater duplicate values	More input required

After adding the computed column to your table, you need to use some programming commands as shown below to query that column. Using the IHASH function directly in a WHERE clause won't use indexes. First set a variable equal to IHASH of the value you're searching for, then use the variable in the WHERE clause. When using an IHASH column for searching, you won't be able to do ad hoc queries from the R:BASE main menu .

```
SET VAR vname = (IHASH('Smith',0))
SELECT * FROM employee WHERE Hash_Lname = .vname
```

This method does provide greater flexibility in that you can have users enter anywhere from 1 character to the entire name based on the number of characters you specify in the IHASH function. For example, add a computed column to the table that will IHASH the first four characters of the name. Then, in your program, check the length that the user enters and if it's greater than four characters use an extra condition on your WHERE clause.

```
DIALOG 'Enter lastname (at least 4 characters):' vname vendkey 1
SET VAR vlen1=(SLEN(.vname)),vname1=(SGET(.VNAME,4,1)),+
  vhash=(IHASH(.vname1,4))
  IF vlen1 > 4 THEN
    CHOOSE vchoice FROM #VALUES FOR (firstname & lastname) +
      FROM employee WHERE Hash_Lname=.vhash AND +
      (SGET(lastname,.vlen1,1))=.vname
  ELSE
    CHOOSE vchoice FROM #VALUES FOR (firstname & lastname) +
      FROM employee WHERE Hash_Lname=.vhash
  ENDIF
```

A user can enter any number of letters for use with an IHASH computation, but must enter at least as many characters as specified in the IHASH column definition or enter the full name. If the entry less than the specified number of characters and less than the full length of the name, the correct data is not found. For example, with a column defined as (IHASH(lastname,7)), entering "WILL" will not find "WILLIAMS", it will only find "WILL".

The advantages of using a computed column with the IHASH function are that you can turn an inefficient TEXT index into an efficient INTEGER index and you can provide flexibility in searching. Users will have a larger selection of names to choose from and can select the appropriate person from the list. For example, entering WILLIAM will find WILLIAMSON, WILLIAM, and WILLIAMS if you use IHASH(lastname,7).

A disadvantage of this method is that you need to add columns to your database. An extra computed column can slow down data entry. If you are tight on disk space this may not be an option. To determine how much additional disk space you'll need for an IHASH column, take the number of rows in the table and multiply by 4. The answer is the number of bytes of disk space you'll need for the additional column.

7.10.23 IHR

(IHR(*arg*))

Returns the integer hour of time, where *arg* is a value that has a TIME or DATETIME data type.

In the following example, the value of *vihr* is *12*.

```
SET VAR vihr = (IHR(12:15:30))
```

7.10.24 IINFO

(IINFO(arg1,*arg2,arg3*))

The IINFO function is used to return information about tables, columns, or indexes by reading internal bitmask flags. The function requires the ID number for the table, column, or index. This ID number can be obtained from the system tables SYS_TABLES, SYS_COLUMNS, or SYS_INDEXES, respectively. IINFO returns 0 if FALSE, or the number in argument 3 if TRUE.

Syntax:

```
(IINFO(flagtype,id,bitmask))
```

Where:

flagtype specifies the type of information returned; table flags, column flags, column flags for server tables, index flags, row ID

id specifies the ID number from the system tables for the table ID, column ID, or index ID

bitmask species the flag in the system table

Remarks:

- The values for arg1, arg2, and arg3 must be non-null integers, even if a particular argument is not needed for that case.
- IINFO returns 0 if FALSE, or the bitmask number in parameter 3 if TRUE (flags 4-7).

Flags:

Info	Flag Type	ID	Bitmask	Description
Row ID	0	0	0	Returns the rowid of the current row. The rowid is the offset from the start of file 2 where that row is stored. The values for arg2 and arg3 are not used. Using a 0 for arg2 and arg3 is recommended.
Minimum Data Type Scale	1	integer	0	Returns the minimum scale for the data type. The value for arg2 must be an integer value. The value for arg3 is not used. Using a 0 for arg3 is recommended.
Maximum Data Type Scale	2	integer	0	Returns the maximum scale for the data type. The value for arg2 must be an integer value. The value for arg3 is not used. Using a 0 for arg3 is recommended.
Table Cascade Flag	3	table ID	0	Returns cascade flag for a table. The value for arg3 is not used. Using a 0 for arg3 is recommended.
Column Flags	4	column ID	1	Returns bitmask value if is an autonumber column
	4	column ID	2	Returns bitmask value if a comment exists for column
	4	column ID	4	Returns bitmask value if column has a default value
	4	column ID	8	Returns bitmask value if column is temporary
	4	column ID	16	Returns bitmask value if column has an index
	4	column ID	32	Returns bitmask value if contains a USER default
	4	column ID	64	Returns bitmask value if contains a Not NULL flag
	4	column ID	128	Returns bitmask value if is a primary key or unique key

Column Flags (Server Tables)	5	column ID	1	Returns bitmask value if column is an optimal row qualifier
	5	column ID	2	Returns bitmask value if server column is read only
	5	column ID	4	Returns bitmask value if server column is autonumbered
	5	column ID	8	Returns bitmask value if server column row version qualifier
Table Flags	6	table ID	1	Returns bitmask value if comment exists for table
	6	table ID	2	Returns bitmask value if table has a primary key
	6	table ID	4	Returns bitmask value if table has a foreign key
	6	table ID	8	Returns bitmask value if table has an autonumbered column
	6	table ID	16	Returns bitmask value if table has a default column
	6	table ID	32	Returns bitmask value if table is readonly (dBASE)
	6	table ID	64	Returns bitmask value if table is temporary
	6	table ID	128	Returns bitmask value if table has a referenced key
	6	table ID	256	Returns bitmask value if table has a Not NULL column
	6	table ID	512	Returns bitmask value if table has a unique key
	6	table ID	1024	Returns bitmask value if table has a column with a data type greater than 10
	6	table ID	2048	Returns bitmask value if table has a VARBIT/VARCHAR column
	6	table ID	4096	Returns bitmask value if a cascade flag updates and deletes through all primary keys and unique keys
	6	table ID	8192	Returns bitmask value if server table has column aliases
	6	table ID	16384	Returns bitmask value if there is at least one trigger defined for the table
	6	table ID	32768	Returns bitmask value if relation as system view which created during multiple inner joins
Index Flags	7	index ID	3	Returns bitmask value for the constraint type: 0 = index, 1 = foreign key, 2 = primary key, 3 = unique key
	7	index ID	4	Returns bitmask value if this is a dBase index
	7	index ID	8	Returns bitmask value if this is a unique index
	7	index ID	16	Returns bitmask value if index is temporary
	7	index ID	32	Returns bitmask value if this is a referenced key
	7	index ID	64	Returns bitmask value if this is a case sensitive index
	7	index ID	128	Returns bitmask value if the is a Foreign Index

Examples:

```
-- Example 01:
-- Using flag type 0 for the Titles table in the RRBYW18 database
SELECT EmpTID,EmpTitle,(IINFO(0,0,0)) FROM Titles

 EmpTID     EmpTitle                       (IINFO(0,0
 ---------- ------------------------------ ----------
          1 Office Manager                     524289
          2 Receptionist                       524341
          3 Sales Clerk                        524393
          4 Director Marketing                 524445
          5 Director Corporate Sales           524497
          6 Director Government Sales          524549
          7 Manager Support & Services         524601
          8 Outside Sales                      524653
```

```
-- Example 02:
-- Returns minimum and maximum data type scales using IINFO and (CVAL('ROWCOUNT'))
with the RRBYW18 database.
-- The values for currency will vary based upon the current CURRENCY DIGITS setting.
SELECT (CVAL('ROWCOUNT')) AS SYS_TYPE, SYS_TYPE_NAME=18, +
(IINFO(1, (INT(CVAL('ROWCOUNT'))), 0)) AS SYS_MIN, +
(IINFO(2, (INT(CVAL('ROWCOUNT'))), 0)) AS SYS_MAX +
FROM SYS_TYPES
```

Here is what it generates:

```
 SYS_TYPE        SYS_TYPE_NAME      SYS_MIN    SYS_MAX
 --------------- ------------------ ---------- ----------
 1               CURRENCY                    2          2
 2               VARBIT             -0-        -0-
 3               BITNOTE            -0-        -0-
 4               BIT                -0-        -0-
 5               VARCHAR            -0-        -0-
 6               BIGNUM             -0-        -0-
 7               BSTR               -0-        -0-
 8               GUID               -0-        -0-
 9               TEXT               -0-        -0-
 10              NUMERIC                     0         15
 11              INTEGER                     0          0
 12              REAL               -0-        -0-
 13              DOUBLE             -0-        -0-
 14              DATE               -0-        -0-
 15              TIME                        0          3
 16              DATETIME                    0          3
 17              NOTE               -0-        -0-
```

```
-- Example 03:
-- Displays tables with Cascade within the RRBYW18 database.
-- Note that the tables with 1 for the cascade value are tables
-- with primary keys that cascade to tables with foreign keys.
SELECT SYS_TABLE_NAME=20, +
SYS_TABLE_ID, +
(IINFO(3,SYS_TABLE_ID,0)) AS SYS_CASCADE +
FROM SYS_TABLES WHERE SYS_TABLE_TYPE = 'TABLE'
```

```
 SYS_TABLE_NAME       SYS_TABLE_ SYS_CASCAD
 -------------------- ---------- ----------
 Customer                     29          1
 CompUsed                     30          0
 SalesBonus                   31          0
 PaymentTerms                 32          0
 Contact                      33          0
 ProdLocation                 34          0
 Levels                       35          0
 Component                    36          0
 Product                      37          0
 SecurityTable                38          0
 InvoiceHeader                39         0
 PrintOptions                 40         0
 InvoiceDetail                41         0
 LicenseInformation           43         0
 Titles                       44         1
 Employee                     45         1
 StateAbr                     46         0
 FormTable                    47         0
 BonusRate                    48         0
 TestNote                     49         0
 RThemes_eXtreme              50         0
 ContactCallNotes             51         0
 tempemployee                 74         0


-- Example 04:
-- Checks that the CustState colun in the Customer table is indexed
SET VAR vCustStateHasIndex = (IINFO(4,191,16))
SHOW VAR vCustStateHasIndex
         16

-- Example 05:
-- Checks that the Employee table has a primary key
SET VAR vEmployeeHasPK = (IINFO(6,45,2))
SHOW VAR vEmployeeHasPK
          2

-- Example 06:
-- Checks that the Component table has a referenced key.
SET VAR vComponentRef = (IINFO(6,39,128))
SHOW VAR vComponentRef
        128

-- Example 07:
-- Returns the constraint type for the EmpID in the SalesBonus table
-- (0 = index, 1 = foreign key, 2 = primary key, 3 = unique key)
SET VAR vIsForeignKey = (IINFO(7,42,3))
SHOW VAR vIsForeignKey
          1
```

7.10.25 ILY

(ILY(*arg*))

Checks to see if the current year is a leap year.

The function returns a 1 if the year is a leap year, and 0 if it is not a leap year.

In the following example, the value of *vily* is *0*.

```
SET VAR vily = (ILY('03/15/2006'))
```

7.10.26 IMIN

(IMIN(*arg*))

Returns the integer minutes of time, where *arg* is a value that has a TIME or DATETIME data type.

In the following example, the value of *vmin* is *15*.

```
SET VAR vmin TO (IMIN(12:15:30))
```

7.10.27 IMON

(IMON(*arg*))

Returns the integer month for a particular date, where *arg* is a value that has a DATE or DATETIME data type.

In the following example, the value of *vimon* is *5*.

```
SET VAR vimon = (IMON('05/20/95'))
```

7.10.28 INT

(INT(*arg*))

Truncates a number that has a REAL, DOUBLE, or CURRENCY data type, returning a value that has an INTEGER data type.

In the following example, the value of *vint* is *1*.

```
SET VAR vint = (INT(1.6))
```

7.10.29 ISALPHA

(ISALPHA(*value*))

Checks the first character of a TEXT string. The function returns a 1 if true and a 0 if false. For example, (ISALPHA('abc')) returns 1 because the first character is a letter.

7.10.30 ISDIGIT

(ISDIGIT(*value*))

Checks the first character of a TEXT string. The function returns a 1 if true and a 0 if false. For example, (IFDIGIT('abc')) returns 0 because the first character is not a number.

7.10.31 ISEC

(ISEC(*arg*))

Returns the integer seconds of time, where *arg* is a value that has a TIME or DATETIME data type.

In the following example, the value of *visec* is *30*.

```
SET VAR visec = (ISEC(12:15:30))
```

7.10.32 ISLOWER

(ISLOWER(*value*))

Checks the first character of a TEXT string. The function returns a 1 if true and a 0 if false. For example, (ISLOWER('abc')) would return 1 because the first character is lower case.

7.10.33 ISSPACE

(ISSPACE(*value*))

Checks the first character of a TEXT string. The function returns a 1 if true and a 0 if false. The ISSPACE function evaluates as true when the first character is a space or one of the following: LF (char (10)), CR (char (13)), VT (char (11)), TAB (char (9)), FF (char (12)), EOL (char (254)).

7.10.34 ISTAT

(ISTAT('keyword'))

Returns the current value or setting of 'keyword'.

Using ISTAT parameters, you can determine database activity, and the current mouse and cursor column or row coordinates.

Use ISTAT to check the efficiency of settings to adjust locking scheme. You should be able to see differences in the results returned by the ISTAT keywords TOTALREADS, TOTALWRITES, and TOTALLOCKS depending on the locking scheme you have set.

The ISTAT function has a number of options that report on available memory in the dynamic data area R:BASE uses for processing.

You must either enclose the keyword in quotation marks or use a dot variable that has a TEXT data type to which you have assigned the SHOW keyword.

You can use majority of SHOW keywords with CVAL.

Following keywords can be used for (ISTAT('keyword')):

- CURRNUMALLOC
- CURSORCOL
- CURSORROW
- DBSIZE
- DISKSPACE
- FORM_CONTROL_TYPE
- FORM_DIRTY_FLAG
- ISRUNTIME
- LIMITNUMALLOC
- MAXFREE
- MAXNUMALLOC
- MEMORY
- MOUSECOL
- MOUSEROW
- PAGECOL
- PAGEROW
- RX1SIZE
- RX2SIZE
- RX3SIZE
- RX4SIZE
- TOTALALLOC
- TOTALFREE
- TOTALLOCKS
- TOTALREADS
- TOTALWRITES

7.10.34.1 CURRNUMALLOC

(ISTAT('CURRNUMALLOC'))

The CURRNUMALLOC parameter allows you to check the number of memory handles R:BASE for DOS has open. This returns 0 when used with R:BASE for Windows.

7.10.34.2 CURSORCOL

(ISTAT('CURSORCOL'))

(ISTAT('CursorCol')) will return an integer value indicating the current column that contains the cursor.

7.10.34.3 CURSORROW

(ISTAT('CURSORROW'))

(ISTAT('CursorRow')) will return an integer value indicating the current row that contains the cursor.

7.10.34.4 DBSIZE

(ISTAT('DBSIZE'))

DBSIZE returns the size of the currently open database (total of the four database files). If no database is connected, ISTAT returns 0.

7.10.34.5 DISKSPACE

(ISTAT('DISKSPACE'))

DISKSPACE returns the amount of free disk space on the current drive. If you wish to check for the free space on drives with over 2 gigs of free you must use syntax similar to this:

```
SET VAR vSpace DOUBLE = (ISTAT('DISKSPACE'))
```

This is necessary because the default datatype, INTEGER, cannot hold the required number of digits to report over 2 gigs of free space.

When depending on this routine it is best to test it on a platform as close to your target platform as possible.

7.10.34.6 FORM_CONTROL_TYPE

(ISTAT('FORM_CONTROL_TYPE'))

FORM_CONTROL_TYPE returns a value based on which object has focus when the entry/exit procedure containing the ISTAT function is run. Returns the following.

0	No current control
1	Edit Field
2	Combo Box
3	Check Box
5	Radio Button
7	Push Button

7.10.34.7 FORM_DIRTY_FLAG

(ISTAT('FORM_DIRTY_FLAG'))

FORM_DIRTY_FLAG Returns 1 if a change has been made to the data, 0 if no change has been made.

7.10.34.8 ISRUNTIME

(ISTAT('ISRUNTIME'))

Use the ISRUNTIME parameter to determine if the end user is accessing R:BASE via a full version or a Runtime version.

If the end user is using Runtime the value returned will be 1. If the end user is using a Full Version then the value returned will be 0.

7.10.34.9 LIMITNUMALLOC

(ISTAT('LIMITNUMALLOC'))

The LIMITNUMALLOC parameter allows you to check the maximum number of memory handles R:BASE for DOS can open. This returns 0 when used with R:BASE for Windows.

Unless you use the -h startup option this will be 300.

7.10.34.10 MAXFREE

(ISTAT('MAXFREE'))

The MAXFREE parameter allows you to determine the largest free memory block within the area allocated to R:BASE. The TOTALALLOC parameter can be used to determine the total amount of dynamic memory allocated to R:BASE. MAXFREE will always be smaller than TOTALALLOC.

This will not return valid information under R:BASE for Windows.

7.10.34.11 MAXNUMALLOC

(ISTAT('MAXNUMALLOC'))

The MAXNUMALLOC parameter allows you to check the highest number of memory handles R:BASE for DOS has held open at any one point during this session. This returns 0 when used with R:BASE for Windows.

7.10.34.12 MEMORY

(ISTAT('MEMORY'))

The MEMORY parameter allows you to retrieve the amount of memory, in bytes, remaining for R:BASE for DOS to use.

This will not return a valid amount when running R:BASE for Windows.

7.10.34.13 MOUSECOL

(ISTAT('MOUSECOL'))

MOUSECOL returns the column where the mouse pointer is.

This is only supported in DOS, and will not return a valid amount when running R:BASE for Windows.

7.10.34.14 MOUSEROW

(ISTAT('MOUSEROW'))

MOUSEROW returns the row where the mouse pointer is.

This is only supported in DOS, and will not return a valid amount when running R:BASE for Windows.

7.10.34.15 PAGECOL

(ISTAT('PAGECOL'))

PAGECOL returns the column location of the cursor on a virtual page when you are using the SET PAGEMODE command. PAGECOL can only be used with the SHOW VARIABLE command; it does not work with the WRITE command.

7.10.34.16 PAGEROW

(ISTAT('PAGEROW'))

PAGEROW returns the row location of the cursor on a virtual page when you are using the SET PAGEMODE command. PAGEROW can only be used with the SHOW VARIABLE command; it does not work with the WRITE command.

7.10.34.17 RX1SIZE

(ISTAT('RX1SIZE'))

The RX1SIZE parameter returns the size, in bytes, of the RX1 file of the currently connect database. You may need to use the following syntax for large databases:

```
SET VAR vSize DOUBLE = (ISTAT('RX1SIZE'))
```

This is simply because the default datatype, INTEGER, may not be able to hold values as large as you need.

For a combined size use the DBSIZE parameter.

7.10.34.18 RX2SIZE

(ISTAT('RX2SIZE'))

The RX2SIZE parameter returns the size, in bytes, of the RX2 file of the currently connect database. You may need to use the following syntax for large databases:

```
SET VAR vSize DOUBLE = (ISTAT('RX2SIZE'))
```

This is simply because the default datatype, INTEGER, may not be able to hold values as large as you need.

For a combined size use the DBSIZE parameter.

7.10.34.19 RX3SIZE

(ISTAT('RX3SIZE'))

The RX3SIZE parameter returns the size, in bytes, of the RX3 file of the currently connect database. You may need to use the following syntax for large databases:

```
SET VAR vSize DOUBLE = (ISTAT('RX3SIZE'))
```

This is simply because the default datatype, INTEGER, may not be able to hold values as large as you need.

For a combined size use the DBSIZE parameter.

7.10.34.20 RX4SIZE

(ISTAT('RX4SIZE'))

The RX4SIZE parameter returns the size, in bytes, of the RX4 file of the currently connect database. You may need to use the following syntax for large databases:

```
SET VAR vSize DOUBLE = (ISTAT('RX4SIZE'))
```

This is simply because the default datatype, INTEGER, may not be able to hold values as large as you need.

Copyright © 1982-2025 R:BASE Technologies, Inc.

For a combined size use the DBSIZE parameter.

7.10.34.21 TOTALALLOC

(ISTAT('TOTALALLOC'))

The TOTALALLOC parameter allows you to determine the amount of dynamic memory allocated to R:BASE. Unless you ZIP out to another program this number should not change during an R:BASE session.

This will not return valid information under R:BASE for Windows.

7.10.34.22 TOTALFREE

(ISTAT('TOTALFREE'))

The TOTALFREE parameter allows you to determine the total amount of free memory within the area allocated to R:BASE. The TOTALALLOC parameter can be used to determine the total amount of dynamic memory allocated to R:BASE. TOTALFREE will always be smaller than TOTALALLOC.

This will not return valid information under R:BASE for Windows.

7.10.34.23 TOTALLOCKS

(ISTAT('TOTALLOCKS'))

TOTALLOCKS is used to determine the total number of locks placed on any assortment of tables, rows, and indexes.

7.10.34.24 TOTALREADS

(ISTAT('TOTALREADS'))

TOTALREADS returns the total number of disk reads since R:BASE began or the number of reads since the statement was last executed.

7.10.34.25 TOTALWRITES

(ISTAT('TOTALWRITES'))

TOTALWRITES returns the total disk writes since opening or since the last time the command was executed.

7.10.35 ISTR

(ISTR('*string*',*position*))

Returns the corresponding integer value.

This function converts a single character, which you specify within a string by position, returning its corresponding ASCII Character Chart Decimal value.

In the following example, the INTEGER value of *vDecimalValue* is 65 for the capital letter A.

Example 01:

Start R:BASE for Windows

At the R> Prompt:

SET VARIABLE vDecimalValue = (ISTR('R:BASE Rocks!',4))

SHOW VARIABLE

vDecimalValue = 65 INTEGER

7.10.36 ISUPPER

(ISUPPER(*value*))

Checks the first character of a TEXT string. The function returns a 1 if true and a 0 if false. For example, (ISUPPER('abc')) returns 0 because the first character is not upper case.

7.10.37 ITEMCNT

(ITEMCNT(*'Text String'*))

Use to count the number of items in a text string separated by current comma delimiter.

In the following example, the value of *vItems* is *3*.

```
SET VAR vItems = (ITEMCNT('a,b,c'))
```

Here is an example of using this function in a command block to format a CHOOSE box:

```
SET VAR vModels TEXT = NULL
SET VAR vLines INTEGER = NULL
SET VAR vModel TEXT = NULL
SELECT (LISTOF(Model)) INTO vModels INDIC IModel FROM Product
SET VAR vLines = (ITEMCNT(.vModels))
IF vLines > 18 THEN
SET LINES = 18
ENDIF
CLS
CHOOSE vModel FROM #LIST .vModels AT 4 30 TITLE 'Choose Model' +
CAPTION 'Available Models' Lines .vLines FORMATTED
IF vModel IS NULL OR vModel = '[Esc]' THEN
GOTO Done
ENDIF

-- Do what you have to do here ...

LABEL Done
CLEAR ALL VAR
QUIT TO MainMenu.RMD
RETURN
```

7.10.38 IWOY

(IWOY(*arg*))

Where *arg* is a value that has a DATE data type, IWOY returns the number week of the year.

In the following example, the value of *viwoy* is *11*.

```
SET VAR viwoy = (IWOY('03/15/2006'))
```

7.10.39 IYR

(IYR(*arg*))

Where *arg* is a value that has a DATE or DATETIME data type, IYR returns a two or four-digit integer year of date, depending on how the DATE format is set.

In the following example, the value of *viyr* is *1995* if the year in the DATE format is set to YYYY. The value of *viyr* is *95* if the year in the DATE format is set to YY.

```
SET VAR viyr = (IYR('09/13/95'))
```

7.10.40 IYR4

(IYR4(date)) or (IYR4(datetime))

Where *arg* is a value that has a DATE or DATETIME data type, IYR4 always returns a four-digit integer year of date, This is different than the IYR function with depends on how the DATE format is set.

This function will allow users to capture four digit year, regardless of the DATE FORMAT or DATE SEQUENCE settings. Results will be based on the DATE YEAR and CENTURY settings. See DATE format.

Example 01 (Database with settings):

```
DATE FORMAT: MM/DD/YYYY
DATE SEQ: MMDDYY
DATE YEAR: 30
DATE CENT: 19
SET VAR v1 = (IYR4(.#DATE))
```

Variable v1 will return the four digit INTEGER value of 2000

Example 02:

```
SET VAR v2 DATETIME = ('06/26/2000 08:00')
SET VAR v3 = (IYR4(.v2))
```

Variable v3 will return the four digit INTEGER value of 2000

Example 03 (Database with settings):

```
DATE FORMAT: MM/DD/YY
DATE SEQ: MMDDYY
DATE YEAR: 30
DATE CENT: 19
SET VAR v1 = (IYR4(.#DATE))
```

Variable v1 will ALSO return the four digit INTEGER value of 2000

Example 04:

```
SET VAR v2 DATETIME = ('06/26/00 08:00')
SET VAR v3 = (IYR4(.v2))
```

Variable v3 will ALSO return the four digit INTEGER value of 2000

7.11 J

7.11.1 JDATE

(JDATE(*arg*))

Where *arg* is a value that has a DATE or DATETIME data type, JDATE returns the Julian date of the date in the form YYYYDDD. This is a change from versions prior to R:BASE 6.1a which returned a value in the format YYDDD. This two digit year format was incompatible with the year 2000 and may require altering database structure or program code.

In the following example, the value of *vjdate* is *1995163*. The year is *1995*, and *163* means that the date is the 163rd day of 1995.

```
SET VAR vjdate = (JDATE('06/12/95'))
```

7.12 L

7.12.1 LASTKEY

(LASTKEY(*arg*))

Where *arg* is *0* or *1*. When *arg* is *1*, LASTKEY returns the original mapping for a key that has been remapped (the key that was pressed). When *arg* is *0*, LASTKEY returns the current mapping (the key that was executed).

In the following example, DIALOG displays the message and the current date, which the user can edit. LASTKEY captures the last key pressed by the user, enabling the user to press [Esc] to avoid entering a date.

```
SET VARIABLE vdate TEXT
SET VARIABLE vdate = (CTXT(.#DATE))
DIALOG 'Enter the invoice date to print:' vdate vendkey 1
SET VARIABLE vlast = (LASTKEY(0))
IF vlast = '[Esc]' THEN
   GOTO skipprnt
ENDIF
```

7.12.2 LAVG

(LAVG(*list*))

Returns the average of the values in *list*. Values in a list can be a CURRENCY, DOUBLE, REAL, INTEGER, NUMERIC, DATE, or TIME data type. The function LAVG is not the same as the function AVG, which is used only with the SELECT command.

In the following example, the value of *vlavg* is *5*. The total of the list is 20, which is divided by 4, the number of values in the list.

```
SET VAR vlavg = (LAVG(2,4,6,8))
```

7.12.3 LJS

(LJS(*text*,*width*))

Left justifies *text* in *width* characters, returning a text string.

