Chapter 1 • SQL statement overview

The basic SQL unit of execution is the statement. SQL program statements allow you to connect to Butler SQL database servers, manipulate data, control program flow, and retrieve data and descriptive information.

About this guide

This guide is a reference to the SQL dialect supported by Butler SQL. Statement functional groups are presented, followed by a complete description of each statement. The statements are listed in alphabetical order.

Statement groups

SQL statements can be divided into the following 4 function groups:

- **Data-manipulation statements** offer complete, SQL-based data access to host databases and files. For example, the UPDATE statement modifies database contents.
- **Program-control statements** support testing, looping, and procedure calls within a SQL program. For example, the WHILE statement provides repeated execution of a SQL loop.
- **Output-control statements** generate output messages from the SQL program for processing by the client application.
- **Database entity-manipulation statements** create and delete databases, tables, and indices.

The following sections describe each statement group in detail and list statements that belong to each group.

Data-manipulation statements

Butler SQL's data-manipulation facilities provide uniform access to databases and files for data retrieval and update. The data-manipulation statements are based on SQL, which has become a *de facto* standard for relational database access. An ANSI standard for SQL has been defined; however, many variations in the SQL dialects of current DBMS systems, and large areas of the language, are left as implementation choices in the current ANSI specification. The variations include the areas of database organization, data types, catalog structures, dynamic query support, buffer management, indicator variable handling, and error codes.

The Butler SQL data-manipulation statements adhere to ANSI standard statement syntax wherever possible. Where the ANSI standard is not implemented, Butler SQL follows IBM's DB2 standard if possible. For example, Butler SQL error codes are compatible with DB2 codes.

The SQL statements that provide the data-manipulation facilities of database access, data retrieval, data update, and transaction management are summarized by function:

Database-access statements

BACKUP DATABASE	Provides a method of on-line back-ups
OPEN DATABASE	Prepares a database for access
CLOSE DATABASE	Terminates access to a database
USE DATABASE	Establishes the default database
OPEN TABLE	Prepares a table for access
CLOSE TABLE	Terminates access to a table
USE LOCATION	Sets the default host directory

RESTORE DATABASE	Restores all completed transactions for every database in the specified transaction journal.

Data-retrieval statements

SELECT	Forms a rowset of retrieved data
FETCH	Retrieves a row from a rowset
DESELECT	Ends access to a rowset

Data-update statements

DELETE (searched)	Deletes rows from a table
DELETE (positioned)	Deletes the current row of a cursor
UPDATE (searched)	Modifies rows in a table
UPDATE (positioned)	Modifies the current row of a cursor

Transaction-management statements

COMMIT	Completes the current transaction
ROLLBACK	Aborts the current transaction

Information-statements

DESCRIBE DATABASES	Describes available databases
DESCRIBE OPEN DATABASES	Describes active databases
DESCRIBE TABLES	Describes tables of a database
DESCRIBE OPEN TABLES	Describes explicitly-opened tables
DESCRIBE COLUMNS	Describes columns of a table
DESCRIBE INDEXES	Describes indexes of a table
DESCRIBE KEYS	Describes components of an index

When SQL executes a data-manipulation statement, it automatically sets the value of the system variable **$sqlcode** to indicate the completion status of the statement. If the ERRORCTL statement has specified processing of errors within the SQL program itself, the value of **$sqlcode** can be examined in subsequent SQL statements to detect and recover from error conditions.

Whether selecting data from a database or requesting descriptions of the data through a DESCRIBE statement, DAL/DAM applications must use output-control statements to place the retrieved data information in the output stream (see the section “Output-control statements” on page 5 for a list of statements that belong to this group).

Program-control statements

Program-control statements support decision making and structured programming in SQL. Using these statements, a SQL program can:

- declare and assign values to a variable,
- unconditionally change its execution flow,
- test a condition and branch on the results,
- repetitively perform a sequence of statements, and
- define and call procedures for frequently performed tasks.

The program-control statements enhance the capability of Butler SQL by allowing parallel processing between the client application and a SQL program executing on its behalf. By passing along an entire SQL program for execution rather than executing each statement individually, the client application can relinquish detailed supervision of a connectivity task (such as row-by-row data fetching) and concentrate on other aspects of the application. In addition, when the Butler server performs data manipulation, network traffic is reduced to the data actually required by the client application.

The SQL program-control statements are summarized by function:

Variable-support statements

DECLARE	Variable declaration
UNDECLARE	Variable redeclaration
SET	Variable assignment

Conditional execution and branching statements

IF	Conditional execution
SWITCH	Multipath branch
GOTO	Unconditional branch

Iteration statements	
WHILE	Looping with pretest
DO	Looping with post-test
FOR	Iteration with pretest
FOR EACH	Cursor-based iteration
BREAK	Premature break of loop/iteration
CONTINUE	Premature repeat of loop/iteration

Procedure call statements	
PROCEDURE	Procedure definition
CALL	Procedure call
RETURN	Return from called procedure

Program Execution statements	
EXECUTE FILE	Loads and executes SQL statements from a file
EXECUTE FROM	Executes SQL statements constructed from a string literal

Output-control statements

The output-control statements are generally applicable to DAL/DAM connections only. You may use them from ODBC, although the ODBC API doesn't require their usage.

A SQL program passes data back to its client application by explicitly generating output messages for the client. The SQL output-control statements generate these output messages and control their format. The client application retrieves the output through SQL API function calls. The DAL/DAM also provides function calls that describe the types and sizes of the data items in the output message that is currently being processed by the client application.