In the following example, the value of *vljs* is *ABCD*. The text string is left justified in the field. In this case, trailing blanks are removed.

```
SET VAR vljs = (LJS('ABCD',10))
```

The value of *vljs2* in the following example is *COLUMN ONE   COLUMN TWO*. You can use LJS to embed spaces and concatenate strings, for example, to create a row of column headings for a screen or variable form. When you concatenate strings before assigning the result to a variable, as in this example, trailing blanks are retained. These spaces are preserved even if the first value is null.

```
SET VAR vljs2 = (LJS('COLUMN ONE',20) + 'COLUMN TWO')
```

The following example returns *COLUMN TWO* if *vcolname* is null.

```
SET VAR vcol2 TEXT = 'COLUMN TWO'
SET VAR vljs3 TEXT = (LJS(.vcolname,20) + .vcol2))
```

7.12.4 LMAX

(LMAX(*list*))

Returns the maximum value in *list*. Values in a list can be a CURRENCY, DOUBLE, REAL, INTEGER, NUMERIC, DATE, or TIME data type. The function LMAX is not the same as the function MAX, which is used only with the SELECT command.

In the following example, the value of *vlmax* is *80*, the highest number in the list.

```
SET VAR vlmax = (LMAX(2,80,14,22))
```

7.12.5 LMIN

(LMIN(*list*))

Returns the minimum value in *list*. Values in a list can be a CURRENCY, DOUBLE, REAL, INTEGER, NUMERIC, DATE, or TIME data type. The function LMIN is not the same as the function MIN, which is used only with the SELECT command.

In the following example, the value of *vlmin* is *2*, the lowest number in the list.

```
SET VAR vlmin = (LMIN(2,80,14,22))
```

7.12.6 LOG

(LOG(*arg*))

Returns log base *e* of *arg* (where *e* = 2.71828182845905). *Arg* must be a positive value and have a DOUBLE, REAL, NUMERIC, or INTEGER data type. LOG performs the inverse operation of EXP.

In the following example, the value of *vlog* is *0.6931*.

```
SET VAR vlog = (LOG(2))
```

7.12.7 LOG10

(LOG10(*arg*))

Returns log base 10 of *arg*. *Arg* must be a positive value and have a DOUBLE, REAL, NUMERIC, or INTEGER data type.

In the following example, the value of *vlog2* is *2*.

```
SET VAR vlog2 = (LOG10(100))
```

7.12.8 LSTDEV

(LSTDEV(*list*))

Returns the standard deviation for a list of values.

Remarks:

- The standard deviation is a measure of how widely values are dispersed from the average value.
- LSTDEV supports CURRENCY, DECIMAL, DOUBLE, FLOAT, REAL, NUMERIC, or INTEGER data type.

Example:

```
SET VAR vListStDev DOUBLE =
(LSTDEV(173.20,189.45,3929.14,434.75,333.25,257.50,88.70,641.86))
SHOW VAR vListStDev
1293.84205284038
```

7.12.9 LSUM

(LSUM(*list*))

Return the sum for a list of values.

Remarks:

- LSUM supports CURRENCY, DOUBLE, REAL, NUMERIC, INTEGER, BIGINT, SMALLINT, or BIGNUM data types.

Example:

```
SET VAR vListSum DOUBLE = (LSUM(189.45,79.14,43.75,33.25,27.58,789.40,6.12))
SHOW VAR vListSum
1168.69
```

7.12.10 LTRIM

(LTRIM(*text*))

Trims leading blanks from *text*, returning a text string.

In the following example, the value of *vltrim* is the text string *ABCDE* without the leading blanks.

```
SET VAR vltrim = (LTRIM('   ABCDE'))
```

7.12.11 LUC

(LUC(*arg*))

Converts *arg* from lowercase to uppercase, returning a text string.

In the following example, the value of *vluc* is an uppercase *A*.

```
SET VAR vluc = (LUC('a'))
```

7.12.12 LVARIANCE

(LSTDEV(*list*))

Return the variance for a list of values.

Remark:

- LVARIANCE supports CURRENCY, DECIMAL, DOUBLE, FLOAT, REAL, NUMERIC, or INTEGER data type.

Example:

```
SET VAR vListVariance DOUBLE =
(LVARIANCE(528.00,175.00,36.63,112.50,456.75,326.25,157.50,27.00,631.88))
SHOW VAR vListVariance
49438.425925
```

7.13 M

7.13.1 MAKEUTF8

(MAKEUTF8(*text*))

Converts upper ASCII characters text data into UTF-8 encoded characters.

Example:

```
R>SHOW VAR vText
Franz Wei? has met Anna G?rtner

R>SET VAR vUTF8Text = (MAKEUTF8(.vText))

R>SHOW VAR v%
Variable           = Value                              Type
------------------   ------------------------------     -------
vText              = Franz Wei? has met Anna            TEXT
                     G?rtner
vUTF8Text          = Franz Weiß has met Anna            TEXT
                     Gärtner
```

7.13.2 MOD

(MOD(*arg1*,*arg2*))

Computes a modulus or remainder of *arg1* divided by *arg2*. *Arg1* and *arg2* must be values that have DOUBLE, REAL, NUMERIC, or INTEGER data types and *arg2* cannot be 0.

In the following example, the value of *vmod* is *1*, the remainder when 3 is divided by 2.

```
     SET VAR vmod = (MOD(3,2))
```

7.14 N

7.14.1 NEXT

(NEXT(tblname,autonumcol))

Returns the next value of an autonumbered column.

Where colname is an autonumbered column in tblname NEXT returns, and increments, the value of the next available autonumber. You cannot use this function with INSERT, but you can use it with LOAD;NONUM. For example: Assume that you have autonumbered the column EmployeeID in the table Employees. The highest number currently used in the database is 134. In this case, in the following example, the value of vNextOne will be 135 and the value of vNextTwo will be 136.

Notice that the value has incremented even though no other commands or functions were issued.

```
SET VAR vNextOne = (NEXT(Employees,EmployeeID))
SET VAR vNextTwo = (NEXT(Employees,EmployeeID))
```

7.14.2 NINT

(NINT(*arg*))

Rounds a number that has a TEXT, REAL, DOUBLE, NUMERIC, or CURRENCY data type to the nearest integer, returning a value that has an INTEGER data type.

In the following example, the value of *vnint1* is *3* and the value of *vnint2* is *4*.

```
SET VAR vnint1 = (NINT(2.6))
SET VAR vnint2 = (NINT(4.4))
```

7.15 P

7.15.1 PMT1

(PMT1(*int,per,pv*))

Returns the amount of the periodic payment needed to pay off the present value, *pv*, based on the periodic interest rate, *int*, for the number of compounding periods, *per*.

In the following example, PMT1 returns the monthly payment amount for a loan of $12,000 with a 12% annual interest rate and five years to pay it off. The value of *vpmt1* (the monthly payment) is *$266.93*.

```
SET VAR vpmt1 = (PMT1(.01,60,12000))
```

7.15.2 PMT2

(PMT2(*int,per,fv*))

Returns the amount of the periodic payment to accrue the future value, *fv*, based on the periodic interest rate, *int*, for the number of compounding periods, *per*.

In the following example, PMT2 returns the monthly payment you must make if you want to have $25,000 in 10 years and your annual interest rate is 7%, compounded monthly. The value of *vpmt2* (the payment) is *$144.44*.

```
SET VAR vpmt2 = (PMT2((.07/12),120,25000))
```

7.15.3 PV1

(PV1(*pmt,int,per*))

Returns the present value of a series of equal payments of the amount, *pmt*, periodic interest rate, *int*, and compounding periods, *per*.

In the following example, PV1 returns the present value of an annuity with an annual interest rate of 9% and for which you want to pay $500 each month for 20 years. The value of *vpv1* (the present value) is *$55,572.48*.

```
SET VAR vpv1 = (PV1(500,.0075,240))
```

7.15.4 PV2

(PV2(*fv,int,per*))

Returns the present value based on the future value, *fv*, interest rate, *int*, and the number of compounding periods, *per*.

In the following example, PV2 returns the amount you must invest for a period of one year to return a future value of $3,500 where the annual rate is 7.6% (compounded daily). The value of *vpv2* (the amount to invest) is *$3,247.63*.

```
      SET VAR vpv2 = (PV2(3500,(.075/365),365))
```

7.16 R

7.16.1 RANDOM

(RANDOM(*value*))

Generates a random number between 0 and the value entered.

7.16.2 RATE1

(RATE1(*fv*,*pv*,*per*))

Returns the periodic interest rate required to return the future value, *fv*, based on the present value, *pv*, over the number of compounding periods, *per*.

In the following example, RATE1 returns the interest rate you must have if your initial investment is $8,500 and you want a future yield of $10,000 after 24 months. The value of *vrate1* (the interest rate) is *.0068*, a monthly rate of .68 percent, or 8.16% annually.

```
      SET VAR vrate1 = (RATE1(10000,8500,24))
```

7.16.3 RATE2

(RATE2(*fv*,*pmt*,*per*))

Returns the periodic interest rate on a series of regular payments, *pmt*, whose future value is *fv* over the number of compounding periods, *per*.

In the following example, RATE2 returns the interest rate you must have if your goal is $37,000 and you deposit $500 each month for five years. The value of *vrate2* is *.0069*, a monthly rate of .69 percent, or 8.28% annually.

```
      SET VAR vrate2 = (RATE2(37000,500,60))
```

7.16.4 RATE3

(RATE3(*pv*,*pmt*,*per*))

Returns the periodic interest rate required for an annuity of value *pv*, to return a series of equal payments, *pmt*, over a number of compounding periods, *per*.

In the following example, RATE3 returns the interest rate of an annuity, purchased at $50,000, which will pay $570 monthly for 10 years. The value of *vrate3* is *.0055*, a monthly rate of .55 percent, or 6.6% annually.

```
      SET VAR vrate3 = (RATE3(50000,570,120))
```

7.16.5 RDATE

(RDATE(*mon*,*day*,*yr*))

Converts integers *mon*, *day*, and *yr* to a DATE data type. *Yr* must be a four-digit year. The result returned by RDATE will vary, depending on the DATE format.

The following command assigns to the *transdate* column in the *transmaster* table the date derived from the values of *vmon* and *vday* (integers) as the month and day, and *1995* as the year in rows where the *transdate* column has a value.

```
UPDATE transmaster SET transdate = (RDATE(.vmon, .vday, 1995)) +
WHERE transdate IS NOT NULL
```

7.16.6 REVERSE

(RTRIM(*text*))

Returns the reverse order of text in a string.

In the following example, the value for vReverse is "erawtfoS evitcaretnI".

```
SET VAR vReverse TEXT = (REVERSE('Interactive Software'))
```

7.16.7 RJS

(RJS(*text*,*width*))

Right justifies *text* in *width* characters returning a text string.

In the following example, the value of *vrjs* is *ABCD*. The text string is right justified in a 10-character field.

```
SET VAR vrjs = (RJS('ABCD',10))
```

7.16.8 RNDDOWN

(RNDDOWN(arg1, arg2))

Returns a number rounded down to a specified number of digits.

Where: arg1 is the value to be rounded
arg2 is the position to be rounded down after the decimal point

Remarks:

- RNDDOWN behaves like ROUND, except that it always rounds a number down.
- If digits is greater than 0 (zero), then number is rounded down to the specified number of decimal places.
- If digits is 0, then number is rounded down to the nearest integer.
- If digits is less than 0, then number is rounded down to the left of the decimal point.

Examples

Example 01:

```
SET VAR v1 DOUBLE = (RNDDOWN(662.79,0))
SHOW VAR v1
662
```

Example 02:

```
SET VAR v2 DOUBLE = (RNDDOWN(662.79, 1))
SHOW VAR v2
662.7
```

Example 03:

```
SET VAR v3 DOUBLE = (RNDDOWN(54.1, -1))
SHOW VAR v3
50
```

Example 04:

```
SET VAR v4 DOUBLE = (RNDDOWN(55.1, -1))
SHOW VAR v4
50
```

Example 05:

```
SET VAR v5 DOUBLE = (RNDDOWN(-23.67, 1))
SHOW VAR v5
-23.6
```

7.16.9 RNDUP

(RNDUP(arg1, arg2))

Return a number value rounded up to a specified number of digits

Where: arg1 is the value to be rounded
arg2 is the position to be rounded up after the decimal point

Remarks:

- RNDUP behaves like ROUNDDWN, except that it always rounds a number up
- If arg2 is greater than 0 (zero), then number is rounded up to the specified number of decimal places.
- If arg2 is 0, then number is rounded up to the nearest integer.
- If arg2 is less than 0, then number is rounded up to the left of the decimal point.

Examples:

Example 01:

```
SET VAR vRoundUp1 DOUBLE = (RNDUP(762.273735,2))
SHOW VAR vRoundUp1
   762.28
```

Example 02:

```
SET VAR vRoundUp2 DOUBLE = (RNDUP(7796162.1455,-2))
SHOW VAR vRoundUp2
   7796200.
```

7.16.10 ROUND

(ROUND(arg1, arg2))

Where: arg1 is the value to be rounded
arg2 is the position to be rounded after the decimal point

Example 01:

```
SET VAR vNumber DOUBLE = 1.4567
SET VAR vRound = (ROUND(.vNumber,1))
```

Resulting vRound will be equal to 1.5

Example 02:

```
SET VAR vNumber DOUBLE = 1.4567
SET VAR vRound = (ROUND(.vNumber,2))
```

Resulting vRound will be equal to 1.46

Example 03:

```
SET VAR vNumber DOUBLE = 1.4567
SET VAR vRound = (ROUND(.vNumber,3))
```

Resulting vRound will be equal to 1.457

7.16.11 RTIME

(RTIME(*hrs*,*min*,*sec*,*frc*))

Converts integers *hrs*, *min*, *sec*, and *frc* to a TIME data type. *Hrs* is on a 24-hour scale. RTIME can be specified for up to thousandths of a second. The *frc* argument is optional.

In the following example, the value of *vrtime* is *12:15:30*, stored in internal R:BASE time format.

The time value will be displayed according to how you have set the TIME format. When you use time data in expressions, the result is given in seconds. You can use RTIME to convert this result to hours, minutes, and seconds. The value of *velapsed1* is *1170* seconds; the value of *velapsed2* is *0:19:30*.

```
SET VAR vrtime = (RTIME(12,15,30))
SET VAR vstart TIME = '1:10:40'
SET VAR vend TIME = '1:30:10'
SET VAR velapsed1 = (.vend - .vstart)
SET VAR velapsed2 = (RTIME(0,0,.velapsed1))
```

7.16.12 RTRIM

(RTRIM(*text*))

Trims trailing blanks from *text*, returning a text string.

In the following example, the value of *vrtrim* is the text string *ABCDE* without the trailing blanks.

```
SET VAR vrtrim = (RTRIM('ABCDE   '))
```

7.17 S

7.17.1 SFIL

(SFIL(*chr*,*nchar*))

Fills a text string with a specified character *chr*, for *nchar* characters up to 500. You cannot use a number as a character. Instead, assign the number to a variable that has a TEXT data type using the variable name in SFIL.

In the following example, the value of *vsfil* is the text string ==========. R:BASE programs often include SFIL to draw lines.

```
SET VAR vsfil = (SFIL('=',10))
```

7.17.2 SGET

(SGET(*text*,*nchar*,*pos*))

Gets *nchar* characters from *text* starting at *pos*, returning a text string.

In the following example, the value of *vsget* is *BCD*, the three characters starting in the second position of the text string.

```
SET VAR vsget = (SGET('ABCDE',3,2))
```

7.17.3 SIGN

(SIGN(*arg1*,*arg2*))

Transfers the sign of *arg2* to *arg1*. *Arg1* and *arg2* must be values that have DOUBLE, REAL, NUMERIC, or INTEGER data types.

In the following example, the value of *vsign* is *-15*, changing the sign of the first argument to the sign of the second argument.

```
SET VAR vsign = (SIGN(15,-20))
```

7.17.4 SIN

(SIN(*angle*))

Returns the trigonometric sine of *angle*.

In the following example, the value of *vsin* is *0.8659*.

```
SET VAR vsin = (SIN(1.047))
```

7.17.5 SINH

(SINH(*angle*))

Returns the hyperbolic sine of *angle*.

In the following example, the value of *vsinh* is *1.2491*.

```
SET VAR vsinh = (SINH(1.047))
```

7.17.6 SKEEP

(SKEEP(*source*, *chars*))

Keeps characters within the *source* string, using *chars* as a comparison.

This function is CASE SENSITIVE.

In the following example, the value of *vskeep* is *ldilliamennighway.*

```
SET VAR vskeep = (SKEEP('3935 Old William Penn
Highway','abcdefghijklmnopqrstuvwxyz'))
```

Spaces are also recognized.

In the following example, the value of *vskeep2* is *ld illiam enn ighway.*

```
SET VAR vskeep2 = (SKEEP('3935 Old William Penn Highway','
abcdefghijklmnopqrstuvwxyz'))
```

7.17.7 SKEEPI

(SKEEPI(*source*, *chars*))

Keeps characters within the *source* string, using *chars* as a comparison.

This function is NOT CASE SENSITIVE.

In the following example, the value of *vskeepi* is *OldWilliamPennHighway.*

```
SET VAR vskeepi = (SKEEPI('3935 Old William Penn
Highway','abcdefghijklmnopqrstuvwxyz'))
```

Spaces are also recognized.

In the following example, the value of *vskeepi2* is *Old William Penn Highway.*

```
SET VAR vskeepi2 = (SKEEPI('3935 Old William Penn Highway','
abcdefghijklmnopqrstuvwxyz'))
```

7.17.8 SLEN

(SLEN(*text*))

Returns the length of *text*. SLEN is often used to ensure that a string does not exceed the space allowed for it on a form, variable form, or report. When strings are concatenated or passed as parameters to a procedure file, the length of a string might be unknown.

In the following example, the value of *vslen* is *5*, the number of characters in the text string.

```
SET VAR vslen = (SLEN('ABCDE'))
```

7.17.9 SLOC

(SLOC(*text,string*))

Locates *string* in *text*, returning the position if the string is found, *0* if it is not found.
In the following example, the value of *vsloc1* is *3*.

```
SET VAR vsloc1 = (SLOC('ABCDE','C'))
```

The value of *vsloc2* in the following example is *0*, since the text string *X* does not exist in the text string *ABCDE*.

```
SET VAR vsloc2 = (SLOC('ABCDE','X'))
```

In the following example, the *custzip* column contains a customer's zip code, in which the first five characters might be followed by a dash and another four characters. If the first row contained the string *02178-5243*, *Sloc3* would capture the position of the dash (*6*), which is in the sixth position.

```
SET VAR v5zip = custzip IN customer WHERE COUNT = 1
SET VAR sloc3 = (SLOC(.v5zip,'-'))
```

7.17.10 SLOCI

(SLOCI(*TextNoteVarcharValue,string,arg*))

Returns the INTEGER value for the number of instances a string appears in a TEXT, NOTE, or VARCHAR value.

The argument parameter determines whether the string search is case sensitive, where "0" is not case sensitive while "1" is case sensitive.

In the following example, the number of instances of a colon is 2.

```
SET VAR vProduct1 INTEGER = (SLOCI('R:BASE Single Seat: Upgrade',':',0))
```

In the following example, the number of instances of an upper case S is 2.

```
SET VAR vProduct2 INTEGER = (SLOCI('RBZip Single Seat License: Upgrade','S',1))
```

In the following example, the number of instances of an upper case or lower case S is 3.

```
SET VAR vProduct3 INTEGER = (SLOCI('RBZip Single Seat License: Upgrade','S',0))
```

7.17.11 SLOCP

(SLOCP(*TextNoteVarcharValue,string,occurrence*))

Locates the exact position of a given string and occurrence in a TEXT, NOTE or VARCHAR value, returning the position if the string is found, *0* if it is not found. Using -1 as the third parameter will return the LAST occurrence.

In the following example, the value of *vslocp1* is 1.

```
SET VARIABLE v1 VARCHAR = 'ABCDEABC_AB'
SET VARIABLE vslocp1 = (SLOCP(.v1,'AB',1))
```

In the following example, the value of *vslocp2* is 6.

```
SET VARIABLE v1 VARCHAR = 'ABCDEABC_AB'
SET VARIABLE vslocp2 = (SLOCP(.v1,'AB',2))
```

7.17.12 SMOVE

(SMOVE(*text,pos1,nchar,string,pos2*))

From *text*, starting at position *pos1*, moves *nchar* characters to *string* starting at position *pos2*.

In the example below, the value of *vsmove1* is *XBCDX*. The characters *BCD* in the first string are moved into the second through fourth positions in the string *XYZXX*.

```
SET VAR vsmove1 = (SMOVE('ABCDE',2,3,'XYZXX',2))
```

In the following example, the *custcity* column is 12 characters wide and contains the string *ANCHORAGE* in the first row. You can use SMOVE to fill in a blank (13 characters in the example above) in order to customize a report title. The value of *vfulltitle* is *ANCHORAGE WAREHOUSE*.

```
SET VAR vcity = custcity IN customer WHERE COUNT = 1
SET VAR vfulltitle = (SMOVE(.vcity,1,12,'            WAREHOUSE',1))
```

7.17.13 SOUNDEX

(SOUNDEX(*value*))

Converts a text value to the corresponding SOUNDEX code.

7.17.14 SPUT

(SPUT(*text,string,pos*))

Puts *string* into *text,* starting at *pos*, returning a text string.

The value of *vsput1* in the following example is *AXCDE*. The character *X* is put into the second position in the string *ABCDE*.

```
SET VAR vsput1 = (SPUT('ABCDE','X',2))
```

7.17.15 SQRT

(SQRT(*arg*))

Returns square root of *arg*. *Arg* must be a positive value with a DOUBLE, REAL, NUMERIC, or INTEGER data type.

In the following example, the value of *vsqrt* is *10*.

```
SET VAR vsqrt = (SQRT(100))
```

7.17.16 SRPL

(SRPL(*sourcestring,searchstring,replacestring,*[0|1]))

Enables searching for and replacing text within a specified string of text.

sourcestring - specifies a string of text to search
searchstring - specifies text to search for
replacestring - specifies the replacement text
flag - 0 = replacement occurs for every matching string
    - 1 = replacement occurs for whole word matches only

The following "0 flag" example replaces *04/20/64* with *04-20-64*:

```
SET VAR vsrpl = (SRPL('04/20/64','/','-',0))
```

The following "1 flag" example replaces *Dear Joe* with *Dear Anne*:

```
SET VAR vsrpl = (SRPL('Dear Joe','Joe','Anne',1))
```

However, with this next "1 flag" example *DearJoe* remains the same as *DearJoe* is a whole word without a space.

```
SET VAR vsrpl = (SRPL('DearJoe','Joe','Anne',1))
```

7.17.17 SSTRIP

(SSTRIP(*source, chars*))

Strips characters from the *source* string, using *chars* as a comparison.

This function is CASE SENSITIVE.

In the following example, the value of *vsstrip* is *3935 O W P H.*

```
SET VAR vsstrip = (SSTRIP('3935 Old William Penn
Highway','abcdefghijklmnopqrstuvwxyz'))
```

Spaces are also recognized.

In the following example, the value of *vsstrip2* is *3935OWPH.*

```
SET VAR vsstrip2 = (SSTRIP('3935 Old William Penn Highway','
abcdefghijklmnopqrstuvwxyz'))
```

7.17.18 SSTRIPI

(SSTRIPI(*source*, *chars*))

Strips characters from the *source* string, using *chars* as a comparison.

This function is NOT CASE SENSITIVE.

In the following example, the value of *vsstripi* is *3935   , 15668.*

```
SET VAR vsstripi = (SSTRIPI('3935 Old William Penn Highway,
15668','abcdefghijklmnopqrstuvwxyz'))
```

Spaces are also recognized.

In the following example, the value of *vsstripi* is *3935,15668.*

```
SET VAR vsstripi = (SSTRIPI('3935 Old William Penn Highway, 15668','
abcdefghijklmnopqrstuvwxyz'))
```

7.17.19 SSUB

(SSUB(*text*,*n*))

Captures substring number *n* from *text*, returning a text string. Substrings in *text* are separated by a comma (or the current delimiter).

The SSUB function is often used with the CHOOSE command when capturing menu options from a pull-down menu.

In the following example, the value of *vssub* is *yearend*.

```
SET VAR vtext = 'reports,yearend'
SET VAR vssub = (SSUB(.vtext,2))
```

The following example shows that SSUB separates items based on a blank rather than the current delimiter when *n* is less than zero.

```
SET VAR vtext = 'reports yearend'
SET VAR vssub2 = (SSUB(.vtext,-2))
```

The following example also returns *yearend*. For this example, assume that *twodim* is a bar with a pull-down menu with the options *Edit* and *Enter* stored as text numbers according to their positions on the menu. The pop-up menu for both options contains a list of form names, so the second item in the CHOOSE variable will be a form name.

```
CHOOSE vtwodim FROM twodim
SET VARIABLE vbar = (SSUB(.vtwodim,1))
SET VARIABLE vpull = (SSUB(.vtwodim,2))
IF vbar = '1' THEN
   EDIT USING .vpull
ELSE
   ENTER .vpull
ENDIF
```

7.17.20 SSUBCD

(SSUBCD(*text,n,delimiter*))

Captures substring number n from text, returning a text string. Substrings in text are separated with a specified custom delimiter.

In the following example, the value for vSubString is "Reports".

```
SET VAR vSubString = (SSUBCD('Forms|Reports|Labels',2,'|'))
```

7.17.21 STRIM

(STRIM(*text*))

Trims trailing blanks from *text*, returning a text string.

In the following example, the value of *vstrim* is the text string *ABCDE* without the trailing blanks.

```
SET VAR vstrim = (STRIM('ABCDE   '))
```

7.18 T

7.18.1 TAN

(TAN(*angle*))

Returns the trigonometric tangent of *angle*.

In the following example, TAN defines a new column *newtan* for the *mytable* table. *Newtan* is a computed column providing the tangent of the angle in radians stored in the *oldangle* column.

```
ALTER TABLE mytable ADD newtan = (TAN(oldangle)) DOUBLE
```

7.18.2 TANH

(TANH(*angle*))

Returns the hyperbolic tangent of *angle*.

In the following example, the value of *vtanh* is *.7616*.

```
SET VAR vtanh = (TANH(1))
```

7.18.3 TDWK

(TDWK(*arg*))

Returns the day of the week as text, where *arg* is a value that has either a DATE or DATETIME data type.

In the following example, the value of *vtdwk* is *Saturday*.

```
SET VAR vtdwk = (TDWK('12/02/95'))
```

7.18.4 TERM1

(TERM1(*pv*,*int*,*fv*))

Returns the number of compounding periods (the term) for a return of future value *fv*, based on the present value, *pv*, and the interest rate, *int*.

In the following example, TERM1 returns the number of months your money must stay invested if you want to accumulate $10,000 on an initial investment of $5,000 at a compounded monthly rate of 1% (12% annually). The value of *vterm1* (the term) is *70*.

```
SET VAR vterm1 = (TERM1(5000,.01,10000))
```

7.18.5 TERM2

(TERM2(*pmt*,*int*,*fv*))

Returns the number of compounding periods (the term) for a return of future value *fv*, based on the payment, *pmt*, and interest rate, *int*.

In the following example, TERM2 returns the number of years you must make payments if you want to accumulate $75,000 by making annual installments of $2,000 at 8% annual interest. The value of *vterm2* (the term) is *18*.

```
SET VAR vterm2 = (TERM2(2000,.08,75000))
```

7.18.6 TERM3

(TERM3(*pmt*,*int*,*pv*))

Returns the number of periods (the term) for the present value, *pv*, to reach 0 based on the payment, *pmt*, and the interest rate, *int*.

In the following example, TERM3 returns the number of payments from a $15,000 annuity if the annual interest rate is 12.5% compounded monthly and you would like to receive $300 every month. The value of *vterm3* is *71*.

```
SET VAR vterm3 = (TERM3(300,(.125/12),15000))
```

7.18.7 TEXTRACT

(TEXTRACT(*datetime*))

Returns the time portion of DATETIME.

In the following example, the value of *vtextract* is *12:15:30.123*.

```
SET VAR vtextract = (TEXTRACT('08/09/95 12:15:30.123'))
```

7.18.8 TINFO

(TINFO(arg1*,arg2,arg3*))

Returns access right information for the current user.

Where: arg1 is zero (0), which specifies to show table permissions
arg2 is the system table ID (SYS_TABLE_ID) value (can be located in the SYS_TABLES system table)
arg3 can be:

a) A specific system column ID (SYS_COLUMN_ID) in that table
b) A value of 0 which means show permissions that apply to all the columns in that table
c) A value of -1 which means show permissions that apply to any of the columns in that table

Remarks:

- TINFO returns a comma delimited string of the access rights.
- In most instances table and column privileges are identical. Cases where they are not are autonumber column (no INSERT), computed columns, (no INSERT or UPDATE), and where the UPDATE privilege was granted to specific columns only.
- Permissions are returned based upon the current USER.

Examples:
The following example is based upon a database configured with an OWNER and users, where limited permissions are supplied to a user Jim for the Component table.

While connected to the database as the OWNER, the system table ID and system column ID can be found for the Component table and CompDesc column.

```
SELECT SYS_TABLE_ID FROM SYS_TABLES WHERE SYS_TABLE_NAME = 'Component'
 SYS_TABLE_ID
 ------------
           29

R>SELECT SYS_COLUMN_ID FROM SYS_COLUMNS WHERE SYS_COLUMN_NAME = 'CompDesc'
 SYS_COLUMN_ID
 -------------
           188

R>SET USER JIM JIM999
```

Example 01:
-- Permissions for the column CompDesc

```
R>SET VAR vUserInfoColumn = (TINFO(0,29,188))
R>SHOW VAR vUserInfoColumn
SELECT, UPDATE
```

Example 02:
-- Permissions for all column. Only SELECT is returned since that is the only permission applied to all columns.

```
R>SET VAR vUserInfoAllColumns = (TINFO(0,29,0))
R>SHOW VAR vUserInfoAllColumns
SELECT
```

Example 03:
-- Permissions for any column. Both SELECT and UPDATE are returned since there are some columns with both of those permissions.

```
R>SET VAR vUserInfoAnyColumn = (TINFO(0,29,-1))
R>SHOW VAR vUserInfoAnyColumn
SELECT, UPDATE
```

7.18.9 TMON

(TMON(*arg*))

Returns the month name as text where *arg* is a value that has either a DATE or DATETIME data type.

In the following example, the value of *vtmon* is *November*.

```
SET VAR vtmon = (TMON('11/12/95'))
```

7.18.10 TRANSLATE

(TRANSLATE(*'inputstring','characters','translations','pad'*))

Returns the string provided as a first argument after some characters specified in the second argument are translated into a destination set of characters. In the process, TRANSLATE replaces a single character at a time. For example, it will replace the 1st character in the "input string" with the 1st character in the "replacement string". Then, it will replace the 2nd character in the "input string" with the 2nd character in the "replacement string", and so forth. The pad parameter is optional which can be used instead of a blank in the return string.

In the following example the square and curly braces in the input string are replaced with parentheses, resulting with "2*(3+4)/(7-2)".

```
SET VAR vBracketSwitch = (TRANSLATE('2*[3+4]/{7-2}','[]{}','()()'))
```

In the following example the "w" and "t" replace the "8" and "7" corresponding number values in the string. The "9" and "0" are replaced with the pad character, resulting in "123456tw..".

```
SET VAR vNumberText = (TRANSLATE('1234567890','8790','wt','.'))
```

7.18.11 TRIM

(TRIM(*text*))

Trims leading and trailing blanks from *text*, returning a text string.

In the following example, the value of *vtrim* is the text string *ABCDE* without the leading and trailing blanks.

```
SET VAR vtrim = (TRIM('   ABCDE   '))
```

7.19 U

7.19.1 ULC

(ULC(*text*))

Converts *text* from uppercase to lowercase, returning a text string.

In the following example, the value of *vulc* is *abcde*. To ensure that your data is consistent, whether it is imported from outside R:BASE or entered through an R:BASE form, use ULC to convert text fields to lowercase.

```
SET VAR vulc = (ULC('ABCDE'))
```

Part VIII

8 R:BASE Reference Topics

8.1 Aggregate Functions

An aggregate function can be used to provide summary data about a rows in a table or for a provided list of values.

Function	Supported Areas	Description
AVG	SELECT, LAVG, Data Browser, Query Wizard, Query Builder, Form/Report/Label Expressions	Computes the numeric average. R:BASE rounds averages of integer values to the nearest integer value and currency values to their nearest unit.
COUNT	SELECT, Data Browser, Query Wizard, Query Builder, Form/Report/Label Expressions	Determines how many non-null entries there are for a particular column item.
LISTOF	SELECT, Query Builder, Form/Report/Label Expressions	Creates a text string of the values separated by the current comma delimiter character. The LISTOF function can be used with to populate a variable with a list of values from multiple rows.
MAX	SELECT, LMAX, Data Browser, Query Wizard, Query Builder, Form/Report/Label Expressions	Selects the maximum numeric, time, date, or alphabetic value in a column.
MIN	SELECT, LMIN, Data Browser, Query Wizard, Query Builder, Form/Report/Label Expressions	Selects the minimum numeric, time, date, or alphabetic value in a column.
STDEV	SELECT, Data Browser, Query Builder	Computes standard deviation. The standard deviation is a measure of how widely values are dispersed from the average value.
SUM	SELECT, LAVG, Data Browser, Query Wizard, Query Builder, Form/Report/Label Expressions	Computes the numeric sum.
VARIANCE	SELECT, Data Browser, Query Builder	Determines variance.

***** Selecting aggregate functions, such as MIN and MAX, requires that R:BASE keeps an accumulator and choose to only use the first 80 characters for NOTE values. This matches the fact that if you sort on NOTE fields, the sort will be based on the first 80 characters only.

8.2 Binary Large Objects (BLOB)

Binary Large Objects or BLOBs refer to images that you can store within your R:BASE database. In earlier Windows versions of R:BASE, the ability to add and manipulate large object data was relatively limited. However, with the latest Windows releases an integrated utility, the R:BASE BLOB Editor, has extended the functionality many times over.

You can add, edit, or delete Binary Large Objects (BLOBs) within your database files. The recommended table data type to store images is VARBIT. The R:BASE BLOB Editor has also been enhanced to manage Multipage Images. This enhancement will allow users to manage multipage images, such as .DCX, .GIF, or .TIFF files, when saved as BLOB data in R:BASE.

The R:BASE BLOB Editor also works with your large ASCII data files (Large Objects or LOBs). These objects refer to text files that can be in any ASCII format, including RTF. The recommended table data type for large text files is VARCHAR.

The data for VARBIT and VARCHAR data types is stored in the fourth R:BASE database file.

See also:

Data Types

8.2.1 Loading BLOB/LOB Data

You can easily load VARBIT and/or VARCHAR data into R:BASE tables directly from the Data Browser.

1. Open any table in the Data Browser where there are column(s) with VARBIT and/or VARCHAR data types
2. Press the [F4] key to turn the Data Browser into the Data Editor
3. Move your cursor focus to the column cell and right click on the field
4. From the speed menu, choose "Load From External File"
5. Browse and select the appropriate image/data file and then click on the "Open" button
6. The image/data file is now stored in the table row
7. To verify, double click the field to open the "R:BASE BLOB Editor". Notice that the appropriate tab is specified based on the type of data file you loaded.

8.2.2 Using Commands with BLOBs

Use the INSERT or LOAD commands to add binary large objects to your database. After the binary large object is loaded, the file is in the database, so you do not need the disk file. Following is an example of an INSERT command that adds a binary large object to a database:

```
INSERT INTO IMAGES (ID, IMAGEDATA) VALUES +
(1, ['filename.bmp'])
```

Binary large objects can be placed for viewing in forms and reports by placing the column or variable containing the binary large object in the form or report. When you run a form, you can enter or edit a reference to a binary large object by pressing [Shift] + [F10]. In an application program, you can use the SET VARIABLE command to define a variable that is equal to the file name that contains the binary large object.

You can write a variable that has a VARBIT or VARCHAR data type back to a file using the WRITE command, for example:

```
WRITE .v1 TO filename
```

The BACKUP and UNLOAD commands create a file with a .LOB extension for binary large objects, and a file for the data and/or structure.

See also:

Data Types

SELECT

8.3 Configuration File

Name: OTERRO11.CFG

The configuration file defines the Oterro database default configuration. Each time you start the Oterro database, OTERRO11.CFG sets the database operating environment by configuring multi-user functions, special character and operating conditions, and how the database processes and displays information.

If OTERRO11.CFG is not on the current drive and directory, or on the current path when you load the Oterro database, the database loads the default settings and creates a new OTERRO11.CFG file in the current directory.

OTERRO11.CFG is an ASCII file that you can edit with any ASCII text editor. This file contains comment lines, and startup settings. To change a startup setting, edit the appropriate line in the file. Changes saved to OTERRO11.CFG take effect the next time you start the Oterro database.

The following settings: MULTI, STATICDB, TRANSACT, and FASTLOCKS must be set before connecting to a database, therefore they cannot be set using the SET command SQLExecDirect. They must be set in the OTERRO11.CFG file or with SQLSetConnectOption.

The Oterro database special characters, case-folding, and case-sensitive character tables are stored in the database; when a database is connected, the database settings override the OTERRO11.CFG setting.

The following table includes examples of settings that are found in the OTERRO11.CFG file. The settings you can change are not limited to the settings included in this table.

Configuration Startup Options

Option	**Looks Like**	**Purpose**
Multi-user switch	MULTI ON	Turns the Oterro database multi-user features on or off.
SET keyword settings	DELIMIT=',' ZERO ON	Sets the Oterro database special characters and operating conditions; enclose the special character setting in quotes.
Case folding table	CASEP 97 65	Establishes correspondences between uppercase and lowercase characters.
Case sensitive collating table	COLLATEC 66 83 COLLATEC 67 87	Used when creating indexes for TEXT data types.
Collating table	COLLATE 97 65 COLLATE 192 65	Performs sorting and equality testing (>, >=, <, and, <=)
Expansion character table	EXPAND 132 97 101 EXPAND 142 65 69	Maps pairs of characters to each other to be treated as synonymous characters.
Character folding table	LCFOLD 65 97	Relates uppercase to lowercase characters for use in the string manipulation functions ULC, LUC, ICAP1, and ICAP2.

8.4 Constraints

To control the data that enters your database, you can apply powerful data-integrity rules called *constraints*. By applying a constraint to a column, you can prevent irreconcilable and empty data from being entered. R:BASE uses the following constraints:

- **Primary Key -** A column or set of columns that uniquely identify a row; in other words, each value in a primary key column is unique. A primary-key constraint prevents duplicate (non-unique) and null values from being entered. Even if you do not specifically define a constraint, all tables (in a well-designed database) should have a primary key. You can define one primary key per table.

- **Foreign Key -** A column or set of columns that match values in a particular primary key or unique key defined in a different table. A value cannot be inserted or changed.

- **Unique Key -** A column or set of columns that uniquely identify a row; in other words, each value in a unique-key column is unique. A unique-key constraint prevents duplicate (non-unique) and null values from being entered. The only difference between a unique key and a primary key is that you can define multiple unique keys per table.

- **Unique Index** - A column or set of columns that uniquely identify a row, but cannot be referenced like a primary key or unique key. The differences between a unique key and a unique index is that the unique key must be defined a Not NULL.

- **Not NULL** - Placing a not null constraint on a column requires that the data in the column must contain a value, and cannot be null. This prevents users from adding a "blank" value. A not null constraint cannot be added if the column already contains null values.

Constraints cannot be turned off and are always enforced; you must delete the constraint if you do not want it. However, because R:BASE works with constraints faster, use constraints instead of rules when possible.

You can remove constraints from columns. If you want to remove a primary- or unique-key constraint, you must first remove the foreign-key constraints that refer to it.

See also:

ALTER TABLE
CREATE INDEX
SET FASTFK

8.5 Cursors Explained

A cursor is a valuable programming tool. It is a pointer to rows in a table. A cursor lets you step through rows one by one, performing the same action on each row. You can set a cursor to point to all the rows in a table or to a subset of rows. A cursor is set using the DECLARE CURSOR command.

The DECLARE CURSOR command does not work by itself, but is really a sequence of commands. In addition to the DECLARE CURSOR, the OPEN and FETCH commands are required. A WHILE loop is used to step through the rows and perform the programmed action on each row. The CLOSE or DROP command is used after the cursor has stepped through all the rows.

The basic sequence of commands for a cursor is as follows:

```
DECLARE c1 CURSOR FOR +
   SELECT custid, company FROM customer
OPEN c1
FETCH c1 INTO vcustid1 INDI ind1, vcompany INDI ind2
WHILE SQLCODE <> 100 THEN
   -- Place code for row by row actions here.
   FETCH c1 INTO vcustid1 INDI ind1, vcompany INDI ind2
ENDWHILE
DROP CURSOR c1
```

The DECLARE CURSOR command names the cursor and defines the set of rows. The cursor name is then used in the OPEN, FETCH, CLOSE, and DROP commands that reference it. A cursor name can be up to 18 characters long and follows the same naming conventions as all other names in R:BASE.

More than one cursor can be defined and open at a time. SELECT is used in the DECLARE CURSOR to identify the rows to step through. The SELECT part of a cursor declaration can point to rows from a single table or from multiple tables, and can choose all or only some of the columns from a table. You can use the GROUP BY clause as well as the WHERE and ORDER BY clauses of SELECT.

The OPEN command initializes the cursor and tells R:BASE you are ready to retrieve a row of data from the cursor. The OPEN command positions the cursor at the first row of the set of data defined by the SELECT in the cursor declaration.

The FETCH command retrieves a row of data into the specified variables. The number of variables must match the number of columns listed in the SELECT part of the DECLARE CURSOR command. Each variable has a corresponding indicator variable, which tells if a NULL value was retrieved. The list of variable pairs - data variable and indicator variable - is separated by commas.

The FETCH command sets SQLCODE, the SQL error variable. If a row was retrieved, SQLCODE is set to 0. After the last row is retrieved, FETCH sets SQLCODE to 100 - no more data. Using SQLCODE as the condition for the WHILE loop lets you easily retrieve and act on each successive row. Placing a second FETCH command immediately before the ENDWHILE command keeps fetching rows until the end of data is reached. Then the loop exits.

Within the WHILE loop, place whatever commands are needed to operate on each row. You can look up additional data, perform mathematical calculations, update data, and so on.

When the cursor completes and the WHILE loop is exited, the cursor is dropped with the DROP CURSOR command. A cursor name must be dropped before it can be declared again. DROP removes a cursor definition from memory; to use the cursor again, it must be declared with the DECLARE CURSOR command. CLOSE leaves a cursor definition in memory; to use the cursor again, it is opened with the OPEN command. After a cursor has been closed, an OPEN repositions the pointer at the first row of the cursor definition. CLOSE is most often used with nested cursors, DROP with individual cursors.

When a cursor is open, you can use a special WHERE clause option, WHERE CURRENT OF cursorname. This WHERE clause works with the UPDATE, DELETE, and SELECT commands to perform the specified action on the row the cursor is currently pointing at. The DELETE deletes the entire row; the SELECT, and UPDATE only operate on columns included in the SELECT part of the DECLARE CURSOR command. Note that not every cursor definition supports use of the WHERE CURRENT OF cursorname.

It is not required to use the WHERE CURRENT OF cursorname in your WHERE clause. A WHERE clause that explicitly points to a row of data using values stored in variables can be used. The unique row identifier is fetched into a variable, then that value is used to access rows in the cursor table or other tables.

```
-- The special WHERE clause WHERE CURRENT OF
-- points to the current row of the cursor.
SELECT custid, company FROM customer +
WHERE CURRENT OF c1

UPDATE customer SET custid = (custid + 1000) +
   WHERE CURRENT OF c1

DELETE FROM customer WHERE CURRENT OF c1

-- Alternatively, use an explicit WHERE
-- clause to access a row.

SELECT custid, company FROM customer +
   WHERE custid = .vcustid

UPDATE customer SET custid = (custid + 1000) +
   WHERE custid = .vcustid

DELETE FROM customer WHERE custid = .vcustid
```

This is the basic cursor structure. Other types of cursors and cursor structures that are used are: multi-table cursors, non-updatable cursors, nested cursors, resettable cursors, and scrolling cursors. Each is briefly described below.

8.5.1 Multi-Table Cursors

A multi-table cursor includes more than one table in the SELECT part of the cursor declaration. The tables can be linked directly within the DECLARE CURSOR command; avoiding steps to define a view to retrieve data from more than one table.

The DECLARE CURSOR command has the full capabilities of the SELECT command to do multi-table queries. As with the SELECT command itself, you list the columns to retrieve, the tables to get the data from, then link the tables in the WHERE clause. For example,

```
-- Select data from both the Customer
-- and Transmaster tables.
```

```
DECLARE C1 CURSOR FOR SELECT +
custid, company, transid, transdate, invoicetotal +
   FROM customer, transmaster +
   WHERE customer.custid = transmaster.custid
OPEN C1

-- The fetch retrieves all the specified columns into variables.
FETCH C1 INTO vcustid1 INDI ind1, vcompany INDI ind2 +
   vtransid INDI ind3, vtransdate INDI ind4, vinvoicetotal INDI ind5
WHILE SQLCODE <> 100 THEN

   -- Place code for row by row actions here.
   -- An explicit WHERE clause must be used,
   -- WHERE CURRENT OF is not supported with
   -- multi-table cursors.

   -- Get the next row
   FETCH C1 INTO vcustid1 INDI ind1, vcompany INDI ind2 +
      vtransid INDI ind3, vtransdate INDI ind4, vinvoicetotal INDI ind5
ENDWHILE
DROP CURSOR C1
```

Notice that the basic structure of the cursor commands doesn't change. You still declare the cursor, open it, fetch the first row, then use a WHILE loop to step through each row. There is no limit to the number of tables that can be included in a DECLARE CURSOR command. The tables are joined together in the same way they are joined with a regular SELECT command.

A multi-table cursor definition is a non-updatable cursor, however. You cannot update the cursor directly by using WHERE CURRENT OF cursorname. You must use explicit WHERE clauses to access the cursor tables.

8.5.2 Non-Updatable Cursors

A non-updatable cursor is one that does not support use of the special WHERE clause WHERE CURRENT OF cursorname. An explicit WHERE clause must be used to access data in the tables.

A non-updatable cursor is a multi-table cursor, or a cursor that is defined, for example, using the GROUP BY clause. The SELECT command that defines the cursor rows does not allow the cursor to point back to a single specific row in a table.

Non-updatable cursors are a very useful part of the DECLARE CURSOR structure. Use the power of the SELECT command in the DECLARE CURSOR declaration to dramatically improve the performance of a cursor. The more work the cursor does, the less your program has to do and the faster and more efficiently it will run.

When using a non-updatable cursor, make sure you fetch a unique row identifier for use in WHERE clauses.

8.5.3 Nested Cursors

A nested cursor involves two DECLARE CURSOR definitions. The second cursor is dependent on the first and its cursor definition uses a variable value fetched by the first cursor.

There is a specific structure recommended for nested cursors - a row is retrieved from cursor one, then the matching rows in cursor two are retrieved and stepped through. Then the next row is retrieved from cursor one and its matching rows from cursor two are stepped through. The process continues until all rows have been retrieved from cursor one.

Example

```
-- The DECLARE commands are done together
-- at the top of the program.
-- An OPEN cursor does not need to immediately
-- follow the corresponding DECLARE CURSOR command
SET VAR vcustid INTEGER
DECLARE c1 CURSOR FOR SELECT custid, company +
   FROM customer ORDER BY company
-- The second cursor uses a variable in the
-- WHERE clause. This variable, vcustid, must be
-- defined earlier in the program.
-- The cursor retrieves rows for a single customer only
DECLARE c2 CURSOR FOR +
   SELECT custid, contfname, contlname +
   FROM contact WHERE custid = .vcustid

-- Cursor c1 is opened and the first row retrieved
-- from the Customer table
OPEN c1
FETCH c1 INTO vcustid1 INDI ind1, vcompany INDI ind2
WHILE SQLCODE <> 100 THEN

   -- Cursor c2 is opened, it points to all the
   -- rows in the Contact table that match the
   -- custid fetched into vcustid by cursor c1.
   OPEN c2

   -- Get the first row from the contact table and step
   -- through all matching rows.
   FETCH c2 INTO vcustid1 INDI ind1, vfirstname INDI ind2, +
      vlastname INDI ind3
   WHILE SQLCODE <> 100 THEN

      -- Place code here to do row by row actions

      --Get the next row for cursor c2
      FETCH c2 INTO vcustid1 INDI ind1, vfirstname INDI ind2, +
         vlastname INDI ind3
   ENDWHILE

   -- After all the matching rows in the contact table
   -- have been processed, close cursor c2 and get the
   -- next row from the Customer table.
   -- Cursor c2 is closed and not dropped because
   -- the definition will be reused for the next
   -- row from cursor c1.
   CLOSE c2

   -- Get the next row for cursor c1
   FETCH c1 INTO vcustid1 INDI ind1, vcompany INDI ind2
ENDWHILE

-- Both cursors are dropped when all the rows
-- in the Customer table have been retrieved.
DROP CURSOR c2
DROP CURSOR c1
```

You can use the same WHILE loop condition, SQLCODE <> 100, for both cursors. This works very well and there is no conflict between the two loops. The relative FETCH command sets the value of SQLCODE.

Notice that the FETCH from cursor c2 is right before the ENDWHILE of the inner WHILE loop ensuring that that FETCH command is the one being tested by the WHILE loop. The FETCH from cursor c1 is right before the ENDWHILE of the outer WHILE loop, which then continues based on cursor c1. This placement of the DECLARE, OPEN, FETCH, WHILE, and ENDWHILE statements will always work. Just make sure the ENDWHILE is the next command after the FETCH.

With nested cursors, the inner cursor is closed and opened so that it always references the matching rows from the outer cursor. An alternative to opening and closing the inner cursor is to use the RESET option on the OPEN command.

8.5.4 Resettable Cursors

A DECLARE CURSOR can use a variable in its WHERE clause. Each time the cursor is opened, the WHERE clause is reevaluated using the current variable value and identifies a new set of data.

You can CLOSE and OPEN a defined cursor, or use the OPEN cursorname RESET command. Don't use the CLOSE command if you place the RESET option on the OPEN command. The RESET option automatically reevaluates the variable value and identifies a new set of data for the cursor.

OPEN cursorname RESET is commonly used with nested cursors. The second cursor is dependent on a variable fetched by the first cursor. By using RESET, you won't need to CLOSE the inner cursor each time.

Using the RESET option on OPEN is faster using than the OPEN, CLOSE sequence of commands.

8.5.5 Scrolling Cursors

Normally, cursors move through the data in one direction only, from top to bottom. They move forward one-by-one through the set of defined rows. Once a row has been accessed and passed over, you can't get back to it. The rows can be ordered in the cursor definition - the top to bottom order is not necessarily the table order.

When a cursor is defined as a scrolling cursor, you gain the capability of moving both forwards and backwards through the rows of data and can also jump past rows.

To define a cursor as a scrolling cursor, include the word SCROLL in the DECLARE CURSOR command. For example,

```
DECLARE c1 SCROLL CURSOR FOR SELECT ...
```

The word SCROLL comes right after the cursor name. If SCROLL is not included in the cursor definition, the cursor can only move forward through the rows one at a time.

Once a cursor is defined as a scrolling cursor, a number of additional options on the FETCH command become available. These options are as follows; note that the directions and positions are based on the order of the rows as specified by the DECLARE CURSOR command, not on the order of the rows in the actual table:

NEXT - The default option if none is specified on the FETCH command. NEXT moves the cursor forward through the rows, it gets the next available row based on the current cursor position. NEXT steps through the rows one-by-one going forward.

PRIOR - Moves the cursor backwards through the rows. The PRIOR option gets the previous row based on the current cursor position, and steps through the rows one-by-one going backwards.

FIRST - Moves the cursor from its current position to the first row. This option jumps immediately to the first row as determined by the DECLARE CURSOR command. A FETCH NEXT then finds the second row. The cursor is repositioned at the beginning of the set of rows.

LAST - Moves the cursor from its current position immediately to the last row as specified by the DECLARE CURSOR command. A FETCH PRIOR then finds the next to last row; a FETCH NEXT returns "end of data encountered". LAST jumps over the rows between the current cursor position and the last row.

ABSOLUTE n - Moves the cursor the specified number of rows from the first row of data as determined by the DECLARE CURSOR and OPEN commands. A positive number must be specified; you can't use this option to move backwards. The intervening rows are jumped over. You can't jump past the last row; if the number given is greater than the number of rows retrieved, an "end of data" error is returned.

RELATIVE n - Moves the cursor the specified number of rows from the current cursor position. This option moves the cursor either forwards or backwards - forwards if a positive number is specified, backwards if a negative number is specified. The intervening rows are jumped over. You can't jump past the last row or the first row; an "end of data" error is returned if the specified number would take you past the beginning or end of the selected rows.

Example
To see how a scrolling cursor can be used in an application, imagine you have a group of customers to contact each day. The scrolling cursor retrieves the list of customers for today. They are ordered by company name. The first row is brought up in a menuless form. The form remains on the screen when you are done with the record, and a CHOOSE menu pops up giving the user choices as to which record to select next.

You can: move through the list of customers one-by-one, both forwards and backwards, jump to the last record and back to the first record, jump past a group of records, and search for a particular record by last name or by company name

Each time you select a record, the cursor is repositioned ready for the next selection.

```
--WALKLIST.RMD
--scroll through a list of customers
SET MESSAGE OFF
SET ERROR MESSAGE OFF
SET VAR vCheckCursor INTEGER = (CHKCUR('c1'))
IF vCheckCursor = 1 THEN
  DROP CURSOR c1
ENDIF
CLS

--Define the scrolling cursor
DECLARE C1 scroll CURSOR FOR +
   SELECT CustId, LastName, Company FROM Customer +
   WHERE calldate = .#DATE ORDER BY Company

--Open the cursor and get the first row
OPEN C1
FETCH FIRST FROM C1 INTO +
   VCustId INDI ICustId, VLastname INDI ILastname, VCompany INDI ICompany
WHILE SQLCODE <> 100 THEN

   --Bring up the form with the data from the first row.
   --After the form is closed, choose from the menu which record to retrieve next
   EDIT USING CustForm WHERE CustId = .VCustId
   CHOOSE VAction FROM #LIST +
      'Next Customer,Previous Customer,Jump Forward "n",Jump Backward "n",+
      Last Customer,First Customer,Search by Last Name,Search by Company' +
      TITLE 'Select Customer' CAPTION 'Choose' LINES 8 FORMATTED
   IF VAction = '[Esc]' THEN
      RETURN
   ENDIF

   --The switch/case block determines which record to retrieve
   SWITCH (.VAction)

   --Move forward one row at a time
```

```
CASE 'Next Customer'
   FETCH NEXT FROM C1 INTO +
      VCustId INDI ICustId, VLastname INDI ILastname, +
      VCompany INDI ICompany

   --If already on the last row, stay there
   IF SQLCODE = 100 THEN
      FETCH LAST FROM C1 INTO +
      VCustId INDI ICustId, VLastname INDI ILastname, +
      VCompany INDI ICompany
   ENDIF
   BREAK

--Move backward one row at a time
CASE 'Previous Customer'
   FETCH PRIOR FROM C1 INTO +
      VCustId INDI ICustId, VLastname INDI ILastname, +
      VCompany INDI ICompany

   --If already on the first row, stay there
   IF SQLCODE = 100 THEN
      FETCH FIRST FROM C1 INTO +
      VCustId INDI ICustId, VLastname INDI ILastname, +
      VCompany INDI ICompany
   ENDIF
   BREAK

--Move forward the specified number of records
CASE 'Jump Forward "n"'
   DIALOG 'How many to jump forward?' VNum=4 VEndKey 1
   SET VAR VPlus = (INT(.VNum))

   --R:BASE counts from the current cursor position
   FETCH RELATIVE .vplus FROM C1 INTO +
      VCustId INDI ICustId, VLastname INDI ILastname, +
      VCompany INDI ICompany

   --If the number of records to jump past takes you beyond the last
   --record, the last record is retrieved
   IF SQLCODE = 100 THEN
      FETCH LAST FROM C1 INTO +
         VCustId INDI ICustId, VLastname INDI ILastname, +
         VCompany INDI ICompany
   ENDIF
   BREAK

--Move backward the specified number of records
CASE 'Jump Backward "n"'
   DIALOG 'How many to jump backward?' VNum=4 VEndKey 1
   SET VAR VMinus = (INT(.VNum) * -1)

   --R:BASE counts from the current cursor position
   FETCH RELATIVE .vminus FROM C1 INTO +
      VCustId INDI ICustId, VLastname INDI ILastname, +
      VCompany INDI ICompany

   --If the number of records to jump past takes you beyond the first
   --record, the first record is retrieved
   IF SQLCODE = 100 THEN
      FETCH FIRST FROM C1 INTO +
         VCustId INDI ICustId, VLastname INDI ILastname, +
         VCompany INDI ICompany
   ENDIF
   BREAK

--Jump to the last record Next customer from the last record returns end-of-
data
CASE 'Last Customer'
   FETCH LAST FROM C1 INTO +
      VCustId INDI ICustId, VLastname INDI ILastname, +
      VCompany INDI ICompany
   BREAK

--Jump to the first record Prior customer from the first record returns end-of-
data
CASE 'First Customer'
   FETCH FIRST FROM C1 INTO +
      VCustId INDI ICustId, VLastname INDI ILastname, +
      VCompany INDI ICompany
   BREAK

--Prompt for the last name to find
CASE 'Search by Last Name'
   SET VAR vsearch = NULL
   DIALOG 'Enter the last name to find' +
      VSearch VEndKey 1
   IF VEndKey = '[Esc]' THEN
      BREAK
   ENDIF

   WHILE #PI <> 0.0 THEN

      --Search forward for a matching record
      FETCH NEXT FROM c1 INTO +
         VCustID INDI ICustId, VLastname INDI ILastname, +
         VCompany INDI ICompany

      --If a match is found, the row is displayed and the cursor repositioned
      --at that row
      IF VLastname CONTAINS .VSearch THEN
         BREAK
      ENDIF

      --If no match was found, the search can be continued from the first row.
      IF SQLCODE = 100 THEN
         DIALOG 'No match found. Continue search from beginning?' +
            VResp VEndKey YES
         IF VEndKey = '[Esc]' THEN
            BREAK
         ENDIF

         IF VResp = 'YES' THEN
            FETCH FIRST FROM c1 INTO +
               VCustID INDI ICustId, VLastname INDI ILastname, +
               VCompany INDI ICompany
            IF VLastname CONTAINS .VSearch THEN
```

```
               BREAK
            ENDIF
         ELSE
            --If the search is not continued, the last row is retrieved
            FETCH LAST FROM c1 INTO +
               VCustID INDI ICustId, VLastname INDI ILastname, +
               VCompany INDI ICompany
            BREAK
         ENDIF
      ENDIF
   ENDWHILE
   BREAK

--Prompt for the company name to find
CASE 'Search by Company'
   SET VAR VSearch = NULL
   DIALOG 'Enter the company to find' +
      VSearch VEndKey 1
   IF VEndKey = '[Esc]' THEN
      BREAK
   ENDIF

   --Search forward for a matching company record If a match is found,
   --the row is displayed and the cursor repositioned at that row
   WHILE #PI <> 0.0 THEN
      FETCH NEXT FROM c1 INTO +
         VCustID INDI ICustId, VLastname INDI ILastname, +
         VCompany INDI ICompany
      IF VCompany CONTAINS .VSearch THEN
         BREAK
      ENDIF

      --If no match was found, the search can be continued from the first row.
      IF SQLCODE = 100 THEN
         DIALOG 'No match found. Continue search from beginning?' +
            VResp VEndKey YES
         IF VEndKey = '[Esc]' THEN
            BREAK
         ENDIF
            IF VResp = 'YES' THEN
            FETCH FIRST FROM c1 INTO +
               VCustID INDI ICustId, VLastname INDI ILastname, +
               VCompany INDI ICompany
            IF VCompany CONTAINS .VSearch THEN
               BREAK
            ENDIF
         ELSE
            --If the search is not continued, the last row is retrieved.
            FETCH LAST FROM c1 INTO +
               VCustID INDI ICustId, VLastname INDI ILastname, +
               VCompany INDI ICompany
            BREAK
         ENDIF
      ENDIF
   ENDWHILE
   BREAK
ENDSW
```

```
ENDWHILE
DROP CURSOR C1
RETURN
```

8.5.6 Optimizing Cursors

DECLARE CURSOR is not always the fastest way to accomplish a task, particularly an UPDATE or an INSERT. If you can replace your DECLARE CURSOR routine with a single SQL command, you will dramatically improve performance. However, some tasks require a DECLARE CURSOR.

Let the cursor do the work
To improve the performance of a DECLARE CURSOR routine, do as much work in the DECLARE CURSOR as possible. This is the single most important factor in improving cursor performance. Do whatever work can be done in the SELECT command part of the DECLARE CURSOR - select as many columns of data as possible and also do calculations there if you can. The DECLARE CURSOR does the operation only once; inside the WHILE loop, the command is repeated for each row that is stepped through.

To do actions for unique rows only, use SELECT DISTINCT in the cursor definition instead of adding code to your WHILE loop to test the row values to see if they are the same or different. Use the SELECT functions to sum, average, count and so on in the cursor definition instead of for each row in the WHILE loop. Select as many columns as possible in the DECLARE CURSOR rather than retrieve the data each row in the WHILE loop.

The fewer commands repeated in the WHILE loop, the faster your DECLARE CURSOR will run. Remember that each command in the WHILE loop is repeated for each row retrieved by the DECLARE CURSOR. Use optimized variables in the WHILE loop -initialize each variable outside the WHILE loop, and do not change the data type of variables in the loop.

Following are two examples showing progressive changes made to a DECLARE CURSOR routine to improve performance.

Example 1 - posting. The task is to sum the extended price column in the transaction detail, Transdetail, table for each transaction ID, then update the transaction header, Transmaster, table with the sum. An initial approach is to declare a cursor on the header table, then step through all matching rows in the detail table. After all the matching detail rows have been processed, the header table is updated.

```
*(POST1.RMD -- the worst case)
-- nested declare cursors
-- strictly linear programming
SET VAR vtotal CURR
DECLARE c1 CURSOR FOR SELECT transid, netamount +
   FROM transmaster
OPEN c1
FETCH c1 INTO vtransid INDI vind1, vnetamount INDI vind2
WHILE SQLCODE <> 100 THEN
   DECLARE c2 CURSOR FOR SELECT extprice +
      FROM transdetail WHERE transid = .vtransid
   OPEN c2
   FETCH c2 INTO vprice INDI vind3
   WHILE SQLCODE <> 100 THEN
      SET VAR vtotal = (.vtotal + .vprice)
      FETCH c2 INTO vprice INDI vind3
   ENDWHILE
   DROP CURSOR c2
   UPDATE transmaster SET netamount = .vtotal +
      WHERE CURRENT OF c1
   SET VAR vtotal = NULL
   FETCH c1 INTO vtransid INDI vind1, vnetamount INDI vind2
ENDWHILE
DROP CURSOR c1
```

We can speed up this code by following the recommended structure for nested cursors. If we move the second DECLARE CURSOR out of the WHILE loop and reset the cursor instead of dropping it, this command file will execute faster. However, the best way to improve this code is by removing the second DECLARE CURSOR altogether. We don't need to step through all the rows in the detail table - we can compute the sum with a single SELECT command.

```
*(POST2.RMD - a little bit better)
-- use the SELECT or COMPUTE command
-- to calculate the sum instead of a nested cursor
SET VAR vprice CURR = NULL
DECLARE c1 CURSOR FOR SELECT transid, netamount +
   FROM transmaster
OPEN c1
FETCH c1 INTO vtransid INDI vind1, vamount INDI vind2
WHILE SQLCODE <> 100 THEN
   SELECT SUM(extprice) INTO vprice +
      FROM transdetail WHERE transid = .vtransid
   -- if no matching rows in the Transdetail table,
   -- vprice is null
   IF vprice IS NOT NULL THEN
      UPDATE transmaster SET netamount = .vprice +
         WHERE CURRENT OF c1
   ENDIF
   SET VAR vprice = NULL
   FETCH c1 INTO vtransid INDI vind1, vamount INDI vind2
ENDWHILE
DROP CURSOR c1
```

This simple change reduced the number of commands in the program, which in turn improved performance. All the commands inside the WHILE loop still need to be executed for as many rows as are in the Transmaster table, however. The Transmaster table has fewer rows than the Transdetail table, so a valid assumption is to place the cursor on the Transmaster table to repeat the WHILE loop the fewest times.

However, if we place the cursor on the detail table instead of on the header table, the sum can be calculated directly in the DECLARE CURSOR. Because the command is grouped by the transaction ID, the same number of rows is retrieved by the cursor. The only commands to repeat in the WHILE loop are the UPDATE and the FETCH to get the next row. At first this might seem backwards, but computing the sum in the DECLARE CURSOR is much faster.

```
*(POST3.RMD - better yet)
-- declare the cursor on the detail table and
-- do the sum directly in the cursor definition
DECLARE c1 CURSOR FOR SELECT transid, SUM(extprice) +
   FROM transdetail GROUP BY transid
OPEN c1
FETCH c1 INTO vtransid INDI vind1, vprice INDI vind2
WHILE SQLCODE <> 100 THEN
   -- this is a non-updatable cursor so an explicit
   -- WHERE clause is used
   UPDATE transmaster SET netamount = .vprice +
      WHERE transid = .vtransid
   FETCH c1 INTO vtransid INDI vind1, vprice INDI vind2
ENDWHILE
DROP CURSOR c1
```

The number of commands has been reduced by over half from the first program, and performance by more than that. The multi-table update command is actually the fastest way to accomplish this task.

```
*(POST4.RMD - do a multi-table update if you can)
-- multi table update command, a view is used
-- to first calculate the sum and create a
-- one-one relationship
DROP VIEW v_trans
CREATE VIEW v_trans (transid, amount) AS +
   SELECT transid, SUM(extprice) +
   FROM transdetail GROUP BY transid
UPDATE transmaster SET netamount = amount +
   FROM transmaster ,v_trans t2 +
   WHERE transmaster.transid = t2.transid
```

Example 2 - a quick report. The task here is to create a quick report of companies from the Customer table and their corresponding contact names from the Contact table. Using nested cursors makes printing the company information once followed by the many rows of contact information easier.

```
*(CUSTREP1.RMD - the worst case)
-- nested cursors are used with the declare for
-- the second cursor inside the while loop of
-- the first cursor. Also, the data is retrieved
-- with a SELECT command instead of in the
-- cursor definition

-- Dropping a cursor before you declare it is a
-- technique used to guarantee that the cursor does
-- not exist in memory. The DROP CURSOR normally
-- returns an error message, so check to verify it
-- exists before dropping it.
SET VAR vCheckCursor1 INTEGER = (CHKCUR('c1'))
IF vCheckCursor1 = 1 THEN
  DROP CURSOR c1
ENDIF
SET VAR vCheckCursor2 INTEGER = (CHKCUR('c2'))
IF vCheckCursor2 = 1 THEN
  DROP CURSOR c2
ENDIF

-- Only the unique row identifier is specified in
-- the cursor definition
DECLARE c1 CURSOR FOR SELECT custid FROM customer +
   ORDER BY custid
OPEN c1
FETCH c1 INTO vcustid INDI ind1
WHILE SQLCODE <> 100 THEN

   -- Retrieve and display the rest of the
   -- data for a customer
   SELECT company, custaddress, custcity, +
      custstate, custzip, custphone INTO +
      vcompany INDI vi1, vaddress INDI vi2, +
      vcity INDI vi3, vstate INDI vi4, +
      vzipcode INDI vi5, vphone INDI vi6 +
      FROM customer WHERE custid = .vcustid
   SET VAR vcsz = (.vcity + ',' & .vstate & .vzipcode)
   WRITE .vcustid, .vcompany
   WRITE .vaddress
   WRITE .vcsz
```

```
      -- Declare a cursor to identify matching contact rows
      DECLARE c2 CURSOR FOR SELECT contfname, contlname +
         FROM contact WHERE custid = .vcustid
      OPEN c2
      FETCH c2 INTO vfname INDI i1, vlname INDI i2
      WHILE SQLCODE <> 100 THEN
         SET VAR vfullname = (.vfname & .vlname)
         WRITE .vfullname
         FETCH c2 INTO vfname INDI i1, vlname INDI i2
      ENDWHILE
      DROP CURSOR c2
      FETCH c1 INTO vcustid INDI ind1
   ENDWHILE
   DROP CURSOR c1
```

The next code segment shows the recommended structure for nested cursors. The second DECLARE CURSOR is moved to the top of the program, and the second cursor is opened and closed, not declared and dropped. Just this simple change improves performance.

```
   *(CUSTREP2.RMD - move cursor out of WHILE loop)
   SET VAR vCheckCursor1 INTEGER = (CHKCUR('c1'))
   IF vCheckCursor1 = 1 THEN
     DROP CURSOR c1
   ENDIF
   SET VAR vCheckCursor2 INTEGER = (CHKCUR('c2'))
   IF vCheckCursor2 = 1 THEN
     DROP CURSOR c2
   ENDIF

   SET VAR vcustid INTEGER
   DECLARE c1 CURSOR FOR SELECT custid +
      FROM customer ORDER BY custid
   DECLARE c2 CURSOR FOR SELECT contfname, contlname +
      FROM contact WHERE custid = .vcustid

   -- Get the first row of data for a customer
   OPEN c1
   FETCH c1 INTO vcustid INDI ind1
   WHILE SQLCODE <> 100 THEN

      -- Retrieve and display the rest of the
      -- data for a customer
      SELECT company, custaddress, custcity, +
         custstate, custzip, custphone INTO +
         vcompany INDI vi1, vaddress INDI vi2, +
         vcity INDI vi3, vstate INDI vi4, +
         vzipcode INDI vi5, vphone INDI vi6 +
         FROM customer WHERE custid = .vcustid
      SET VAR vcsz = (.vcity + ',' & .vstate & .vzipcode)
      WRITE .vcustid, .vcompany
      WRITE .vaddress
      WRITE .vcsz

      -- Open cursor c2, retrieve and display
      -- the matching contact data
      OPEN c2
      FETCH c2 INTO vfname INDI i1, vlname INDI i2
      WHILE SQLCODE <> 100 THEN
```

```
         SET VAR vfullname = (.vfname & .vlname)
         WRITE .vfullname
         FETCH c2 INTO vfname INDI i1, vlname INDI i2
      ENDWHILE

      -- Close cursor c2 and get the next row of
      -- customer data
      CLOSE c2
      FETCH c1 INTO vcustid INDI ind1
   ENDWHILE
   DROP CURSOR c1
   DROP CURSOR c2
```

Moving the data retrieval to the DECLARE CURSOR command instead of using a separate SELECT command again improves performance.

```
   *(CUSTREP3.RMD)
   --retrieve data through DECLARE CURSOR
   SET VAR vCheckCursor1 INTEGER = (CHKCUR('c1'))
   IF vCheckCursor1 = 1 THEN
     DROP CURSOR c1
   ENDIF
   SET VAR vCheckCursor2 INTEGER = (CHKCUR('c2'))
   IF vCheckCursor2 = 1 THEN
     DROP CURSOR c2
   ENDIF

   -- retrieve all the data through the DECLARE CURSOR
   -- command instead of SELECT
   SET VAR vcustid INTEGER
   DECLARE c1 CURSOR FOR SELECT custid, company, +
      custaddress, custcity ,custstate, custzip, +
      custphone FROM customer ORDER BY custid
   DECLARE c2 CURSOR FOR SELECT contfname, contlname +
      FROM contact WHERE custid = .vcustid
   OPEN c1

   -- Get the first row of customer data
   FETCH c1 INTO vcustid INDI ind1, vcompany INDI ind2,+
      vaddress INDI ind3, vcity INDI ind4, vstate INDI ind5, +
      vzip INDI ind6, vphone INDI ind7
   WHILE SQLCODE <> 100 THEN

      -- Display the customer data and open cursor c2 to
      -- retrieve the matching contact data
      SET VAR vcsz = (.vcity + ',' & .vstate & .vzipcode)
      WRITE .vcustid, .vcompany
      WRITE .vaddress
      WRITE .vcsz
      OPEN c2
      FETCH c2 INTO vfname INDI i1, vlname INDI i2
      WHILE SQLCODE <> 100 THEN
         SET VAR vfullname = (.vfname & .vlname)
         WRITE .vfullname
         FETCH c2 INTO vfname INDI i1, vlname INDI i2
      ENDWHILE

      -- Close cursor c2 and get the next row of
      -- customer data
      CLOSE c2
      FETCH c1 INTO vcustid INDI ind1, vcompany INDI ind2,+
         vaddress INDI ind3, vcity INDI ind4, vstate INDI ind5, +
         vzip INDI ind6, vphone INDI ind7
   ENDWHILE
   DROP CURSOR c1
   DROP CURSOR c2
```

Another small change also improves performance - instead of using SET VAR commands within the WHILE loops to concatenate city, state and zipcode together, and first and last name together, the concatenation operation can be done in the DECLARE CURSOR command. The concatenation in the DECLARE CURSOR reduces the number of commands that are repeated for each row and moves the work to the DECLARE CURSOR command.

```
   *(CUSTREP4.RMD add the concatenation to the DECLARE CURSOR)
   SET VAR vCheckCursor1 INTEGER = (CHKCUR('c1'))
   IF vCheckCursor1 = 1 THEN
     DROP CURSOR c1
   ENDIF
   SET VAR vCheckCursor2 INTEGER = (CHKCUR('c2'))
   IF vCheckCursor2 = 1 THEN
     DROP CURSOR c2
   ENDIF

   SET VAR vcustid INTEGER

   -- Replace SET VAR commands with expressions in
   -- the DECLARE CURSOR
   DECLARE c1 CURSOR FOR SELECT custid, company, +
   custaddress, (custcity + ',' & custstate & custzip), +
      custphone FROM customer ORDER BY custid
   DECLARE c2 CURSOR FOR SELECT (contfname & contlname) +
      FROM contact WHERE custid = .vcustid
   OPEN c1

   -- Retrieve and display the customer data
   FETCH c1 INTO vcustid INDI ind1, vcompany INDI ind2, +
      vaddress INDI ind3, vcsz INDI ind4, vphone INDI ind5
   WHILE SQLCODE <> 100 THEN
      WRITE .vcustid, .vcompany
      WRITE .vaddress
      WRITE .vcsz

      -- Retrieve and display the contact data
      OPEN c2
      FETCH c2 INTO vfullname INDI i1
      WHILE SQLCODE <> 100 THEN
         WRITE .vfullname
         FETCH c2 INTO vfullname INDI i1
      ENDWHILE

      -- Close cursor c2 and get the next row of
      -- customer data
      CLOSE c2
      FETCH c1 INTO vcustid INDI ind1, vcompany INDI ind2, +
         vaddress INDI ind3, vcsz INDI ind4, vphone INDI ind5
   ENDWHILE
```

```
DROP CURSOR c1
DROP CURSOR c2
```

The final change to improve performance is to use the RESET option on the OPEN c2 command instead of CLOSE c2. Overall, we have improved performance on this small set of rows by a full second. On a larger data set you can expect to see a greater performance improvement.

```
*(CUSTREP5.RMD)
--reset cursor 2 instead of close and open

SET VAR vCheckCursor1 INTEGER = (CHKCUR('c1'))
IF vCheckCursor1 = 1 THEN
  DROP CURSOR c1
ENDIF
SET VAR vCheckCursor2 INTEGER = (CHKCUR('c2'))
IF vCheckCursor2 = 1 THEN
  DROP CURSOR c2
ENDIF

SET VAR vcustid INTEGER
DECLARE c1 CURSOR FOR SELECT custid, company, +
   custaddress, (custcity + ',' & custstate & custzip), +
   custphone FROM customer ORDER BY custid
DECLARE c2 CURSOR FOR SELECT (contfname & contlname) +
   FROM contact WHERE custid = .vcustid
OPEN c1
FETCH c1 INTO vcustid INDI ind1, vcompany INDI ind2, +
   vaddress INDI ind3, vcsz INDI ind4, vphone INDI ind5
WHILE SQLCODE <> 100 THEN
   WRITE .vcustid, .vcompany
   WRITE .vaddress
   WRITE .vcsz

   -- Open cursor c2 with the RESET option,
   -- no CLOSE command is needed
   OPEN c2 RESET
   FETCH c2 INTO vfullname INDI i1
   WHILE SQLCODE <> 100 THEN
      WRITE .vfullname
      FETCH c2 INTO vfullname INDI i1
   ENDWHILE
   FETCH c1 INTO vcustid INDI ind1, vcompany INDI ind2, +
      vaddress INDI ind3, vcsz INDI ind4, vphone INDI ind5
ENDWHILE
DROP CURSOR c1
DROP CURSOR c2
```

As you can see from the above examples, maximizing the work of the DECLARE CURSOR command provides significant performance improvements. The changes were small and they didn't involve a lot of time or programming effort, but these changes did result in definite performance benefits.

Customize the environment
In addition to optimizing your programming code, you can improve cursor performance by optimizing the environment. Obviously, code runs faster on a newer computer. Outside of upgrading your hardware, however, certain R:BASE environment settings can be used to improve performance. These settings generally improve overall performance as well as cursor performance.

Look at the EXPLAIN.DAT output file generated by the MICRORIM_EXPLAIN variable to see the cursor query optimization. The OPEN command actually executes the query. Each query executed in your program puts an entry in EXPLAIN.DAT; for example, SELECT or UPDATE commands in the WHILE loop are reflected. You might also see a query reference to the SYS_RULES table, which is used for multi-user locking control.

By using EXPLAIN.DAT, you can easily see why using the RESET option on OPEN is faster. Normally, each OPEN redoes the query. When RESET is used, the query is only optimized once.

The EXPLAIN.DAT entries for the last two command files (CUSTREP4.RMD and CUSTREP5.RMD) from **Example 2** are shown here. The first entry shows nested cursors using the OPEN and CLOSE commands. The second entry shows using the RESET option on OPEN.

Cursor c1 on the Customer table is accessed sequentially, all rows in the table are retrieved, and no WHERE clause is used. If an indexed WHERE clause was used, EXPLAIN.DAT would show the index used. The second cursor on the Contact table does use an indexed WHERE clause to define the query. This query is redone each time the cursor is opened with a different vcustid value.

```
SortStrategy = DB_TAG  (internal=1)

SelectCost=1.  (OptimizationTime=0ms)
  Customer Sequential

SelectCost=2.904827e-002  (OptimizationTime=0ms)
  Contact (ColumnName=CustID,Type=F) Random Dup=1.296296 Adj=0.9714286

SelectCost=2.904827e-002  (OptimizationTime=0ms)
  Contact (ColumnName=CustID,Type=F) Random Dup=1.296296 Adj=0.9714286

SelectCost=2.904827e-002  (OptimizationTime=0ms)
  Contact (ColumnName=CustID,Type=F) Random Dup=1.296296 Adj=0.9714286
....

SelectCost=2.904827e-002  (OptimizationTime=0ms)
  Contact (ColumnName=CustID,Type=F) Random Dup=1.296296 Adj=0.9714286

SelectCost=2.904827e-002  (OptimizationTime=0ms)
  Contact (ColumnName=CustID,Type=F) Random Dup=1.296296 Adj=0.9714286

SelectCost=2.904827e-002  (OptimizationTime=0ms)
  Contact (ColumnName=CustID,Type=F) Random Dup=1.296296 Adj=0.9714286

SelectCost=1.  (OptimizationTime=0ms)
  SYS_RULES Sequential
```

The following EXPLAIN.DAT entry uses OPEN c2 RESET. The same query is used each time cursor c2 is accessed. The query does not need to be reoptimized each time the cursor is opened.

```
SortStrategy = DB_TAG  (internal=1)

SelectCost=1.  (OptimizationTime=0ms)
  Customer Sequential

SelectCost=2.904827e-002  (OptimizationTime=0ms)
  Contact (ColumnName=CustID,Type=F) Random Dup=1.296296 Adj=0.9714286

SelectCost=1.  (OptimizationTime=0ms)
  SYS_RULES Sequential
```

For additional information on using the MICRORIM_EXPLAIN variable to see the cursor query optimization, refer to the "Environment Optimization" chapter within the Reference Index of the R:BASE Help.

8.5.7 Questions & Answers

Q. When should I use a cursor?
A. Use a cursor when it seems like the best way to get a task done. There are no rules or standards to say when you should use a cursor and when you shouldn't. Often the logic behind a cursor is easier to understand than the logic behind a complex SELECT or UPDATE command that works across a group of rows. Many programmers have replaced DECLARE CURSOR routines with a single INSERT, UPDATE or DELETE command, most often for performance reasons, but not all cursors can be replaced with a single SQL command.

Deciding to use a cursor will depend on your level of programming expertise and understanding of the task to be accomplished. First get the program to work; once it works, look at ways to make the program run more efficiently and faster.

Q. How do I make a cursor faster?

A. Using a DECLARE CURSOR is slower than using just a single SQL command working across a group of rows, but some tasks just can't be done without using a cursor. You can use certain techniques to maximize the performance of DECLARE CURSOR routines. However, just like deciding when to use a cursor, there are no rules or standards about improving the performance of a cursor.

One of the best ways to make a cursor faster is to move as much of the work as possible into the DECLARE CURSOR command itself. Let the cursor select as many columns as possible. If you are doing calculations for each row, see if you can use one of the SELECT functions with the GROUP BY option.

For additional suggestions to improve cursor performance, see the Optimizing Cursors chapter in this document.

Q. Should I use WHERE CURRENT OF or an explicit WHERE clause?
A. In terms of performance, there is very little difference between the two options. Not all cursors can be used with the WHERE CURRENT OF syntax. Getting the most out of your DECLARE CURSOR statement is more important in terms of performance than making your cursor an updatable cursor.

Q. I'm trying to UPDATE data using WHERE CURRENT OF and I get a syntax error. I have checked and double checked the syntax, and it is fine.
A. You get this error when you have a non-updatable cursor. A non-updatable cursor does not support use of WHERE CURRENT OF. Use an explicit WHERE clause to update the table instead of WHERE CURRENT OF.

Q. What is a non-updatable cursor?
A. A cursor knows what data to retrieve based on the SELECT statement that is part of the DECLARE CURSOR command. Like a regular SELECT command, the SELECT that is part of the DECLARE CURSOR can retrieve data from multiple tables or use a GROUP BY. It has all the features of the regular SELECT. However, only a single table SELECT with no GROUP BY is updatable; this option is the only one that guarantees the cursor is pointing to a single row in a table. If the cursor can't point back to and identify a single row, it doesn't know what to update.

Q. Is it faster to retrieve data inside my WHILE loop using the SET VAR command or the SELECT...INTO command?
A. It's just a little bit faster to retrieve additional data using a SET VAR command instead of the SELECT...INTO command. The SELECT has more overhead. The fastest way to retrieve column data into variables, however, is to retrieve whatever columns possible through the DECLARE CURSOR command. That method can be almost twice as fast as using either SET VAR or SELECT...INTO.

Q. My WHILE loop never ends. It just keeps repeating the last row.
A. FETCH, which sets SQLCODE, should be the last command in the WHILE loop. When no more data is available, SQLCODE is set to 100. If FETCH is the last command in the WHILE loop, the next command executed is the WHILE statement, which tests the current value of SQLCODE. Other SQL commands placed after the FETCH and before the ENDWHILE might reset SQLCODE to a value other than 100.

Also, if your WHILE condition is not SQLCODE <> 100, make sure you are checking the condition correctly. If the WHILE loop doesn't exit, the WHILE condition is never false. Use TRACE and set up watch variables to see what is happening with your variable values.

Q. Why won't WHENEVER work with DECLARE CURSOR?

A. WHENEVER is an SQL error trap command that executes a GOTO whenever the data not found situation (SQLCODE = 100) occurs. At first glance, WHENEVER seems ideal for use with a DECLARE CURSOR. However, if your DECLARE CURSOR routine uses any other SQL commands that can return a "data not found" error, such as SELECT, INSERT or UPDATE, the WHENEVER immediately exits the DECLARE CURSOR WHILE loop even though all the data has not been processed. The R:BASE error "No rows exist or satisfy the WHERE clause" is a "data not found" error and sets SQLCODE to 100.

Q. I use DECLARE CURSOR to find out if a row exists in a table. Is there a way to do this check faster?
A. If you only want to see if a row exists in a table, don't use DECLARE CURSOR. The DECLARE CURSOR command by itself doesn't check this. You need to OPEN the cursor and FETCH before you know if a row has been found. Instead use the SELECT command; SELECT INTO a variable and test the variable value, or test SQLCODE immediately after the SELECT command. If no row is found, SQLCODE is set to 100. Using just the SELECT command is much faster than using the DECLARE CURSOR.

Q. My DECLARE CURSOR command is giving me a syntax error. Is there an easy way to check the syntax?
A. First make sure the cursor name is in the correct place in the command. A common error is to use DECLARE CURSOR c1 instead of DECLARE c1 CURSOR. The SELECT part of the DECLARE CURSOR command can get quite complex, particularly when more than one table is involved. Test the SELECT part of the DECLARE CURSOR command at the R> Prompt, which executes just like a regular SELECT command. You can test and debug the SELECT part of your DECLARE CURSOR before putting it into the DECLARE CURSOR structure.

8.6 Database Files

The Oterro database stores all information about a database in four files. Each file has the name of the database followed by an extension number of .RX1, .RX2, .RX3, or .RX4

The four database files operate together but have the capability to exist in separate directories, or on separate hard drives. They are not text files and cannot be modified directly by another program, text editor, or word processor without damaging the database.

As an example, the Oterro sample bluzvan database has these files:

- **BLUZVAN.RX1**
  File 1 contains the definition of the database structure.
- **BLUZVAN.RX2**
  File 2 contains the data in the database.
- **BLUZVAN.RX3**
  File 3 contains the indexes to the data stored in BLUZVAN.RX2.
- **BLUZVAN.RX4**
  File 4 contains the data in the database from the binary and character large object data types.

The Oterro database coordinates the four database files, which can exist in separate directories, by checking the timestamp recorded in each database file. When a database is created or opened, a timestamp is encoded in each of the four database files, enabling the database files to be stored in separate directories apart from one another. To ensure that the correct database files are accessed, the Oterro database evaluates the timestamp values of the files as it searches for them on the path statement. Oterro tries to connect to File 1 first (DBNAME.RX1), then it searches the path for the other database files (DBNAME.RX2, DBNAME.RX3, and DBNAME.RX4).

Updating the encoded timestamp values in the four database files (DBNAME.RX1, DBNAME.RX2, DBNAME.RX3, and DBNAME.RX4) so that they all have the same timestamp values might be necessary if the Oterro Engine is halted or if your computer or operating system crashes. When you try to connect to a database that is out of synchronization, the Oterro Engine returns an error message if the AUTOSYNC setting is off. Otherwise, the Oterro Engine synchronizes the database automatically.

File 1 (.RX1) contains:

- Current size of File 2 (.RX2), File 3 (.RX3), and File 4 (.RX4)
- Current timestamp for matching with File 2 (.RX2), File 3(.RX3) , and File 4 (.RX4)
- Encrypted database owner's user identifier
- Number of tables, columns, and indexes in the database

- Settings for all environment commands and special characters stored with the database, including formats for DATE, DATETIME, TIME, and CURRENCY
- Table structure including table name, row width, number of columns and rows in the table, and starting and ending rows for the table in File 2 (.RX2)
- Column structure including column name, data type, size, and pointer to the index in File 3 (.RX3)
- Index structure including index name, size, and table and column references

Data integrity depends on storing the types of information in the preceding list. Storing information about column structure, for example, ensures that you cannot redefine a column that exists in more than one table. Nor can you remove a column that is part of an expression for a computed column, preventing you from inadvertently changing or removing information. You also cannot define two columns with the same name and different data types or sizes.

The Oterro database updates File 1 every time you start an operation or enter a command that modifies database structure or change the settings that are stored with the database. The update occurs as soon as the Oterro database completes the command.

File 1 is also updated when the database is closed.

The Oterro database updates File 2 at the completion of every command that adds, modifies, or deletes rows from a database. In multi-user mode, when a command adds, modifies, or deletes multiple rows, the Oterro database updates the file after each row.

If SET CLEAR is set off, the Oterro database uses a 4K buffer as a workspace and writes information to File 2 only when the buffer is full. The default setting for SET CLEAR is on, which means the buffer is written to the file after every command.

8.7 Data Types

These data types can be specified within the R:BASE command syntax using the SET VARIABLE command.

BIGINT
- Holds a 64-bit integer value
- Offers a range of ±999,999,999,999,999,999
- Delimiters (such as commas) cannot be used in entry
- No length is needed

BIGNUM
- Holds decimal numbers whose precision and scale can be set
- When specifying BIGNUM, specify a precision (the total number of digits) from 1 to 38 (default 18) and a scale (the number of decimal places) from zero to any positive integer up to the precision value (default 0)
- R:BASE reserves a minimum of forty bytes of internal storage
- BIGNUM numbers are stored as DECIMAL

BIT
- Holds binary data
- The default length is 1 bit
- The fixed length is 1 to 1,500 bytes

BIT VARYING
Maps to VARBIT

BITNOTE
- Holds binary data
- No length is needed
- The variable length is 0 to 4,088 bytes of binary data

BOOLEAN
- Hold true/false values
- Internally stored as 0 for false and 1 for true
- Accepted values for false include: 0, false, 'false'

- Accepted values for true include: 1, true, 'true'

Note: For multilingual applications, the values of 0 and 1 are recommended.

BSTR
- Holds binary string data
- String data type that is used by COM (Component Object Model), Automation, and Interop functions
- Used to support Unicode in table data
- Composite data type that consists of a length prefix, a data string, and a terminator

Length Prefix
- Consists of a four-byte integer
- Occurs immediately before the first character of the data string
- Contains the number of bytes in the following data string
- Does not include the terminator

Data String
- Consists of a string of Unicode characters (wide or double-byte characters)
- May contain multiple embedded null characters

Terminator
- Consists of two null characters (0x00)

CHAR VARYING
Maps to VARCHAR

CHARACTER
Maps to TEXT

CURRENCY
- Holds monetary values of up to 23 digits represented in the currency format, established using SET CURRENCY
- Dollar amounts are in the range ±$99,999,999,999,999.99
- Commas or the current delimiter can be used. If no decimal point is included, .00 is assumed
- Data is stored as two long integer values, reserving four bytes of internal storage
- The negative currency format with parenthesis around the negative value e.g. ($500.00), is not recognized

DATE
- Holds dates in a 1- to 30-character format based on month, day, and year as established using SET DATE
- The minimum and default format to display month, day, and year is MM/DD/YY
- The allowable date range is January 1, 3999 BC to December 31, 9999 AD
- R:BASE reserves four bytes of internal storage

DATETIME
- A concatenation of the DATE and TIME data types, resulting in a sequence and display format as set by DATE and TIME
- DATETIME cannot be SET directly, but does permit extraction of its value by the DATETIME functions into a DATETIME variable
- For example: SET VAR vdatetime = (DATETIME(06/12/93, 12:15:30.123)), results in vdatetime = '06/12/93', 12:15:30.123'
- For identification purposes, DATETIME values are automatically stamped into R:BASE databases, also known as the *timestamp*
- DATETIME occupies 8 bytes of internal storage
- The time portion of the value does not have to be specified in a DATETIME data type. If omitted, it defaults to 0:0:0

DECIMAL
Maps to NUMERIC

DOUBLE
- Holds double-precision real numbers in the range ±1.7E308 with a precision of up to 15 digits
- DOUBLE numbers longer than 15 digits are stored as scientific notation
- R:BASE reserves eight bytes of internal storage

- Because DOUBLE numbers are stored in a binary form, the displayed value may not be the stored value
- Calculations are performed on the stored values
- Among numeric data types, DOUBLE provides the greatest range of values

GUID
- Binary global unique identifier to store unique values, which is represented as a 32-character hexadecimal string
- As the GUID data type is a binary value, it will increase retrieval of data from tables for indexed columns

An example of a GUID value is: **8C20005C-0E2A-47E0-B2BE-57E67961628B**

INTEGER
- Holds whole numbers in the range of ±1,999,999,999
- Delimiters (such as commas) cannot be used in entry
- R:BASE reserves four bytes of internal storage space

LONG VARBINARY
Maps to VARBIT

LONG VARBIT
Maps to VARBIT

LONG VARCHAR
Maps to VARCHAR

NOTE
- Holds alphanumeric data
- The default length is 0, where the length is determined by the data
- Holds variable length text of up to 4,092 characters
- Maximum length of a NOTE column can be set
- Indexes and constraints are allowed on NOTE data types
- R:BASE reserves a minimum of four bytes of internal storage space

NUMERIC
- Holds decimal numbers whose precision and scale can be set
- When specifying NUMERIC, specify a precision (the total number of digits) from 1 to 15 (default 9) and a scale (the number of decimal places) from zero to any positive integer up to the precision value (default 0)
- R:BASE reserves a minimum of eight bytes of internal storage
- NUMERIC numbers are stored as DOUBLE

REAL
- Holds real number amounts in the range of ±1E38 with six-digit accuracy
- Real numbers with up to seven digits are displayed as decimal numbers; for example, 321.414
- Real numbers with more than seven digits are represented in scientific notation; for example, 9.8E32
- R:BASE reserves four bytes of internal storage space
- REAL numbers are stored in a binary form; therefore, the displayed value may not be the actual stored value
- Calculations are performed on the stored values

SMALLINT
- Holds a 16-bit integer value
- Offers a range of ±32767
- Delimiters (such as commas) cannot be used in entry
- No length is needed

TEXT
- Holds alphanumeric data
- The default length is eight characters
- The maximum is 1,500 characters
- Maximum length of a TEXT column can be set
- R:BASE reserves a minimum of four bytes of internal storage space

TIME
- Holds time values in a 1- to 20-character format based on hours, minutes, and seconds, established using SET TIME
- The minimum format to display hours, minutes, and seconds is HH:MM:SS
- TIME can be specified up to thousandths of a second
- Time can be displayed or entered as a 12- or 24-hour clock
- R:BASE reserves four bytes of internal storage

VARBINARY
Maps to VARBIT

VARBIT
- Holds binary data
- No length is needed
- Is ideal for storing external files, like images, PDF files, etc.

VARCHAR
- Holds alphanumeric data
- No length is needed
- Is ideal for storing large text data

WIDENOTE
- Holds Unicode data
- The default length is 0, where the length is determined by the data
- Holds variable length text of up to 4,092 characters
- Maximum length of a WIDENOTE column can be set
- Indexes and constraints are allowed on WIDENOTE data types
- R:BASE reserves a minimum of four bytes of internal storage space, with 2 bytes per character

WIDETEXT
- Holds Unicode data
- The default length is eight characters
- The maximum is 1,500 characters
- Maximum length of a WIDETEXT column can be set
- R:BASE reserves a minimum of four bytes of internal storage space, with 2 bytes per character

8.8 Indexes

An index provides a pointer to the location of a column value in each row of a table, which allows R:BASE to search for rows of data much faster than searching rows sequentially. In general, R:BASE processes an operation that contains an indexed column faster than it processes one without.

The R:BASE index is similar to an index in a book; both indexes allow you to find information faster. Instead of searching through a book page by page, you can look up the topic in the index and find the exact page number of the topic. Similarly, you can apply indexes to columns so that R:BASE finds data faster.

When you apply an index to a column, R:BASE records the location of every value in that column. Then, when you look for or sort information in the column, R:BASE uses the index to find the rows you need quickly. For example, you want to list the bonuses that employee 102 earned; if the *empid* column in the *salesbonus* table is indexed, R:BASE finds and searches that column faster. Indexes are most useful when you have tables with many rows.

This means that by indexing the appropriate columns, you can speed up your applications. Processes that formerly took 20 minutes or more may be completed in only a few seconds. An index on an R:BASE column speeds up access to a row of data in much the same way that an index in a book speeds up access to a page.

To understand how indexes increase processing speed, it's helpful to know how R:BASE stores a database. An R:BASE database consists of four files; each file has a different file extension - RX1, RX2, RX3, and RX4 for R:BASE 11. File 1 contains the definition of the database structure; file 2 contains the data in the database; file 3 contains the indexes to the data stored in file 2, and file 4 contains the large binary object data (LOB). LOBs include graphic files or large text data. R:BASE also stores forms, reports, and labels as LOBs in File 4.

When you include an indexed column in a WHERE clause, R:BASE uses the index in file 3 to find the location of each row in the table in file 2. Without an indexed column, R:BASE must sequentially search each row in that table in file 2 to find the data. By sorting an index list, you give R:BASE nearly instantaneous access to the specific index reference to a column name.

An indexed column improves the performance of the following commands, clauses, or operations:

- CREATE VIEW
- DELETE DUPLICATES
- PROJECT
- RULES
- SELECT (when it includes a WHERE clause)
- WHERE...

8.8.1 Choosing the Columns to Index

It's a very good idea to put an index on key columns and on linking columns. A key column is a column that uniquely identifies rows. A linking, or common column, is a column that exists in two or more tables in order to establish a relationship between the tables. In effect, by choosing to link the common columns between two tables by adding a primary key and foreign key relationship, the columns are automatically indexed. You can also add a Unique key to a column, which is also automatically indexed. Primary keys, foreign keys, and unique keys are all types of constraints, which specifically control the data that enters your database by applying powerful data-integrity rules. By applying a constraint to a column, you can prevent irreconcilable and empty data from being entered add at the same time add an index.

Aside from constraints, an index can be added to a column for faster data retrieval in cases where the columns are not used in a primary key/foreign key relationship, and the column is likely to be used as part of a WHERE Clause whose values are at least moderately unique. In some cases, an index can be applied to columns that have RULES applied to them, which allows R:BASE to check the RULE faster. An index can also be added to the linking columns in views. An indexed column can contain null values, but R:BASE uses an index most efficiently if each row in the indexed column contains a value.

You can apply the following types of indexes:

- **Unique index** - Ensures that the values entered in the indexed column are unique

- **Full or partial text index** - For columns with NOTE or TEXT data types. R:BASE preserves each character in the indexed column (a full text index). Or, you can specify the number of characters to preserve, and R:BASE hashes (converts characters to a 4-byte integer) the remaining characters (a partial index).

  To keep the index file from becoming too large, use a partial index--specify enough characters to guarantee the values are unique. If the preserved values are not unique, R:BASE must unhash the values before it can identify the rows, which slows performance. If you do not specify the number of characters to preserve, R:BASE preserves all of them, unless there are more than 200 characters defined; then, R:BASE preserves the first 32 and hashes the rest.

- **Multi-column index** - A combination of up to 8 columns in one index. For example, if you consistently search three columns when working with a certain database, you can define a separate index for each column. Or, you can define one index for all three columns. R:BASE searches a multi-column index faster than three separate indexes. In a situation where you need to use more than one column to uniquely identify a row, try combining them into a computed column and then indexing the computed column.

Primary, unique, and foreign key columns are indexed automatically. In addition, the following types of columns are good for indexing:

- Columns that are neither primary nor foreign keys, but are frequently referred to in queries and sorts.
- Columns that have rules applied to them--in most cases, R:BASE can use indexing to check rules faster.

- Linking columns in views.

Although indexes speed up searches, they may slow down data entry for 2 reasons:

- they occupy space on the disk in the number 3 database file
- it takes time to build the index for each value as it is entered

Therefore, try to limit the use of indexes.

8.8.2 Assigning and Removing an Index

You can add an index to or remove an index from a column at any time by using either the Data Designer (RBDefine) or the CREATE INDEX or DROP INDEX commands. When you use the database-building menus to define a table, you can index columns as you define them. When you define a table using CREATE TABLE, you must add the index after you define the table.

Valid names must start with a letter, and can include the following characters:

- Letters (A-Z)
- Numbers (0-9)
- **#** (pound sign)
- **_** (underscore)
- **$** (dollar sign)
- **%** (percent sign)

To speed up some operations, remove the indexes from the columns before the operation and then rebuild the indexes after the operation finishes. For example, you can load a large number of rows into a table without RULES by using the LOAD and INSERT commands or by using GATEWAY to import data. First, you remove the indexes from the columns in the table, then load the records, and finally rebuild the indexes. This method speeds up processing because all of the data writes to file 2 occur at the same time, followed by all of the index writes to file 3.

When dropping and creating indexes constantly, it is recommended that you maintain the index file and perform a PACK on the indexes. Use the PACK KEYS in single-user mode with MULTI set to OFF, or PACK INDEX in a multi-user mode with MULTI set to ON.

8.8.3 Optimizing Indexes

You can create optimal indexes by paying attention to the data type of the indexed column and by minimizing the number of duplicate values in the column. The more frequently a particular value occurs, the less efficient that column becomes as an indexed column.

The fastest, most efficient data types for indexed access are INTEGER, REAL, DATE, TIME, and TEXT with a defined length of four characters or less.

You may find that indexing a table with 1500 or fewer rows actually increases the total time to search for rows of data, since the task also includes the time it takes to build the index.

8.8.4 Indexing Long TEXT Values

Because the defined length of TEXT values is often more than four characters, R:BASE must hash (convert) the TEXT values containing more than four characters to create a four-byte index called a hash value. Even though a TEXT column may contain unique text values, it may generate duplicate hash values.

How R:BASE Hashes TEXT Data

R:BASE attempts to create unique hash values by performing the following operations:

- Fills the first and second bytes of the four-byte hash value with the binary equivalents of the first and second characters in the TEXT string.

- Sums the binary values of all the odd-numbered characters in the TEXT string beginning with the third. Then it divides the sum by 256 and uses the remainder to fill the third byte of the hash value.
- Sums the binary values of all the even-numbered characters in the TEXT string beginning with the fourth. Then it divides the sum by 256 and uses the remainder to fill the fourth byte of the hash value.

If two hash values created by hashing two different TEXT strings are identical, R:BASE builds a multiple occurrence table in file 3 to store the duplicate hash values.

This is why TEXT strings that share only their beginning two characters (often the case with part numbers and with form, report, and label names) may have the same hash value, thereby reducing the effectiveness of the index. If you can ensure that the first two characters of each TEXT column value are unique, you can improve the performance of an index on that TEXT column, because you'll eliminate all of the multiple occurrence tables.

Another way to make hash values unique is to define indexed TEXT columns with as few characters as possible while remembering to make the column values unique, especially with regard to the first two characters.

IHASH to the Rescue

If the first two characters of TEXT values are identical, which is often true with part numbers, such as AB-100, AB-101, AB-102, and so on, try using the IHASH function and a computed column to reduce the likelihood of duplicates. It's important to understand that IHASH doesn't guarantee a unique hash value. IHASH may actually hash two different TEXT values to the same hash value (index). What IHASH does is to create more non-duplicate values, though it doesn't guarantee that all values will be unique.

How to Use IHASH

IHASH is a conversion function, not a command. Its syntax is simple:

```
(IHASH(arg,n))
```

Arg can be a TEXT column, a dotted variable, or a value. The width, n, tells R:BASE how many characters, starting with the first, are needed to establish a unique string. If n is zero, R:BASE uses the entire length of the string.

The first consideration in using IHASH is to decide whether to use it at all. For example, IHASH may not improve performance when used on a TEXT column that holds unique text names of four characters or less. But it will improve performance on a TEXT column that has many values with identical first and second characters. It all depends on whether IHASH can reduce the number of identical values in the index.

IHASH Step by Step

Here's an example that uses IHASH to speed up indexed access and joins by increasing the speed of a key, indexed TEXT column named *partid.* Follow these steps:

1. Define an INTEGER computed column, including the IHASH function:

   ```
   ALTER TABLE tblname ADD hashid = (IHASH(partid,0)) INTEGER
   ```

   If *partid* exists in other tables, you may want to add *hashid* to them too. For example, if *partid* links a parts table with an *invoice* table, use these commands to add *hashid*:

   ```
   ALTER TABLE parts ADD hashid = (IHASH(partid,0)) INTEGER
   ALTER TABLE invoice ADD hashid
   ```

2. Drop the index from *partid* and index the computed column *hashid*:

   ```
   DROP INDEX partid IN tblname
   CREATE INDEX ON tblname hashid
   ```

Index *hashid* in all the tables where it appears.

3. Use the newly indexed computed column in WHERE clauses and multi-table SELECT commands, but be sure to continue to include the actual TEXT column also. For example, use it to find a specific part number instead of using this WHERE clause:

   ```
   command ... WHERE partid = .vpartid
   ```

   Use this SET VAR and WHERE clause:

   ```
   SET VAR vhash = (IHASH(.vpartid,0))
   command ... WHERE (hashid = .vhash AND partid = .vpartid)
   ```

   You must use the dotted variable (*.vhash*) instead of directly using the expression in the WHERE clause, because R:BASE doesn't use an index if the WHERE clause uses an expression.

   You must include AND *partid* = . *vpartid* to ensure that you get the right value. Remember, there's no guarantee that the IHASH value will be unique.

   If you use the computed IHASH column (hashid) in a multi-table join, you must also remember to include both conditions in the WHERE clause. For example, the following rule checks for a unique value:

   ```
   RULES 'Value must be unique.' +
   FOR tblname SUCCEEDS +
   WHERE partid IS NOT NULL +
   AND NOT EXISTS +
   (SELECT partid FROM tblname t2 +
   WHERE t2.hashid = tblname.hashid +
   AND t2.partid = tblname.partid)
   ```

   Correlated sub-SELECTs force R:BASE to do an internal join using indexes.

   Here's another example of a multi-table join using *hashid*:

   ```
   SELECT t1.partid, SUM(t2.price) +
   FROM parts t1, invoices t2 +
   WHERE t2.hashid = t1.hashid +
   AND t2.partid = t1.partid +
   GROUP BY t1.partid
   ```

Speeding Up IHASH

When you use IHASH, try not to use zero as the width (n). Zero causes R:BASE to use the entire string. For example, if you're sure that the first 10 characters in a TEXT 50 column are enough to establish it as unique, use 10 rather than zero with IHASH.

8.8.5 Using WHERE Clauses with Indexes

A WHERE clause can be in a command, a rule, or a lookup expression. If a WHERE clause has only one condition, and if the conditional operator following the indexed column is =, BETWEEN, or IS NULL, R:BASE uses the index on the column.

The BETWEEN operator doesn't use the index on a data type that must be hashed. So if the operator is BETWEEN, the data type of the indexed column must be DATE, TIME, INTEGER, REAL, or TEXT with a defined length of 4 or less.

R:BASE doesn't use the index if the WHERE clause contains a wildcard character or an expression. To make R:BASE use the index, replace expressions with constants or dotted variables and get rid of the wildcards.

When a WHERE clause contains more than one condition and all conditions are combined with AND, the clause must have at least one indexed column that uses =, BETWEEN, or IS NULL if you want R:BASE to use indexes. Under these conditions, R:BASE chooses the condition that places the greatest restriction on the WHERE clause for the indexed search. R:BASE uses the first of two conditions when both are at the same level of restriction. Here are the three levels of restriction:

- = is most restrictive
- IS NULL is less restrictive
- BETWEEN is least restrictive

(R:BASE does not use indexes on BETWEEN when the command allows data modifications.)

8.8.6 Using ORDER BY with Indexes

You can significantly reduce the time R:BASE takes to process an ORDER BY clause when the column or columns listed in the ORDER BY clause are included in an index with the same column sort order as that specified in the ORDER BY clause.

An example would be if you are processing data by the invoice date and listing the results descending order, so the most current invoices appear first, you would have an index on the table's invoice date column that was created to process the data in the descending order as well.

8.8.7 Using Index-Only Retrieval

If it can, R:BASE will do an index-only retrieval. This is the fastest method of retrieving data. When the columns selected for display are limited to the column or columns in the index used in the WHERE clause, R:BASE will retrieve the data as it reads the index information from file 3. It does not need to look at the data stored in file 2. Index-only retrieval is done only when the columns to be retrieved are all included in the index. If the table is small, however, index-only retrieval may not be faster. As a rule of thumb, R:BASE will choose index-only retrieval if the length of the index columns is less than 50% ofthe row length (in bytes). If the table is small (not many columns), other retrieval methods are usually faster than index-only.

The SELECT COUNT(*) command uses index-only retrieval if there is an indexed column in the table. Instead, use a SELECT COUNT(colname) command, where the column specified in the command is not an indexed column, and the column has a value for every row. If the column is indexed, R:BASE will use index-only retrieval. If the column contains nulls, those rows are not counted. Do not use this command to check for broken pointers or to compare performance with other commands.

8.8.8 Indexing Computed Columns

Adding indexes can speed up your applications in several ways. One method that allows very quick data retrieval involves creating a computed column of unique values based on one or more columns.

For example, here is part of a program that helps in the scheduling of flight simulators for pilot training. A simulator code (*scode*) in combination with a date (*sdate*) uniquely identifies each row in the table. To make WHERE clauses fast, the following indexed computed INTEGER column (*sindex*) contains this expression:

```
sindex = (JDATE(sdate) + (scode * 100000)) INTEGER
```

The application prompts for a simulator code and date, which it puts in two variables: *vscode* (INTEGER) and *vsdate* (DATE). Then to quickly find the row, the application uses the following code (replace "command" with any command that uses a WHERE clause):

```
SET VAR vsindex = (JDATE(.vsdate) + (.vscode * 100000))
command ... WHERE sindex = .vsindex
```

The above code proved twice as fast as this slower alternative:

```
command ... WHERE scode = .vscode AND sdate = .vsdate
```

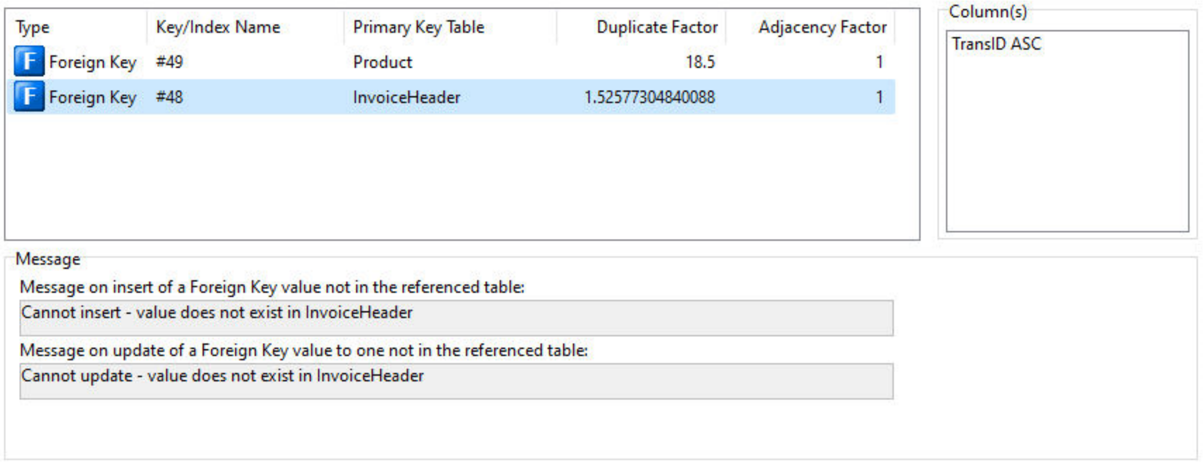
The more rows you have, the greater the efficiency.

8.8.9 Index Efficiency

To assist in determining the efficiency of indexes, check the **Duplicate Factor** and **Adjacency Factor** values within the Data Designer.

For Duplicate Factor, this is the average number of times each value appears in the column. The number 1 means that values are never duplicated, or unique. The "higher" the number, the less efficient the index.

For Adjacency Factor, this number is the estimate of the probability that two rows with similar index values will be physically located together in the #2 data file. A "higher" number means more efficient retrieval when reading rows in index order.



Type	Key/Index Name	Primary Key Table	Duplicate Factor	Adjacency Factor
Foreign Key	#49	Product	18.5	1
Foreign Key	#48	InvoiceHeader	1.52577304840088	1

Column(s)

TransID ASC

Message

Message on insert of a Foreign Key value not in the referenced table:

Cannot insert - value does not exist in InvoiceHeader

Message on update of a Foreign Key value to one not in the referenced table:

Cannot update - value does not exist in InvoiceHeader

Duplicate Factor is the computed duplicate factor, used by R:BASE during retrieval to guess the fastest way to find the result set. This number is the average number of times each value appears in the column. A number of 1.0 means that values are never duplicated, or always unique. Zero means that the value is unknown. The higher the number, the less efficient the index.

Adjacency Factor is the computed adjacency factor, used by R:BASE during retrieval to guess the fastest way to find the result set. It is the estimate of the probability that two rows with similar index values will be physically located together in the .RB2 file. A higher number means more efficient retrieval when reading rows in index order. Zero means that the value is unknown.

The values within these two columns are computed when indexes are rebuilt or reloaded, for example, during a PACK or RELOAD.

It is recommended that you reconsider the use of indexes with a high Duplicate Factor value. But, there is not an exact Duplicate Factor benchmark for which to warrant its value "too high". It's value is calculated based upon the number of duplicate values that are within the column, which would be directly related to how many rows are in the table.

A Duplicate Factor with a value of 500 within a table containing 750 rows would be an example of a poorly implemented index. On the other hand, a Duplicate Factor of 500 within a table containing 785,000 rows would be considered productive. The Duplicate Factor value is used by R:BASE to guess the fastest way to find your results. The recommendation to reconsider the use of indexes with a high Duplicate Factor value is just that; a recommendation. It is up to the developer to decide if they are truly taking advantage of the index, or hurting their system's performance.

8.8.10 Smart Indexing

The single most important factor in determining the effectiveness of an index is the uniqueness of index values. A unique index value is found faster than a value with multiple index occurrences. You can have multiple occurrences of index values if you have more than one row with the same data value in a column, or if you have a computed column (such as IHASH) whose values share the same result, or if the index is hashed.

Note that we are talking about unique index values, not data values. An index value may or may not be the same as a data value. Long text columns that are hashed when indexed have a higher probability of unique data values creating non-unique index values. An index loses its effectiveness as duplicate values increase and R:BASE must make more reads and comparisons.

Indexed columns also affect performance when adding or changing data in a table. Indexed columns must be updated when a row is added or when the index column value is changed. The number of indexed columns in a table affects the speed with which rows are added or changed in the table. Take this into account when defining indexes and constraints. It is faster to load and change data on a table with one indexed column than on a table with seven indexed columns.

8.8.11 Summary

Using indexes effectively requires understanding how they work. This means deciding which columns to index, which data types to use, and whether to use IHASH. You also need to know how to improve performance when using a text index.

When adding an index to a column, consider the following criteria:

- Columns that are neither primary nor foreign keys, but are frequently referred to in queries and sorts
- Columns that have rules applied to them
- Linking columns in views
- The index is more efficient if each row contains a value
- The index is more efficient based on the uniqueness of the data

When properly implemented, indexes can greatly improve the data retrieval performance of your applications!

8.9 Information Management with R:BASE

R:BASE is a relational database product that lets you design sets of tables to store and retrieve your data easily. Each table contains information arranged in columns and rows about a single object or event, such as a customer list or a list of sales transactions. A column is a specific fact, such as a customer's name, about the object or event; a row is a cross-section of columns that is unique for a particular instance in the table.

An R:BASE database is a collection of tables. For each column in a table, you specify a data type that tells R:BASE what kind of data the column will hold - such as dates, currency, or text. You can also have columns that hold a value computed from other columns. For example, if you have sales stored in one column and a tax rate in another, a computed column can hold the tax on each sale (sales * tax rate).

The power of R:BASE lies in the control it gives you over the tables you create. You can manipulate the data from one or any combination of the tables in your database, and you can organize tables so that you rarely need to enter a particular piece of information more than once.

With R:BASE you can combine information from the tables in a database to provide answers for your questions and create another permanent table that stores the combined information. You can also create views, which are temporary tables that display all or part of the current contents of one or more tables. A view does not require you to store the data in more than one table -it always shows you the most up-to-date information, and it can combine data from several tables. A view is like a television screen that is split to show two camera angles at once.

8.10 International Characters

The R:BASE configuration file contains tables that define how R:BASE processes and prints characters. If you want to change how R:BASE evaluates characters, you can modify the information in these tables, which are described below.

R:BASE saves table configurations with the database. If you make modifications to these tables in the configuration file, use the PACK command with the WITH USER CASE option to compress the database and apply the new configuration.

The Case Folding Table

The Case Folding table establishes the correspondences between uppercase and lowercase characters, such as "A" and "a." R:BASE uses this table when testing characters for equality when the CASE setting is off.

Each line in this table starts with "CASEP" and is followed by two ASCII character codes corresponding to the uppercase and lowercase characters. For example, the following line shows that "a" (ASCII code 97) corresponds to "A" (ASCII code 65):

```
CASEP 97 65
```

The Collating Table

The Collating table equates two characters in sorting and inequality testing (>, >=, <, and <=).

Each line in this table begins with "COLLATE" and is followed by two ASCII character codes, whose corresponding characters are considered equal in a sorting sequence. For example, the following lines indicate that "a" (ASCII code 97), "ä" (ASCII code 228), and "A" (ASCII code 65) are all equal in a sorting order:

```
COLLATE 97 65

COLLATE 228 65
```

The Printer Table (DOS Only)

The Printer table tells the printer how to print certain characters. Some printers cannot print certain international characters, such as characters with accents or umlauts, so the printer must combine two or more characters to create the international character.

Each line in this table begins with "FOLD" and is followed by a character and its ASCII code, then one or more ASCII codes whose corresponding characters must be combined to create the first character. For example, the following line tells the printer how to print "à" (ASCII code 133); print "a" (ASCII code 97), backspace (BS), then print an accent "`" (ASCII code 96):

```
FOLD à 133 97 BS 96
```

If your printer can print a character without combining other characters, do not delete the line. Instead, edit the line. After "FOLD," enter the character, the character's ASCII code two times, then "00 00." For example, if your printer can print "à," edit the line as follows:

```
FOLD à 133 133 00 00
```

The Expansion Character Table

The Expansion Character table equates one character to two other characters. For example, you can equate "ß" to "SS." This table is used in tables, columns, variables, WHERE clauses, ORDER BY clauses, IF and WHILE commands, and indexed and non-indexed columns.

Each line in this table begins with "EXPAND" followed by three ASCII character codes. You can have up to seven lines in this table. For example, the following line equates "ö" (ASCII code 246) to "oe" (ASCII codes 111 and 101):

```
EXPAND 246 111 101
```

The Character Folding Table

The Character Folding table equates uppercase characters to lowercase characters. This table is used in string-manipulation functions.

Each line in this table begins with "LCFOLD" and is followed by two ASCII character codes. For example, the following line equates "A" (ASCII code 65) to "a" (ASCII code 97):

```
LCFOLD 65 97
```

The Case-Sensitive Collating Table

The Case-Sensitive Collating table lists characters and their position in the sequence order of all characters. This table is used when the CASE setting is on and when building indexes for columns with the TEXT data type.

Each line in this table begins with "COLLATEC" and is followed by an ASCII character code and its sequence position. For example, the following line indicates that "B" (ASCII code 66) is in the 76th position in the sequence order:

```
COLLATEC 66 76
```

If a character is not in this list, its sequence position number is the same as its ASCII character code.

8.11 Multi-User Considerations

R:BASE is a multi-user system that can be used on a single-user computer. In a multi-user environment, you must ensure that the databases to be shared are located properly, that R:BASE is prepared to protect your data, and that all workstations are properly configured to share databases.

In addition to the security systems that most networks provide, the R:BASE command GRANT, with which access rights are assigned, provides additional security for R:BASE databases.

Multi-User Mode Topics:

Introduction to Using R:BASE on a Network
Setting Up for Network Use
Sharing Network Resources
Setting the Multi-User Default
Concurrency Control
Resource Waiting
Schema Reading Mode with SET STATICDB
Locks

8.11.1 Introduction to Using R:BASE on a Network

R:BASE in a multi-user environment allows multiple users to simultaneously access, view, update, insert, and delete data. To ensure data integrity, R:BASE uses various types of locks and a waiting period. To most efficiently use R:BASE on a network, avoid situations that cause R:BASE to work more at preventing user conflicts than at storing, sorting, and retrieving data. To a large degree, the speed of a network and the network card in your computer dictates how fast R:BASE can perform. The faster the network and network card, the better the R:BASE performance.

Although the techniques for making R:BASE perform most efficiently depend on your particular database and network, follow these guidelines to limit resource locking and improve response time:

- Schedule data entry among users at different times of the day, or have users enter data into temporary tables that are inserted into a master table at the end of the day.
- When designing a database, take advantage of the relational capabilities in R:BASE for linking and manipulating data stored in several tables. When possible, store data in several small tables rather than in one large one.
- Design applications so that each user requires the fewest shared resources to do his or her work.
- Avoid using commands that lock a database when other users need access to it. For example, avoid commands that change the structure of the database. You can run updates and backups at night, when an entire database is available without conflicts.

Multi-User Mode Topics:

Setting Up for Network Use
Concurrency Control
Sharing Network Resources
Setting the Multi-User Default
Resource Waiting
Schema Reading Mode with SET STATICDB
Locks

8.11.2 Other Multi-User Considerations

Only a limited number of users can access certain parts of R:BASE at the same time. Be sure that all network users are aware of the limitations and plan ahead to avoid conflicts in accessing database and application files. A user should operate in single-user mode when performing any operation that modifies the structure of the database or using the PACK command.

The limitations to access for users are listed in the following table:

R:BASE Commands	Operating System Files Used	Access Limitations
CODELOCK	From ASCII to converted binary file	One user at a time can convert a given ASCII file
FORMS	Database files	One user at a time can modify a form in a given database.
REPORTS	Database files	One user at a time can modify a report in a given database.
GATEWAY	Database files	Data files for transfer to database, database files. One user at a time can use data files.
RBDEFINE	Database files	One user at a time can create or change database structure.
RBEDIT	ASCII file	One user at a time can create or edit a file.
LAUNCH	External programs	All users can share the program, within the limitations of the external program.
ZIP	External programs	All users can share the program, within the limitations of the external program.

Locks Topics:

Using the SET ROWLOCKS Command to Lock Rows
Using the SET VERIFY Command to Verify Data Entry
Using SET LOCK to Set Exclusive Table Locks
Displaying Multi-User Locks
Clearing Buffers with the SET CLEAR Command
Other Multi-User Considerations

Multi-User Mode Topics:

Introduction to Using R:BASE on a Network
Setting Up for Network Use

Sharing Network Resources
Setting the Multi-User Default
Concurrency Control
Resource Waiting
Schema Reading Mode with SET STATICDB
Locks

How to...

Operate in single-user mode

8.11.3 Clearing Buffers with the SET CLEAR Command

Setting CLEAR off gives you the option of waiting for R:BASE to fill the memory buffer allocated for updates before writing data to the disk. However, in multi-user mode, R:BASE disables the command and immediately updates the disk to prevent multi-user conflicts. If you have the SET CLEAR OFF command in a program file, you might want to remove it to prevent the display of an error message, though the presence of the command does not affect the running of the program.

Locks Topics:

Using the SET ROWLOCKS Command to Lock Rows
Using the SET VERIFY Command to Verify Data Entry
Using SET LOCK to Set Exclusive Table Locks
Displaying Multi-User Locks
Other Multi-User Considerations

Multi-User Mode Topics:

Introduction to Using R:BASE on a Network
Setting Up for Network Use
Sharing Network Resources
Setting the Multi-User Default
Concurrency Control
Resource Waiting
Schema Reading Mode with SET STATICDB
Locks

8.11.4 Managing Scratch Files ($$$)

Did you know that you can automate the default directory for the R:BASE/OTERRO scratch files without knowing the user's rights to their temporary directory?

SET SCRATCH sets the drive and directory location for temporary files created when sorting data.

`SET SCRATCH ON` - stores temporary files on the database drive and directory
`SET SCRATCH OFF` - stores temporary files on the current drive and directory
`SET SCRATCH <path>` - provides the path to the location where temporary files are stored
`SET SCRATCH TMP -` stores temporary files on the user's local temporary file directory by reading the user's TMP operating system environment setting

Using the latest builds of R:BASE/OTERRO, you need to either update the line for "SCRATCH TMP" to automatically define the windows temporary environment or use "SCRATCH C:\TEMP", for example, in R:BASE/OTERRO configuration files. If you define the "SCRATCH C:\TEMP", make sure that you actually have the C:\TEMP directory. We suggest that you keep all configuration (*.CFG) file in Windows or WinNT directory.

This option is also helpful when you establish a universal naming convention (UNC) network connection for an R:BASE database as a System DSN defined with UNC, such as "\\FileServerName\SharedDirectoryName\dbname.rx1". This type of environment requires you to sure that you have also set the path for R:BASE and OTERRO temporary SCRATCH files.

Edit the configuration file to read SCRATCH settings as follows:

```
SCRATCH TMP
```

This will help eliminate all issues related to the access rights, disk space and so on, when running R:BASE/OTERRO on enterprise servers and when accessing R:BASE database defined as a System DSN using the UNC option.

Multi-User Mode Topics:

Introduction to Using R:BASE on a Network
Setting Up for Network Use
Sharing Network Resources
Setting the Multi-User Default
Concurrency Control
Resource Waiting
Schema Reading Mode with SET STATICDB
Locks

8.11.5 Displaying Multi-User Locks

When R:BASE is used on a network, the LIST command displays the names of any locked tables by highlighting the table name.

The LIST TABLE *tblname* command tells you whether the lock is an edit, cursor, row, local, or remote lock, as shown in the following table:

Lock	Description
Row lock	Another workstation is using EDIT ALL or a form that accesses this table or other commands that use row locks
Local lock	The SET LOCK command was issued from this workstation
Remote lock	A table lock has been applied with a command issued at another workstation
Cursor lock	A cursor is open on the table

Locks Topics:

Using the SET ROWLOCKS Command to Lock Rows
Using the SET VERIFY Command to Verify Data Entry
Using SET LOCK to Set Exclusive Table Locks
Clearing Buffers with the SET CLEAR Command
Other Multi-User Considerations

Multi-User Mode Topics:

Introduction to Using R:BASE on a Network
Setting Up for Network Use
Sharing Network Resources
Setting the Multi-User Default
Concurrency Control
Resource Waiting
Schema Reading Mode with SET STATICDB
Locks

8.11.6 Using SET LOCK to Set Exclusive Table Locks

The SET LOCK command sets locks on tables. SET Lock should be set on when a user wants to be certain that no other user will alter data in the tables being updated during a procedure. This command is useful in conjunction with the DECLARE CURSOR command. If R:BASE cannot lock all tables listed in the command, it does not lock any of the tables listed in the command.

Exclusive table locks are cumulative--that is, for each SET LOCK *tblname* on command you issue, you must issue a corresponding SET LOCK *tblname* off command to remove the lock from that table. Also, the user who locked the table must issue the SET LOCK *tblname* off command.

The SET LOCK command is typically used in an application program to set locks, allow a procedure to be performed, then remove locks.

Automatic locking is in effect even if the SET LOCK command is issued. Setting locks off affects only locks set by the SET LOCK *tblname* ON command--not locks that R:BASE sets automatically.

Locks Topics:

Using the SET ROWLOCKS Command to Lock Rows
Using the SET VERIFY Command to Verify Data Entry
Displaying Multi-User Locks
Clearing Buffers with the SET CLEAR Command
Other Multi-User Considerations

Multi-User Mode Topics:

Introduction to Using R:BASE on a Network
Setting Up for Network Use
Sharing Network Resources
Setting the Multi-User Default
Concurrency Control
Resource Waiting
Schema Reading Mode with SET STATICDB
Locks

8.11.7 Using the SET VERIFY Command to Verify Data Entry

The R:BASE default for concurrency control is COLUMN checking. That is, R:BASE does not alert a user to a possible conflict unless two users attempt to update the same column in the same row. The SET VERIFY ROW command checks all values in a row for possible conflicts.

Locks Topics:

Using the SET ROWLOCKS Command to Lock Rows
Using SET LOCK to Set Exclusive Table Locks
Displaying Multi-User Locks
Clearing Buffers with the SET CLEAR Command
Other Multi-User Considerations

Multi-User Mode Topics:

Introduction to Using R:BASE on a Network
Setting Up for Network Use
Sharing Network Resources
Setting the Multi-User Default
Concurrency Control
Resource Waiting
Schema Reading Mode with SET STATICDB
Locks

8.11.8 Using the SET ROWLOCKS Command to Lock Rows

When R:BASE is running in multi-user mode, the user has the added capability of forcing R:BASE to use row-level locking on some R:BASE commands. By default, the SET ROWLOCKS command is set on. You can set ROWLOCKS by choosing **Utilities: Settings...** then click the **Multi-User** button, or using the SET ROWLOCKS command at the R> Prompt. Setting ROWLOCKS off is not recommended if all users intend to update the same tables.

R:BASE always uses row-level locking for the EDIT, EDIT USING, and ENTER commands.

Locks Topics:

Multi-User Mode Topics:

8.11.9 Setting Up for Network Use

To set R:BASE up in a network environment, you need to follow several simple steps:

- Verify how many seats that you are licensed to install R:BASE on.
- Delete ALL old configuration files on ALL local and network drives.
- Install R:BASE locally, to all the workstations that you are licensed for, and will be using.
- Place ALL database and application files on a shared network drive that is accessible from all the workstations.
- Create either a startup file, or a Desktop Shortcut Icon to connect to your database and run your applications.

The following topics provide information you need to know after you have installed R:BASE on a multi-user system.

Multi-User Mode Topics:

8.11.10 Sharing Network Resources

When R:BASE is on a local area network, users can share R:BASE program files, printers, hard disks, directories and databases. You can also have R:BASE program files on a local drive or workstation; however those files will only be available to the user of that particular machine.

R:BASE on a network allows multiple users to simultaneously create and share new and existing databases, add, change, and print data, as if each user were the only one using that database. To ensure data integrity, R:BASE includes automatic concurrency control and a locking system that eliminates conflicts, such as two users attempting to change the same data at the same time. R:BASE has commands that allow a user to manually set the locks and wait periods that prevent conflicts.

Whether an R:BASE program or database files are stored on a network server or local hard drive, the limit on the number of seats using R:BASE remains in effect. Also, only a limited number of users can access some parts of R:BASE at the same time. For example, although users can share program files, only one user at a time can import data to a given database with the Import/Export utility, create or alter the structure of a database, or create, edit, or convert a given application file.

R:BASE can use most printers that are compatible with your network operating system. In some cases, you need only attach the printer to print from R:BASE. In other cases, you must identify the printer to the network operating system and initialize the print spooler.

Multi-User Mode Topics:

Introduction to Using R:BASE on a Network
Setting Up for Network Use
Concurrency Control
Setting the Multi-User Default
Resource Waiting
Schema Reading Mode with SET STATICDB
Locks

8.11.11 Setting the Multi-User Default

To set R:BASE so that it starts in multi-user mode by default, edit the R:BASE configuration file to include the command SET MULTI ON. This file may contain the user's name, but does not need to be unique to all workstations.

Multi-User Mode Topics:

Introduction to Using R:BASE on a Network
Setting Up for Network Use
Sharing Network Resources
Concurrency Control
Resource Waiting
Schema Reading Mode with SET STATICDB
Locks

8.11.12 Concurrency Control

On a network, R:BASE automatically performs concurrency control, locking, and resource waiting to prevent two users from modifying the same data at the same time.

Concurrency control is how R:BASE controls multiple access to tables when users are using forms or the EDIT command. Concurrency control, row, table and database locks only affect operations in which users are storing or altering data already stored on a disk.

R:BASE allows as much access to database resources as possible and keeps the duration of locking as brief as possible. However, the extent to which users are locked out of resources depends on the operations that cause R:BASE to issue the locks.

Multi-User Mode Topics:

Introduction to Using R:BASE on a Network
Setting Up for Network Use
Sharing Network Resources
Setting the Multi-User Default
Resource Waiting
Schema Reading Mode with SET STATICDB
Locks

8.11.13 Resource Waiting

When a command has been locked out of a resource--database, table, column--R:BASE checks to see if the requested resource is available while in the waiting period. If the resource becomes available, the user can proceed. If not, R:BASE prompts the user to either terminate the request or continue waiting. While the user waits, R:BASE displays a message indicating the approximate percentage of wait time remaining.

Multi-User Mode Topics:

Introduction to Using R:BASE on a Network
Setting Up for Network Use
Sharing Network Resources
Setting the Multi-User Default
Concurrency Control
Schema Reading Mode with SET STATICDB

Locks
SET INTERVAL
SET WAIT

8.11.13.1 Schema Reading Mode with SET STATICDB

In a multi-user environment, R:BASE must be aware of any schema modifications made by a user. Being aware of schema modifications, which is necessary for database integrity, entails constantly re-reading schema information.

Because many data-processing operations do not make frequent schema changes to the database, such a re-reading unnecessarily slows down processing.

The command SET STATICDB ON instructs R:BASE to prevent any schema modifications, therefore, R:BASE does not read schema information. By default, STATICDB is set off. SET STATICDB OFF forces a re-read of schema information before running any command.

When connecting to a database with STATICDB set on, R:BASE displays "Database Schema is Read-Only." When STATICDB is set on, the Data Designer and the "New" and "Design" buttons for Tables and Views in the Database Explorer are inactive. Also, the following R:BASE commands are not allowed when STATICDB is set on.

ALTER TABLE*	CREATE INDEX	
ATTACH	DETACH	
COMMENT ON*	DROP*	
* Access not allowed to tables created prior to database connection.		

If a user has STATICDB set off, that user cannot connect to a database being used by a user who has STATICDB set on. All users accessing the same database must have matching STATICDB settings.

When STATICDB is set on, only temporary tables and views can be created; these are deleted when disconnecting from the database.

The BACKUP ALL and UNLOAD ALL commands do not act upon temporary tables. You can, however, backup individual temporary tables with the BACKUP command. Similarly, UNLOAD permits copying of individual temporary tables to the selected output device.

You can find out what the current STATICDB setting is with the SHOW STATICDB command. When STATICDB is set on, the LIST TABLE command will display the temporary table and view names dimmed. Also, you can capture the current STATICDB setting as a value with CVAL, for example:

```
SET VAR vcval = (CVAL('STATICDB'))
```

Multi-User Mode Topics:

Introduction to Using R:BASE on a Network
Setting Up for Network Use
Sharing Network Resources
Setting the Multi-User Default
Concurrency Control
Resource Waiting
Locks

8.11.14 Types of Locks

The Multi-User Concurrency Control and Locking Table describes the automatic concurrency control and locks used in multi-user mode and the commands that initiate them. Operations that only view data or the database structure are not affected by concurrency control or locking.

The Accessing Tables and Databases Table shows what happens when different commands try to access a table or database already being used by another command. The columns represent the type of control or lock already placed on the table or database. Each row has a heading with the name of the control or lock needed by the command trying to gain access to a table or database.

Locks Topics:

Using the SET ROWLOCKS Command to Lock Rows
Using the SET VERIFY Command to Verify Data Entry
Using SET LOCK to Set Exclusive Table Locks
Displaying Multi-User Locks
Clearing Buffers with the SET CLEAR Command
Other Multi-User Considerations

Multi-User Mode Topics:

Introduction to Using R:BASE on a Network
Setting Up for Network Use
Sharing Network Resources
Setting the Multi-User Default
Concurrency Control
Resource Waiting
Schema Reading Mode with SET STATICDB

8.11.14.1 Multi-User Concurrency Control and Locking

The following table displays multi-user concurrency control and locking used for several R:BASE commands.

Type of Lock	**Command**	**Description**
Concurrency Control	EDIT EDIT USING ENTER	Prevents one user from accidentally overwriting another user's changes. These commands can access the same table simultaneously.
Row Lock	DELETE ROWS ENTER *form* INSERT *values* LOAD FROM *filespec* UPDATE	When SET ROWLOCKS is on, a lock is only applied to a row. When off, a table lock is in effect.
Table Lock	BACKUP* DROP RULES FORMS GRANT INSERT *subselect* RBLABELS LOAD *from filespec* REPORTS RESTORE REVOKE RULES SET LOCK ON UNLOAD*	Must wait for commands that obtain table and database locks. Once obtained, this lock excludes all other concurrency control and locking until these commands have finished.
Full Database Lock	BACKUP ALL RELOAD UNLOAD ALL	All tables in the database are locked.
Database Schema Lock	ALTER TABLE CREATE SCHEMA CREATE TABLE CREATE VIEW CREATE INDEX DROP INDEX DROP COLUMN DROP TABLE DROP VIEW PROJECT RENAME	Engages a full database lock preventing schema modifications, then releases the schema lock remaining in table lock.
Cursor Lock	OPEN *cursorname*	Stops database schema commands but allows table locks; acts as a table lock.

*A table lock is placed only if one table is unloaded. A database lock is placed if more than one table is unloaded.

8.11.14.2 Accessing Tables and Databases

The following table displays the access available to tables and databases during locks.

	Concurrency Control	Cursor Lock	Row Lock	Table Lock	Database Lock	Schema Lock
Cursor Lock	Access	Access	Access	Wait	Quit	Wait
Row Lock	Access	Access	Access	Wait	Quit	Wait
Concurrency Control	Access	Access	Access	Wait	Quit	Wait
Table Lock	Wait	Wait	Wait	Wait	Quit	Wait
Database Lock	Wait	Wait	Wait	Wait	Quit	Wait
Schema Lock	Wait	Wait	Wait	Wait	Quit	Wait

8.11.15 Effects of the SET INTERVAL Command

The SET INTERVAL command specifies the amount of time that is to elapse before R:BASE retries the command that caused a conflict. This period is a maximum of nine tenths of a second, with the default of five tenths.

Multi-User Mode Topics:

Introduction to Using R:BASE on a Network
Setting Up for Network Use
Sharing Network Resources
Setting the Multi-User Default
Concurrency Control
Resource Waiting
Schema Reading Mode with SET STATICDB
Locks

8.11.16 Effects of the SET WAIT Command

The default waiting period is four seconds. You can change the waiting period by choosing **Settings > Configuration Settings**, or at the R> Prompt with the SET WAIT command.

The SET WAIT command specifies the period in seconds in which R:BASE continues to retry the last command before stopping processing. Rather than stopping processing as soon as it encounters a lock, R:BASE retries some commands. For more information about how R:BASE handles lock conflicts, see the Accessing Tables and Databases Table.

If you set the wait period too long, you can have your users sitting unproductively for long periods. Be sure you set manual table locks (discussed later) only when absolutely necessary. If users are running programs that lock tables, set resource wait periods reasonably short to avoid users leaving workstations for a long without exiting the database.

A wait period can help prevent a deadlock. A deadlock occurs if a user locks a table and then waits for a table previously locked by a second user, who in turn is waiting for the table locked by the first user.

R:BASE prevents deadlocks because a command either accesses a resource or eventually stops trying to obtain access. If you issued a command from the R> Prompt, R:BASE asks you when the wait period expires whether to keep waiting or cancel the command. If the command was issued from a command file, R:BASE retries the command until the wait period expires and then proceeds to the next command.

Multi-User Mode Topics:

Introduction to Using R:BASE on a Network
Setting Up for Network Use
Sharing Network Resources
Setting the Multi-User Default
Concurrency Control
Resource Waiting
Schema Reading Mode with SET STATICDB
Locks
SET INTERVAL

8.11.17 Single-User Mode

To operate in single-user mode:

1. Have all users disconnect from the database.
2. Enter SET MULTI OFF at the R> Prompt.
3. Reconnect one user to the database.

SET CLEAR

If you do a process in a single-user environment, you will improve performance by using SET CLEAR OFF before executing commands that change or add rows to the database. SET CLEAR OFF sets up a 5K buffer to hold changes. Changes are written to disk when you SET CLEAR ON, when the buffer is full or is needed for the next page of data, or when you disconnect the database or exit from R:BASE. CLEAR can be set differently by individual workstations and is ignored in multi-user mode.

8.12 Purpose of a Rule

A data entry rule ensures that the data entered into a column meets the criteria you specify. You can use rules to do the following:

- Prevent duplicate information from being entered. For example, a new stock number cannot be the same as an existing one.
- Verify that the data being entered corresponds with data elsewhere in the database. For example, the product stock number must exist in a table before you enter a sales transaction for the product.
- Prevent a row from being deleted if it corresponds with data elsewhere in the database. For example, if a transaction refers to a customer, the customer cannot be deleted from the *customer* table.
- Define a value range. For example, when you enter a salary, it must be between $15,000 and $50,000. You can either specify a maximum value, a minimum value, or a range of values.

When you enter or edit data, R:BASE checks the data you enter with the corresponding rules. If the conditions of a particular rule are not met, R:BASE displays the error message defined for that rule and does not add the row. You can turn the rules setting on or off; the RULES setting must be on for R:BASE to check them.

As you use and modify the tables in your database, you might need to revise the data entry rules. For example, if you rename a column or table used in the WHERE clause of a rule definition, you must update the rule to reflect the change.

You can delete rules that no longer apply to your database. For example, delete rules that are associated with a column you have deleted from a table.

See also:

DROP Command
RENAME Command
RULES Command
SET RULES Command

8.13 Reserved Words

You must review all table and column names to be sure that no invalid names are defined and no reserved words are used.

In the current version, a 1-128 character alpha-numeric name must be specified for a column. Spaces are NOT permitted. Valid names must start with an alpha character and can include the following symbols:

- Letters (A-Z)
- Numbers (0-9)
- _ (underscore)
- # (pound sign) *
- $ (dollar sign) *
- % (percent sign) *

  ***Note:** For ODBC compliance, it is NOT recommended to use the #, $, or % even though R:BASE permits it.

Review your table and column names that may begin with a number. Check for any characters that are not supported like a question mark "?" or a greater than character ">".

Also review your table and column names to verify that no reserved words are used. These words, commands, keywords, and other names are to be used exclusively by the R:BASE database. Names used by System Tables, System Views and System Columns are also reserved words. As a rule, all words beginning with "**SYS**" are reserved.

Do not use reserved words or any shorter forms of them as names for columns, tables, or views. As a rule, if the word is a reserved word, R:BASE will flag it. R:BASE will not allow you to use a reserved word as a column or table name, but this MAY NOT always be the case. For example in the table designer, R:BASE may not warn you about **REF**, short for **REFERENCES**, but **REF** will not be allowed in a command file. If a particular column or table is giving you problems, please check out the list below and consider all shortened versions of the words listed here.

The following is a list of all known reserved words, commands, keywords, and other names that are in effect when ANSI is set on, which is the default setting.

ABORT	CHDIR	ENDIF	IFLT	MOD
ABS	CHDRV	ENDS	IFRC	MODULE
ABSOLUTE	CHECK	ENDSW	IHASH	MOUSE
ACOS	CHKDSK	ENDWHILE	IHR	MOVE
ADA	CHKKEY	ENTER	IMIN	MPW
ADD	CHOOSE	ENVVAL	IMON	MULTI
AINT	CLEAR	EQ	IN	NAME
ALL	CLOSE	EQNULL	INDEX	NE
ALTER	CLS	ERASE	INDICATOR	NEW
AND	COBOL	ERROR	INI	NEWPAGE
ANINT	CODELOCK	ESC	INITPOS	NEWROW
ANSI	COLLATE	ESCAPE	INPUT	NEXT
ANY	COLLATEC	EXCEPT	INSERT	NEXTROW
APPEND	COLOR	EXECUTE	INT	NEXTTAB
AS	COLUMN	EXISTS	INTEGER	NINT
ASC	COLUMNS	EXIT	INTENSITY	NOCHECK
ASCII	COMMENT	EXP	INTERSECT	NOECHO
ASIN	COMMIT	EXPAND	INTERVAL	NOFILL
ASSIGN	COMPUTE	EXPLODE	INTO	NOHEADER
AT	COMPUTED	EXPRESS	IS	NONE
ATAN	CONNECT	FAILS	ISEC	NONUM
ATAN2	CONSTRAINT	FASTFK	ISTAT	NOT
ATT	CONTAINS	FASTLOCK	IYR	NOTE

ATTACH	CONTINUE	FEEDBACK	IYR4	NOTE_PAD
AUTHORIZATION	COPY	FETCH	JDATE	NULL
AUTHORIZE	COS	FILES	JOIN	NUM
AUTOCHK	COSH	FILL	KEY	NUMERIC
AUTOCOMMIT	COUNT	FILLIN	KEYBOARD	OF
AUTOCONVERT	CREATE	FIRST	KEYMAP	OFF
AUTODROP	CROSSTAB	FLOAT	LABEL	ON
AUTONUM	CTR	FLUSH	LANGUAGE	OPEN
AUTONUMBER	CTXT	FOLD	LAST	OPTION
AUTORECOVER	CURRENCY	FOR	LASTKEY	OPTIONS
AUTOROWVER	CURRENT	FOREGRND	LAUNCH	OR
AUTOSKIP	CURSOR	FOREIGN	LAVG	ORDER
AUTOSYNC	CURSORS	FORMAT	LAYOUT	OUTER
AUTOUPGRADE	CVAL	FORMATTED	LBLPRINT	OUTPUT
AVERAGE	DAT	FORMS	LCFOLD	OWNER
AVG	DATA	FORTRAN	LE	PACK
BACKGRND	DATE	FOUND	LIKE	PAGE
BACKUP	DATETIME	FROM	LIMIT	PAGEMODE
BEEP	DEBUG	FULL	LINEEND	PAGEROW
BEGIN	DECIMAL	FV1	LINES	PASCAL
BEGINS	DECLARE	FV2	LIST	PASSTAB
BELL	DEFAULT	GATEWAY	LISTATT	PASSTHROUGH
BETWEEN	DEFINE	GE	LISTREL	PAUSE
BIT	DELETE	GET	LJS	PLAYBACK
BITNOTE	DESC	GETDATE	LMAX	PLI
BLINK	DETACH	GETKEY	LMIN	PLUGINS
BLINKING	DEXTRACT	GETVAL	LOAD	PMT1
BOTH	DIALOG	GO	LOCK	PMT2
BREAK	DIM	GOTO	LOG	POINTER
BRND	DIR	GRANT	LOG10	PRECISION
BROWSE	DISCONNECT	GROUPED	LOOKUP	PREF
BUILD	DISPLAY	GT	LOOKUPS	PREFIX
BY	DISTINCT	HAVING	LT	PREVROW
C	DOUBLE	HEADING	LUC	PREVTAB
CALL	DRAW	HEADINGS	MANOPT	PRIMARY
CASCADE	DROP	HELP	MAX	PRINT
CASE	DUPLICAT	ICAP1	MAXIMUM	PRINTER
CASEP	DUPLICATE	ICAP2	MAXTRANS	PRIOR
CD	ECHO	ICHAR	MD	PRIVILEGE
CGA	EDIT	IDAY	MENU	PRNSETUP
CENTURY	EDITOR	IDWK	MESSAGES	PROC
CHANGE	ELSE	IF	MIN	PROCEDURE
CHAR	END	IFEQ	MINIMUM	PROJECT
CHARACTER	ENDC	IFGT	MKDIR	PROMPT

PROMPTS	SATTACH	TAN
PROPERTY	SAVEROW	TANH
PUBLIC	SCHEMA	TDWK
PUT	SCONNECT	TERM1
PV1	SCRATCH	TERM2
PV2	SCREEN	TERM3
QBE	SCROLL	TERMINAL
QUALCOLS	SDETACH	TEXT
QUERY	SECTION	TEXTRACT
QUIT	SELECT	TIME
RATE1	SELMARGIN	TIMEOUT
RATE2	SEQUENCE	TITLE
RATE3	SERVER	TMON
RBAPP	SET	TO
RBBEDIT	SFIL	TOLERANCE
RBDEFINE	SGET	TRACE
RBEDIT	SHOW	TRANSACT
RBGSIZE	SIGN	TRIG
RBLABELS	SIN	TYPE
RBSYNC	SINH	UDF

RBSYSTEM	SKIP	ULC
RD	SLEN	UNION
RDATE	SLOC	UNIQUE
READ	SMALLINT	UNLOAD
READ/WRITE	SMOVE	UNNAMED
REAL	SNAP	UPDATE
RECALC	SOME	UPGRADE
RECORD	SORT	USER
RED	SORTED	USERAPP
REDEFINE	SOUNDS	USING
REFERENCES	SOUNDS_L	VALUES
REFRESH	SOUNDS_LIKE	VARBIT
RELATIVE	SPUT	VARCHAR
RELOAD	SQLCODE	VARIABLE
REMOVE	SQLERROR	VARIANCE
RENAME	SQRT	VERIFY
REPORTS	SRPL	VERSION
RESET	SSQL	VIEW
RESTORE	SSUB	VIEWS
RETURN	STARTUP	WAIT
REVERSE	STATICDB	WALKMENU
REVOKE	STDEV	WHENEVER
RHIDE	STORE	WHERE
RJS	STRIM	WHILE
RMDIR	STRUCTURE	WHILEOPT
ROLLBACK	SUB	WIDTH
ROUND	SUBTRACT	WITH
ROW	SUFFIX	WORK
ROWLOCKS	SUM	WRAP
RPHONE	SWITCH	WRITE
RPW	SYS	YEAR
RSHOW	TABLE	ZERO
RTIME	TABLES	ZIP
RULES	TABSIZE	
RUN	TALLY	

If you are still using a legacy version of R:BASE, it is recommended that you perform these table and column name changes in that version. After making the table and column names changes, R:BASE will update the form, report, and label column objects with the appropriate new name. However, if any of the columns and tables are listed in the form, report, and label variable expressions, you must manually edit these expressions.

On paper, record any table and column name changes. Later in the conversion process when you're updating your command files, you will need to make these necessary name changes for the command files as well.

8.14 SQL - Information

Structured Query Language (SQL) commands provide a standard, machine-independent relational database language.

The American National Standards Institute (ANSI) provides a basic description of the SQL language. The R:BASE/Oterro database provides a superset of this standard. The Oterro database implementation meets the requirements of ANSI 1989 Level 2 SQL with 1992 extensions.

SQL, a language developed specifically for relational databases, enables you to define, modify, and query a relational database. SQL is not a programming language, but a standard set of commands that work with a relational database.

The R:BASE/Oterro database incorporates the SQL commands into its broader range of commands. SQL commands are not treated separately in the Oterro database command because they are not special in any sense. SQL was originally defined as and must be considered an intrinsic part of a database management system.

SQL provides the following sets of commands:

Data Definition Language

Includes the commands needed to create the basic database structures—tables, columns, and views.

- ALTER TABLE (extension to SQL)
- COMMENT ON (extension to SQL)
- CREATE INDEX (extension to SQL)
- CREATE SCHEMA AUTHORIZATION
- CREATE TABLE
- CREATE VIEW
- DROP INDEX (extension to SQL)
- DROP TABLE (extension to SQL)
- DROP VIEW (extension to SQL)

Data Manipulation Language

Provides modification and query capabilities.

- DELETE
- INSERT
- SELECT
- UPDATE

Data Security Language Commands

Control access to the database.

- GRANT
- REVOKE (extension to SQL)

Transaction Processing Commands

Control when data is saved in the database, thereby allowing the restoration of data to a previous state.

- SET AUTOCOMMIT (extension to SQL)
- SET LOCK (extension to SQL)
- SET MAXTRANS (extension to SQL)
- SET TRANSACT (extension to SQL)

8.15 Stored Procedures & Triggers

A Stored Procedure is a collection of R:BASE commands and/or SQL statements that are stored in the database.

Stored Procedures offer a powerful way for developers to add value and ease of maintenance to their R:BASE databases and applications. Moving some of the common business and data access logic out of the R:BASE program into the database centralizes functionality in one place, making it accessible to the R:BASE program as well as any third party ODBC application.

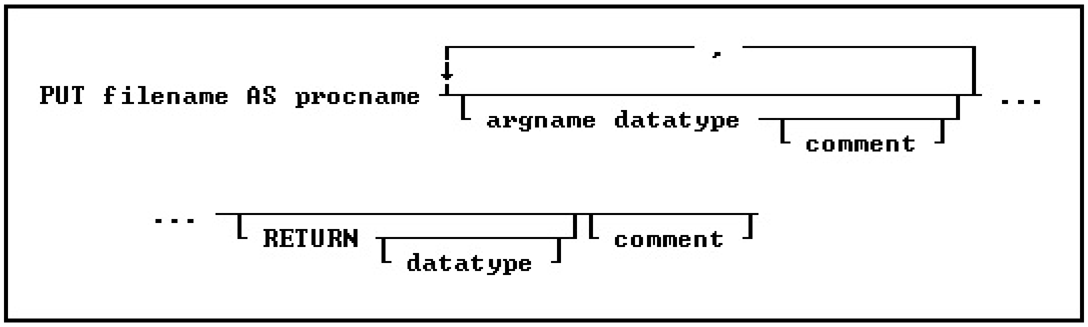
You can run a Stored Procedure "manually" using the CALL command, or "automatically" by using database Triggers.

8.15.1 Creating Stored Procedures

Stored Procedures are created based upon an existing command file. The process of loading the command file into the database as a Stored Procedure can be performed through the Database Explorer "Stored Procedure" Group Bar menu option, or by using the PUT command.

PUT

A Stored Procedure can be created in the R:BASE Editor, and then can be loaded into the database as follows:



Options:

argname datatype
The argument name and data type.

comment
An optional comment for the argument or, if placed after RETURN, an optional comment for the entire procedure. The comment must be enclosed in the current QUOTES character.

filename
The filename in ASCII text format, with full path, to load as the Stored Procedure.

procname
Specifies the procedure name. If a procedure by this name already exists in the database, an error is generated. The procedure name is limited to 128 characters.

RETURN datatype
Datatype of the value returned by the procedure.

Note:

- To clear any previous arguments that were stored for a procedure, use the PUT command as follows:

```
PUT filename AS procname RETURN
```

8.15.2 Using Stored Procedures

Argument List
When you load a Stored Procedure into a database, you specify arguments to be passed to it. These arguments are used within the procedure. When the procedure is called, the number and type of arguments passed must match the number and type specified when the procedure was stored in the database. When an argument name is referenced in the Stored Procedure code, the argument name must be preceded by a period unless it is a table or column name, then it must be preceded by an ampersand (&). For example:

```
UPDATE &p1_table SET col  = 99 WHERE col = .p2
```

The arguments names are specified when the procedure is stored in the database.

Return Values

The value to be returned by a Stored Procedure is specified in the procedure code following the keyword RETURN. For example, RETURN 'Los Angeles'.

The value returned must match the data type specified when the procedure was stored.

Notes:

- If you are replacing an existing procedure, you must LOCK the procedure first with either the GET LOCK or the SET PROCEDURE command. Once the procedure is locked, it is replaced by an updated file using the PUT command. A procedure cannot be replaced unless it is locked. A procedure is automatically unlocked when replaced with the PUT command.
- The RETURN varname option is used ONLY within a Stored Procedure to return a value. The returned value is stored in the STP_RETURN system variable. This option will return an -ERROR- when used outside a Stored Procedure. The default is TEXT 8 characters, but if you want more, you can set it to a larger value.

  You can control the maximum length at procedure definition time, or by editing the SYS_PROC_LEN column in SYS_PROC_COLS system table.

  Example 01 (to set the limit for the RETURN value to 30 characters):

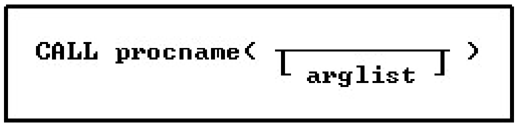
  ```
  PUT MyTest.PRC AS MyTest P1 INTEGER RETURN TEXT (30)
  ```

- To clear any previous arguments that were stored for a procedure, use the PUT command as follows:

  ```
  PUT filename AS procname RETURN
  ```

8.15.2.1 CALL

To run a Stored Procedure you can use the CALL command like a function or run the CALL command at the R> Prompt.



Use the CALL command like a function with the following syntax:

```
SET VAR v1 = (CALL procname(arglist))
```

Run the CALL command from the R> Prompt with the following syntax:

```
CALL procname(arglist)
```

When the CALL command is run at the R> Prompt, the return value from the Stored Procedure is placed in the variable STP_RETURN. The return value can be an expression.

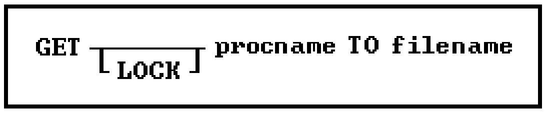
procname
The procedure name.

arglist
The argument values separated by commas. An *arglist* must always be used, even if empty. For example:

```
SET VAR v1 = (CALL procname ( )).
```

8.15.2.2 GET

The GET command is used to read a Stored Procedure from the database into an ASCII command file. If the LOCK option is used with the GET command, the procedure can be replaced by using PUT.



Options:

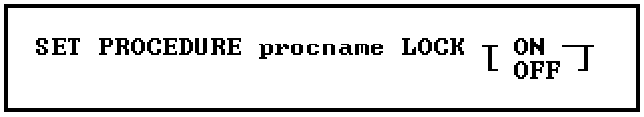
filename
The name of the ASCII text format file the Stored Procedure is placed in.

LOCK
Locks the procedure so it cannot be locked or unlocked by another user. When a procedure is locked, only the user placing the lock can replace the procedure. The NAME setting is used for identification of the user.

procname
Specifies the procedure name. The procedure name is limited to 128 characters.

8.15.2.3 SET PROCEDURE

The SET PROCEDURE command locks a procedure so it can be replaced. It works like the GET LOCK command without retrieving the Stored Procedure into an ASCII file. ON sets the lock; OFF releases the lock placed by the SET PROCEDURE or the GET commands.



When a procedure is locked, only the user placing the lock can replace the procedure or remove the lock. The NAME setting is used for identification of the user.

8.15.2.4 Examples

To Create a Stored Procedure

Create the following command file, INS.RMD, in R:BASE Editor:

```
*(INS.RMD:)
IF (.p1 > 105) THEN
  INSERT INTO contact (custid, contlname) VALUES (.p1, .p2)
  RETURN 1
ELSE
  RETURN 0
ENDIF
```

To create the Stored Procedure from the .RMD file:

```
PUT INS.RMD AS proc1 p1 INT, p2 TEXT RETURN INTEGER
```

The following Stored Procedure example will generate one new row in *contact* and set v1 = 1.

```
SET VAR vname = 'Dunn'
SET VAR v1 = (CALL proc1 (106, .vname))
```

The following Stored Procedure example will set v1 = 0.

```
SET VAR vname = 'Dunn'
SET VAR v1 = (CALL proc1 (100, .vname))
```

To Delete a Stored Procedure

To delete a Stored Procedure, use the DROP command with the following syntax:

```
DROP PROCEDURE procname
```

Optionally, you can enter the following code at the R> Prompt:

```
SET VAR vProcID = sys_proc_id IN sys_procedures +
  WHERE sys_proc_name = 'procname'
DELETE ROWS FROM sys_procedures +
  WHERE sys_proc_id = .vProcID
DELETE ROWS FROM sys_proc_mods +
  WHERE sys_proc_id = .vProcID
DELETE ROWS FROM sys_proc_cols +
  WHERE sys_proc_id = .vProcID
```

To Rename a Stored Procedure

To rename a Stored Procedure, use the RENAME command with the following syntax:

```
RENAME PROCEDURE procname1 TO procname2
```

To List Stored Procedures

With the LIST command, you can list every Stored Procedure in a database or list information about a specific Stored Procedure.

To display the name and a description for every procedure in the open database, use the following syntax:

```
LIST PROCEDURE
```

To display a specific procedure and its attributes, use the following syntax:

```
LIST PROCEDURE procname
```

This option displays the name, description, ID, date last modified, version, locked by (if locked), and if applicable, the return type and description for the specified Stored Procedure. If the Stored Procedure has arguments, the number of arguments and argument names and attributes will be listed.

8.15.3 Restricted Commands

Stored procedures can contain all R:BASE/SQL commands except for the following:

CODELOCK	FORMS
RBEDIT	RESTORE
CONNECT	PACK
RBLABELS	RULES
DISCONNECT	SET (without a keyword)
RBAPP	RBDEFINE
REPORTS	REVOKE
GRANT	

8.15.4 Stored Procedure System Tables

The R:BASE system tables are created by R:BASE when a database is created. They contain system information. The following are new system tables. Stored Procedures are stored in the database in the system table called SYS_PROCEDURES. Supporting system tables are SYS_PROC_COLS and SYS_PROC_MODS.

Table: sys_PROCEDURES

Column Name	Data Type	Description
sys_proc_id	INTEGER	Procedure identification
sys_proc_name	NOTE	Procedure name
sys_proc_locked_by	NOTE	Last user to do a PUT or LOCK
sys_proc_comment	NOTE	Descriptive comment for procedure
sys_proc_src	LONG VARCHAR	Procedure source
sys_proc_mod_ts	DATETIME	Timestamp of procedure
sys_proc_obj	LONG VARBIT	Reserved for future use
sys_proc_usage	INTEGER	Reserved for future use
sys_proc_flags	INTEGER	Internal binary flags. Bit 0 is the LOCK flag.
sys_proc_version	INTEGER	Version number of procedure

Table: sys_proc_cols

Column Name	Data Type	Description
sys_proc_col_id	INTEGER	Argument identification
sys_proc_id	INTEGER	Procedure identification
sys_proc_col_name	NOTE	Name of argument
sys_proc_iotype	INTEGER	Argument type values equal to: SQL_RETURN_VALUE (5) SQL_PARAM_OUTPUT (4) SQL_PARM_INPUT (1) SQL_PARAM_INPUT_OUTPUT (2) Currently only INPUT and RETURN types are supported.
sys_proc_datatype	INTEGER	Datatype of argument
sys_proc_len	INTEGER	Argument data length
sys_proc_scale	INTEGER	Argument data scale
sys_proc_flags	INTEGER	Internal binary flags
sys_proc_comment	NOTE	Descriptive comment for argument
sys_proc_defvalu	NOTE	Reserved for future use

Table: sys_proc_mods

Column Name	Data Type	Description
sys_proc_mod_id	INTEGER	Archive identification
sys_proc_id	INTEGER	Procedure identification
sys_proc_mod_ts	DATETIME	Timestamp of archived procedure
sys_proc_user	NOTE	User who did a PUT on procedure
sys_proc_comment	NOTE	Descriptive comment for procedure
sys_proc_fc	LONG VARCHAR	Reserved for future use
sys_proc_delta	LONG VARBIT	Reserved for future use
sys_proc_version	INTEGER	Version of archive

Table: sys_TRIGGERS

Column Name	Data Type	Description
sys_table_id	INTEGER	Table identification
sys_trig_ins	INTEGER	Id of INSERT procedure

sys_trig_upd	INTEGER	Id of UPDATE procedure
sys_trig_del	INTEGER	Id of DELETE procedure

8.15.5 Triggers

A database trigger is procedural code that is automatically executed in response to certain events on a particular table in the database. Triggers can restrict access to specific data, perform logging, or audit data modifications.

There are "BEFORE" triggers and "AFTER" triggers which identifies the time of execution of the trigger. A trigger can be set to automatically run a Stored Procedure before and/or after an update, delete, or insert event occurs in a table.

There are three triggering events that cause triggers to fire:

- INSERT event (as a new record is being inserted into the database).
- UPDATE event (as a record is being changed).
- DELETE event (as a record is being deleted).

Notes:

- R:BASE triggers occur only once per INSERT, UPDATE, or DELETE event.
- A table can have both "BEFORE" and "AFTER" Triggers, only one, or none.
- Triggers are stored in the System Table SYS_TRIGGERS
- Procedures that are run with triggers must be stored with no arguments. See "**Argument List**" under Using Stored Procedures

8.15.5.1 Using Triggers

The update, delete, or insert event can be initiated through the UPDATE, DELETE, or INSERT R:BASE/SQL commands, or through an R:BASE form.

Typical Trigger Usage:

- BEFORE - data validation before the action (inventory checks, account limit checks)
- AFTER- update of data after the action (dependent on primary keys, automated post transaction steps)

Since a "BEFORE" Trigger runs a Stored Procedure before the row that triggered it is updated, inserted, or deleted, you can cancel the update, insert, or delete with the ABORT TRIGGER command in the Stored Procedure. Since the modified data has been "committed" with an "AFTER" trigger, you cannot abort the action in the Stored Procedure.

Also, you can verify the action being performed in an update trigger on the row by using a SELECT command with the WHERE CURRENT OF SYS_OLD/SYS_NEW syntax to check the row before/after the update.

Creating a Trigger

Triggers can be created using the Data Designer, or with the CREATE TABLE or ALTER TABLE commands. When you use the ALTER TABLE command you must define the insert triggers in the same command. The same applies for update and delete. Do not use one alter table command to add the "BEFORE" insert trigger and then another alter table to add the "AFTER" trigger. Do them both in the same command.

Removing a Trigger

When you drop a trigger with the ALTER TABLE command, you do not have to specify the "BEFORE" or "AFTER" trigger. The DROP of the insert trigger, for example, drops both parts if they exist.

Listing Defined Triggers

To LIST all the tables in the open database that have triggers and the triggers, use the following syntax:

```
        LIST TRIGGER
```

To list triggers for a specified table, use the following syntax:

```
        LIST TRIGGER tblname
```

Example:

```
  ALTER TABLE TableName ADD TRIGGER INSERT ProcName
  ALTER TABLE TableName ADD TRIGGER UPDATE ProcName
  ALTER TABLE TableName ADD TRIGGER DELETE ProcName
  ALTER TABLE TableName ADD TRIGGER INSERT AFTER ProcName
  ALTER TABLE TableName ADD TRIGGER UPDATE AFTER ProcName
  ALTER TABLE TableName ADD TRIGGER DELETE AFTER ProcName
```

See also:

SYS_NEW
SYS_OLD

For a sample database using Triggers, please locate the "Stored Procedures, Triggers and After Triggers" sample located at http://www.razzak.com/sampleapplications/

8.15.5.2 SYS_NEW

The SYS_NEW parameter is used in a WHERE clause within, and only within, the context of a Trigger.

This virtual pointer is available to INSERT and UPDATE triggers. It allows code to access the contents of the row as it will be after the INSERT or UPDATE action. Using this in the body of a WHERE clause allows code to act on the contents of that virtual row and NOT fire off another trigger.

The following is a list the trigger types available for use with SYS_NEW, and whether or not they are updatable:

- BEFORE INSERT: Updatable
- AFTER INSERT: Read only
- BEFORE UPDATE: Updatable
- AFTER UPDATE: Read only

Note: The use of functions or expressions must be performed outside of the virtual pointer SELECT statement, after the the variable values are captured.

Example

The following command is within the body of an Insert Trigger and is being used to increment the count of how many items have been used.

```
SELECT ProductType INTO vPType INDIC v1 +
  FROM SalesDetails +
  WHERE CURRENT OF SYS_NEW
UPDATE ProductCount +
  SET ProductCount = (ProductCount + 1) +
  WHERE ProductType = .vPType
```

8.15.5.3 SYS_OLD

The SYS_OLD parameter is used in a WHERE clause within, and only within, the context of a Trigger.

This virtual pointer is available to UPDATE and DELETE triggers. It allows code to access the contents of the row as it will be prior the UPDATE or DELETE action. Using this in the body of a WHERE clause allows code to act on the contents of that virtual row and NOT fire off another trigger.

The following is a list the trigger types available for use with SYS_OLD, and whether or not they are updatable:

- BEFORE DELETE: Read only
- BEFORE UPDATE: Read only

Note: The use of functions or expressions must be performed outside of the virtual pointer SELECT statement, after the the variable values are captured.

Example

The following command is within the body of an Delete Trigger and is being used to automatically archive a message from an employee messaging table.

```
SELECT EmployeeID,MsgDate,MsgTime,MsgBody +
  INTO vEID INDIC v1,vMsgDate INDIC v2, +
  vMsgTime INDIC v3,vMsgBody INDIC v4 +
  FROM EmpMessage WHERE CURRENT OF SYS_OLD
INSERT INTO ArchMessage +
  (EmployeeID,MsgDate,MsgTime,MsgBody,DeletedOn) +
  VALUES .vEID,.vMsgDate,.vMsgTime,.vMsgBody,.#DATE
```

8.16 Table Joins

A join is an SQL clause that combines records from two tables in a database. When you perform a join, you specify one column from each table to join on. These two columns contain data that is shared across both tables. You can use multiple joins in the same SQL statement to query data from as many tables as you like.

A join can be an inner join, an outer join, or a self join. An inner join includes only those rows that match on the linking columns. An outer join includes all rows that match as well as all rows that don't match on linking columns.

Most of the time, you'll do an inner join, though you will sometimes find it useful to do an outer join. For example, you need an outer join (like the UNION command) to get all rows in these cases:

- When joining a *customer* table with an *orders* table to list the customers who ordered something in the current month (inner join) as well as those who didn't order anything (outer join).
- When joining a *budget* table with an *expense* table to list each budget item, whether or not there was an expense for that item in the current month.
- When comparing a header (master table) on the "one" side of a one-to-many relationship against a detail (transaction table) on the "many" side to see all the rows of data, whether or not they have associated details.

In R:BASE, there are two different syntactical ways to express joins. The first, called *explicit join notation*, uses the keyword JOIN, whereas the second is the *implicit join notation*. The implicit join notation uses commas to separate the tables to be joined in the FROM clause of a SELECTstatement. Thus, it always computes a cross join and the WHERE clause may apply additional filtered criteria. That filter criteria is comparable to join predicates in the explicit notation.

Example of an explicit "inner" join:

```
    SELECT ALL FROM Product +
    INNER JOIN TransDetail ON +
    TransDetail.Model = Product.Model
```

Example of an implicit "inner" join:

```
SELECT ALL FROM Product t3, TransDetail t2 +
  WHERE t3.Model = t2.Model
```

Both of the above examples will result in the same output, only the example of the explicit "inner" join will be faster.

8.16.1 Join Types

Depending on your requirements, you can do an "INNER" join or an "OUTER" join. The differences are:

- **INNER JOIN:** This will only return rows when there is at least one row in both tables that match the join condition.
- **LEFT OUTER JOIN:** This will return rows that have data in the left table (left of the JOIN keyword), even if there's no matching rows in the right table.
- **RIGHT OUTER JOIN:** This will return rows that have data in the right table (right of the JOIN keyword), even if there's no matching rows in the left table.
- **FULL OUTER JOIN:** This will return all rows, as long as there's matching data in one of the tables.

8.16.2 More About OUTER JOIN

When you use an outer join, rows are not required to have matching values. The table order in the FROM clause specifies the left and right table. You can include a WHERE clause and other SELECT clause options such as GROUP BY. The result set is built from the following criteria:

- In all types of outer joins, if the same values for the linking columns are found in each table, R:BASE joins the two rows.
- For a left outer join, R:BASE uses each value unique to the left (first) table and completes it with nulls for the columns of the right (second) table when the linking columns do not match.
- A right outer join uses unique values found in the right (second) table and completes the rows with nulls for columns of the left (first) table when the linking columns do not match.
- A full outer join first joins the linking values, followed by a left and right outer join.

Four Examples of Outer Joins

Below, from slowest to fastest, are four examples of how to list all the invoice numbers in an *invoice* table, whether or not they have related rows in a *transx* table.

In each example, *invoice* has 1,000 rows and *transx* has 8,000 rows. There are 20 matches and 980 non-matches. In other words, in the first three examples the first SELECT (inner join) finds 20 rows and the second SELECT (outer join) finds 980 rows. And, in the last example, the LEFT OUTER JOIN performs the query with just one SELECT.

Uncorrelated Sub-SELECT
l
This first example shows how to list the invoice numbers with a simple sub-SELECT that doesn't have a WHERE clause correlating it to the main SELECT. It's slow, taking a long time to complete.

```
SELECT t2.InvId, SUM(t3.TPrice) +
  FROM Invoice t2, Transx t3 +
  WHERE t3.InvId = t2.InvId +
  GROUP BY t2.InvId +
  UNION +
    SELECT Invoice.InvId, $0.00 +
      FROM Invoice +
      WHERE InvId NOT IN +
          (SELECT InvId FROM Transx)
```

Correlated Sub-SELECT

Adding a correlated WHERE clause to the sub-SELECT makes it many times faster.

```
SELECT t2.InvId, SUM(t3.TPrice) +
  FROM Invoice t2, Transx t3 +
  WHERE t3.InvId = t2.InvId +
  GROUP BY t2.InvId +
  UNION +
    SELECT t1.InvId, $0.00 +
      FROM Invoice t1 +
      WHERE InvId NOT IN +
        (SELECT InvId FROM Transx +
        WHERE Transx.InvId = t1.InvId)
```

Correlated and NOT EXISTS

By changing NOT IN to NOT EXISTS for use with the correlated sub-SELECT, you can add a little more speed.

```
SELECT t2.InvId, SUM(t3.TPrice) +
  FROM Invoice t2, Transx t3 +
  WHERE t3.InvId = t2.InvId +
  GROUP BY t2.InvId +
  UNION +
    SELECT tl.InvId, $0.00 +
      FROM Invoice t1 +
      WHERE NOT EXISTS +
        (SELECT InvId FROM Transx +
        WHERE Transx.InvId = t1.InvId)
```

LEFT OUTER JOIN

By using a LEFT OUTER JOIN, and bypassing the second SELECT, you can add even more speed.

```
SELECT t2.InvId, SUM(t3.TPrice) +
  FROM Invoice t2 LEFT OUTER JOIN Transx t3 ON +
  t3.InvId = t2.InvId +
  GROUP BY t2.InvId
```

8.17 Temporary Tables and Views

Temporary tables and views are non-permanent tables/views that only exist for the duration of a database session. When a database session terminates, its temporary tables/views are automatically destroyed. Temporary tables/views are only visible to the R:BASE session that creates them and remain invisible to other R:BASE users. Several users can create temporary tables/views with the same name, and each user will see only that particular version of the table/view.

Temporary tables are ideal for holding short-term data used by the current R:BASE session. For example, suppose you need to do many SELECTs on the result of a complex query. An efficient strategy is to execute the complex query once, then store the result in a temporary table. You can also create an index on the temporary table to speed up queries with the CREATE INDEX command. In addition to indexes, you can create rules, constraints and triggers on temporary tables.

You may CREATE or turn a permanent table into a TEMPORARY table using the enhanced Data Designer.

8.17.1 Using Temporary Tables/Views

Creating Temporary Tables/Views
Use the CREATE TEMPORARY TABLE command to create a temporary table. Use the CREATE TEMPORARY VIEW command to create a temporary view. You can also use the PROJECT command to create a temporary table based upon another table. The main difference between using the original command to create a table or view and creating a temporary table/view is the TEMORARY parameter in the command syntax.

Temporary Table Setting
Within the R:BASE Data Designer, there is a setting, under "Table", that will save any temporary table as a regular permanent table. The setting can also be used to save any existing table as a temporary table.

Removing Temporary Tables/Views
You can use the DROP command to remove a temporary table or view. This is important when you are creating temporary tables or views as you should always DROP the table or view before creating it. This technically only affects the user running the program. To avoid this error message, use the following technique:

Example 01:
```
SET ERROR MESSAGE 2038 OFF
DROP TABLE temptablename
SET ERROR MESSAGE 2038 ON
```

Example 02:
```
SET ERROR MESSAGE 677 OFF
DROP VIEW tempviewname
SET ERROR MESSAGE 677 ON
```

In addition to the DROP command, you can simply DISCONNECT and CONNECT the database to remove any temporary tables/views. Temporary tables/views along with temporary System Tables are removed when the database is disconnected.

Using Forms, Labels or Reports based upon Temporary Tables/Views
You may define all required TEMPORARY tables/views for Forms as "On Before Design Action" before designing a form and "On Before Start" EEP before using (EDIT USING, ENTER USING, BROWSE USING) the form.

You may define all required TEMPORARY tables/views for Reports as "On Before Design ..." action before designing a report/sub-report and "On Before Generate ..." action.

You may define all required TEMPORARY tables/views for Labels as "On Before Design ..." action before designing a label and "On Before Generate ..." action.

A command file, such as TempTables.RMD, can also be used to pre-define all required temporary tables/views.

Notes:

- In regards to temporary tables in the database, always DISCONNECT and then CONNECT before using the AUTOCHK, PACK or RELOAD commands.
- When creating a temporary table/view, the following temporary System Tables are also created:
  - SYSTMP_COMMENTS
  - SYSTMP_CONSTRAINTS
  - SYSTMP_DEFAULTS
  - SYSTMP_RULES
  - SYSTMP_SERVERS
  - SYSTMP_TRIGGERS
  - SYSTMP_VIEWS
- Temporary tables/views were first introduced in R:BASE 6.1 (July 1997).

8.17.2 Differentiate between Regular and Temporary Tables/Views

There are two ways to differentiate between regular and temporary tables/views. One way is to view the Tables or Views menu within the Database Explorer. You will notice that any temporary tables or views will have a faded icon next to the appropriate table/view name.

Another way is using the LIST command at the R> Prompt. To safely indicate which tables are temporary tables in the LIST or LIST TABLES command, you will now see a "(T)" in front of the table name.

Here's how using the ConComp sample database:

1. Launch R:BASE
2. CONNECT ConComp
3. Switch to the R> Prompt and create a temporary tables using the following:

```
PROJECT TEMPORARY tCustomer FROM Customer USING ALL
```

Or use the following block of code in a command file to create two temporary tables

```
SET ERROR MESSAGE 2038 OFF
DROP TABLE tInvoiceHeader
CREATE TEMPORARY TABLE `tInvoiceHeader` +
(`TransID` INTEGER, +
`CustID` INTEGER, +
`EmpID` INTEGER, +
`TransDate` DATE, +
`BillToCompany` TEXT (40), +
`BillToAddress` TEXT (30), +
`BillToCity` TEXT (20), +
`BillToState` TEXT (2), +
`BillToZip` TEXT (10), +
`ShipToCompany` TEXT (40), +
`ShipToAddress` TEXT (30), +
`ShipToCity` TEXT (20), +
`ShipToState` TEXT (2), +
`ShipToZip` TEXT (10), +
`NetAmount` CURRENCY, +
`Freight`= ( netamount* .01) CURRENCY, +
`Tax`= ( netamount* .081) CURRENCY, +
`InvoiceTotal`= (NetAmount+Freight+Tax) CURRENCY)
COMMENT ON TABLE tInvoiceHeader IS 'Invoice Header Information'
DROP TABLE tInvoiceDetail
CREATE TEMPORARY TABLE `tInvoiceDetail` +
(`TransID` INTEGER, +
`DetailNum` INTEGER, +
`Model` TEXT (6), +
`Units` INTEGER,  +
`Price` CURRENCY, +
`Discount` REAL, +
`SalePrice`= (Price-(Price*Discount/100)) CURRENCY, +
`ExtPrice`= (Units* SalePrice) CURRENCY)
AUTONUM `DetailNum` IN `tInvoiceDetail` USING 1,1
COMMENT ON TABLE tInvoiceDetail IS 'Invoice Header Information'
SET ERROR MESSAGE 2038 ON

LIST TABLES
```

You will notice the (T) in front of the temporary tables.

4. Now create a temporary view using following:

```
SET ERROR MESSAGE 677 OFF
DROP VIEW tYTDInvoiceTotal
CREATE TEMP VIEW `tYTDInvoiceTotal` +
(CustID, YTDInvoiceTotal) AS  +
SELECT CustID,(SUM(InvoiceTotal)) FROM TransMaster +
GROUP BY CustID
COMMENT ON VIEW `tYTDInvoiceTotal` IS +
'Year-To-Date Invoice Total by Customer'
SET ERROR MESSAGE 677 ON

LIST VIEWS
```

You will notice (T) in front of the tYTDInvoiceTotal view.

8.17.3 Advantages of Temporary Tables/Views

Raw Speed

Temporary Tables/Views are lightening fast! There is no multi-user checking going on.

Find a report that prints from a view whose performance is extremely slow, project a temporary table containing only the rows needed and drive the report from the temporary table. A report that takes 5-10 minutes to print might print in under a minute.

Flexibility

Because of the speed, you can do things you would never do with permanent tables. If you have systems that do an extraordinary amount of massaging of data placed in a temporary table. The work would take far longer (we are talking 10-100 times) to accomplish with a permanent table. And with permanent tables, deleting your scratch work takes a great deal of time and each record must have a user id in it to work correctly. With temp tables you just reconnect the database and all the temp tables are gone, just like that.

Temporary tables/views are also supported when STATICDB is set to ON.

No database growth

The data in the temporary tables is not part of File 2- the data file. The most important thing about temporary tables is that the actual data for a given user is written into a scratch file (.$$$). With an actual table that data is stored in the data file.

For example, if you are using an actual table and start with a 100MB data file and add 5MB of "temporary data" to a database, then drop the working table. Now, your data file is 105MB. Then, run the procedure again and you'll have 110MB. Running the procedure two more times and your data file is 120MB, and so on. You would have perform a PACK or RELOAD just to return back to the 100MB data file after processing temporary data within actual tables. On the other hand, if you use temporary tables your data file doesn't grow at all.

Views don't really make much difference since the only thing that would be stored in the database itself (and they still might be with temporary views) would be the structure. Everything else is generated when you actually use the view.

Independent

Temporary Tables/Views are specific to each session of R:BASE and that specific user.

For example, five different users or sessions, can create the same temporary table with the same name and not interfere with the others. Only the user that created the table can see/use it. So what it means is that 5 different uses can be using a running a report on the temporary table/view and all 5 users have different data in the table.

The "Sales Order" option in the "Running R:BASE Your Way" sample applications bundled with R:BASE, demonstrate the typical use of this feature.

Usability

You can treat a temporary table/view as a regular table/view. You can create Forms, Reports and Labels based upon the temporary table/view.

A powerful use of temporary tables is to PROJECT or CREATE a temporary table to collect (LOAD) data and allow easy editing prior to an INSERT. Since each session of R:BASE will project/create its own private temporary table (of the same name) this is an ideal solution, say for collecting some accounting data prior to allowing the user to post the transaction to the formal journal tables. As soon as the insert is done, a DISCONNECT/CONNECT will eliminate the temporary table and you are ready for next time.

Temporary tables are great when you are trying to take a huge vertical table with hundreds of thousands of rows and farm it out to some aggregate tables.

Sometimes when you need to insert row(s) into a table based on rows in the table, (the where clause cannot refer to the same table for the insert) You may project a temp table of the correct where conditions and insert where column in permanent table in (select column from temp table). You can do all kinds of variations of this one, such as using existing rows as a template which you house in a temp table, edit for the new values and re-insert back to the permanent table.

Disadvantages of Temporary Tables/View:

The advantage of temporary tables/views is also their disadvantage. They are not permanent. Any data you wish to be persistent must go in real tables/views.

Part IX

9 Troubleshooting

If you are having problems with Oterro, there are several things you should check. The list below is not intended to be exhaustive, but it should cover the common reasons you cannot connect to or use your Oterro databases.

1. Have you created a Data Source Name or DSN? If not, you may need to create one. However, not all products require a DSN. R:BASE and Visual Basic, for example, can utilize DSN-less connections.

2. Does the program using Oterro have access to the DSN? For example, some applications are installed by default as a system level application. If you create a User DSN, then the system application will not be able to access that data source. Conversely, Visio seems to require User based DSNs. Other programs may vary.

3. Do the database character settings match the ODBC defaults? To check this, go to an R> Prompt and type SHOW CHAR while connected to the database. QUOTES should be set to ' (the single quote or apostrophe), MANY should to be set to the % (the percent sign), SINGLE should be set to _ (the underscore) and IDQUOTES should be set to ` (the back quote, usually found above the tilde or ~ character).

4. Can you use the "Check Version" utility to get the Oterro Version? If not, the driver may not be properly installed.

5. Does the Oterro Driver display a version? If the driver is listed in the ODBC applet, but the version is not, then make sure the Oterro DLLs were installed to your Windows System directory (OTERRO11.DLL and OTERRO11_INS.DLL in the SysWOW64 folder for 32-bit operating systems and OTERRO11_64.DLL and OTERRO11_64_INS.DLL in the System32 folder for 64-bit operating systems). If these files are not present, be sure you have write or create permissions on the system directory. If you do not, have an Administrator remove and reinstall the program.

6. Can you connect to the database with R:BASE? Are there errors?

7. Is there a Multi-User, StaticDB, Transaction or other database mode conflict? To test this, see if you can connect to the sample DSNs with nobody else connected. You are not likely to have other connections with the samples. If you can connect to the sample, check that the other R:BASE or Oterro Engines connected to the database are of the same version (R:BASE 11, Runtime for R:BASE 11, R:Compiler for R:BASE 11, and Oterro 11), and are using the same settings.

8. Have you exceeded your License Count with a Numbered Version of Oterro 11? If you have, then you will not be able to connect to the database. For example, if you have a 5 User version of Oterro 11 and there are already 5 connections to the database, then you will not be able to connect. It does NOT matter if those other connections were made with Oterro 11 OR R:BASE OR Runtime. Once the user count is reached, Oterro 11 Numbered will not be able to connect.

9. Is the database itself unhealthy? Check to see if the database is out of sync (Run RBSYNC), if the database needs transaction recovery (Run RECOVER), or is in need of a Pack or Reload.

10. Can you use R:BASE to connect to the DSN? Use the following commands at the R> Prompt to create a dummy database and attempt to connect to your DSN.

    - CREATE SCHEMA AUTH TestDSN
    - SCONNECT dsn_name_here
    - SATTACH a_table_name_in_your_database

    If you have an Owner or Password on the database, or if you don't know your DSN Name, then replace step b. with just SCONNECT and you will be prompted. Similarly using just SATTACH will list all available tables.

11. Are you using reserved characters or words in your DSN name? This will vary based on the environment used to connect to the DSN. The Borland Database Engine, for example, doesn't seem to deploy (but you can develop) with a DSN that contains the _ (underscore character). This may depend on the version or compilation method of the product that you are using as well.

12. Are you using reserved characters or words in your Table and Column names? For example, R:BASE cannot use columns from another DBMS that contain spaces or that exceed 128 characters in length.

13. Are you using SQL that is compatible with the Oterro 11 engine? For example, some other DBMSs use the following syntax to denote a date: {d 12/27/2000}. Oterro will not understand this format. Check the Oterro documentation for a complete listing of supported commands and functions. Crystal Reports is notorious for generating syntax that Oterro does not comprehend. A well-developed tool such as Crystal Reports will allow you to at least view, if not edit, the SQL that is sent to the Oterro 11 Engine.

Part X

10 Technical Support

Please read over the help documentation at least once before seeking support. We have worked very hard to make the help documentation clear and useful, but concise. It is suggested that you reread these instructions once you have become accustomed to using the software, as new uses will become apparent.

If you have further questions, and cannot find the answers in the documentation, you can obtain information from the below sources:

- Email our Technical Support Staff at: support@rbase.com
- Access the R:BASE Technologies Support home page online at https://www.rbase.com/support

You may be required to purchase a technical support plan. Several support plans are available to suit the needs of all users. Available Technical Support Plans

Please be prepared to provide the following:

- The product registration number, which is located on the invoice/order slip for the purchased product
- The type of operating system and hardware in use
- Details regarding your operating environment; such as available memory, disk space, your version of R:BASE, local area network, special drivers, related database structures, application files, and other files that are used or accessed by your application

All provide information will be used to better assist you.

R:BASE Technologies has a number of different services available for R:BASE products. As a registered user, you will receive information about new features for R:BASE and other R:BASE Technologies products. Please remember to register your software. https://www.rbase.com/register/

Copyright © 1982-2025 R:BASE Technologies, Inc.

Part XI

11 Useful Resources

. R:BASE Home Page:	https://www.rbase.com
. Up-to-Date R:BASE Updates:	https://www.rbaseupdates.com
. Current Product Details and Documentation:	https://www.rbase.com/rbg11
. Support Home Page:	https://www.rbase.com/support
. Product Registration:	https://www.rbase.com/register
. Official R:BASE Facebook Page:	https://www.facebook.com/rbase
. Sample Applications:	https://www.razzak.com/sampleapplications
. Technical Documents (From the Edge):	https://www.razzak.com/fte
. Education and Training:	https://www.rbase.com/training
. Product News:	https://www.rbase.com/news
. Upcoming Events:	https://www.rbase.com/events
. R:BASE Online Help Manual:	https://www.rbase.com/support/rsyntax
. Form Properties Documentation:	https://www.rbase.com/support/FormProperties.pdf
. R:BASE Beginners Tutorial:	https://www.rbase.com/support/rtutorial
. R:BASE Solutions (Vertical Market Applications):	https://www.rbase.com/products/rbasesolutions

Part XII

12 Feedback

Suggestions and Enhancement Requests:

From time to time, everyone comes up with an idea for something they'd like a software product to do differently.

If you come across an idea that you think might make a nice enhancement, your input is always welcome.

Please submit your suggestion and/or enhancement request to the R:BASE Developers' Corner Crew (R:DCC) and describe what you think might make an ideal enhancement. In R:BASE, the R:DCC Client is fully integrated to communicate with the R:BASE development team. From the main menu bar, choose "Help" > "R:DCC Client". If you do not have a login profile, select "New User" to create one.

If you have a sample you wish to provide, have the files prepared within a zip archive prior to initiating the request. You will be prompted to upload any attachments during the submission process.

Unless additional information is needed, you will not receive a direct response. You can periodically check the status of your submitted enhancement request.

If you are experiencing any difficulties with the R:DCC Client, please send an e-mail to rdcc@rbase.com.

Reporting Bugs:

If you experience something you think might be a bug, please report it to the R:BASE Developers' Corner Crew. In R:BASE, the R:DCC Client is fully integrated to communicate with the R:BASE development team. From the main menu bar, choose "Help" > "R:DCC Client". If you do not have a login profile, select "New User" to create one.

You will need to describe:

- What you did, what happened, and what you expected to happen
- The product version and build
- Any error message displayed
- The operating system in use
- Anything else you think might be relevant

If you have a sample you wish to provide, have the files prepared within a zip archive prior to initiating the bug report. You will be prompted to upload any attachments during the submission process.

Unless additional information is needed, you will not receive a direct response. You can periodically check the status of your submitted bug.

If you are experiencing any difficulties with the R:DCC Client, please send an e-mail to rdcc@rbase.com.

Index

Copyright © 1982-2025 R:BASE Technologies, Inc.

Notes