Client applications vary widely in the types of data that they naturally support. A spreadsheet, for example, may use integer and floating-point data but lack packed-decimal support. A word processor or HyperCard® application may be capable of working

only with text data. To support this range of needs, Butler's SQL supports output data mapping, which automatically converts the data types generated by SQL output-control statements into types acceptable to the client application. In all cases, the returned data is in the native data types of the client system.

When execution of a SQL program fragment is complete, the SQL runtime environment automatically generates a completion-status message for the client application, indicating either successful completion or an error condition. Data Access Language also provides an output-control statement to control the generation of error-status messages by SQL data-manipulation statements.

The SQL output-control statements are as follows:

PRINT	Outputs a row of data items
PRINTROW	Outputs a single row of a rowset
PRINTALL	Outputs all rows of a rowset
PRINTF	Outputs a formatted string
PRINTINFO	Outputs a description of a rowset
PRINTCTL	Controls output data mapping
ERRORCTL	Controls SQL error handling
ERROR	Generates SQL errors
WARNCTL	Controls SQL warning processing

Database entity-manipulation statements

Database entity-manipulation statements give developers the ability to create databases from scratch from within SQL programs. This capability is useful for providing custom front-ends for database creation and for automatic generation of required databases when installing a Butler SQL client application.

Here is a summary of the statements within this group:

Database entity-creation statements

CREATE DATABASE	Creates a database
CREATE TABLE	Creates a table
CREATE INDEX	Creates an index

Database entity-deletion statements

ALTER TABLE	Allows users to add, modify, or drop columns from a table.
DROP DATABASE	Deletes a database
DROP TABLE	Deletes a table
DROP INDEX	Deletes an index

Chapter 2 • Statement Reference

This chapter describes Butler SQL's SQL dialect. The statements are listed alphabetically. Statement definitions contain some or all of the following: *Syntax*, *Parameters*, *Notes*, and *Examples*.

ALTER TABLE

The Alter Table statement allows users to add, modify, or drop columns.

Category Database-access statements.

Syntax `ALTER TABLE` *base-table base-table-alteration*

base-table a string or alphanumeric literal

base-table-alteration

column-alteration-action

column-alteration-action

add-column-action | alter-column-action drop-column-action

add-column-action `ADD [ COLUMN ]` column-definition

alter-column-action `{ALTER | MODIFY} [COLUMN]` column

{ data-type | `[SET]` default-definition | `[SET]` name "newcolumnname" | `DROP DEFAULT }`

drop-column-action `DROP [COLUMN]` column `{RESTRICT |CASCADE}`

column-definition column { data-type | domain }

[default-definition]

		[column-constraint-definition-list]
	default-defintion	`DEFAULT` { literal | niladic-function | `NULL`}
	niladic-function	`USER`
		`CURRENT_USER`
		`SESSION_USER`
		`SYSTEM_USER`
		`CURRENT_DATE`
		`CURRENT_TIME`
		`CURRENT_TIMESTAMP`
Parameters	*base-table*	The base-table is a string literal reference which represents the name of a table in the database that physically exists, therefore an alter statement cannot be done on a view or virtual table.
	add-column-action	
	column-definition	The ALTER TABLE statement allows you to add, (modify | alter) or drop columns. When adding columns you specify a column definition which you specify its datatype, default definition and constraint. The data-type encompasses not only the type but also any appropriate length or scale.
	column { data-type | domain }	
	[default-definition]	
	default-definition	The default definition specifies an explicit default value for the column. If no default clause is specified, and the column is defined on a domain, then if

the domain has an explicit default its value is used (otherwise the default is null).

niladic-function The niladic functions are a set of functions that can be used to define the initial state of a column.

The function is not evaluated until a row is inserted into the table containing the column.

- `USER` current authorized username which executes the INSERT statement
- `CURRENT_USER` current authorized username
- `SESSION_USER` current authorized username
- `CURRENT_DATE` date when column data inserted
- `CURRENT_TIME` time when column data inserted
- `CURRENT_TIMESTAMP` timestamp when column data inserted

alter-column-action The alter column action statements allow you to set or drop a column's default-definitions.

The alter column statement also allows for you to specify a new data-type for the column, subsitute the keyword MODIFY for ALTER in the ALTER COLUMN.

There is only one default definition allowed per column so dropping a default is done using the DROP DEFAULT key word.

Examples Drop the column named "PhoneNumber" from the Customer table:

```
alter table Customer drop column PhoneNumber;
```

Change the name of column "col2" to "Zip_Code":

```
alter table Customer modify column col2 set name
Zip_Code;
```

Change the default value of the TotalSales column to 1000.00:

```
alter table OrderSummary modify TotalSales set default
'1000.00';
```

Add a VarChar column named "Notes" to a table; set the length of the column to 2000:

```
alter table Customer add column notes varchar 2000;
```

Add a time column named "LastContactDate" to the Customer table:

```
alter table Customer add LastContactDate time;
```

Notes

Use of the ALTER TABLE command is not recorded in the database journal. As a result, you can not use the ROLLBACK command to undo the change made with this command. Additionally, you do not need to use the COMMIT command to make the change permanent.

Because the ALTER command is not recorded in the journal, you need to exercise caution when recovering a database using journal files. The ALTER command will not be executed as part of the recover, which may cause restoration of journal entries that depend on the altered column to fail. For the best security and recoverability, you should backup your database and detach the journal file before using the ALTER command.

BACKUP DATABASE

The BACKUP statement provides a method of performing on-line backups with no interuption to the connected users on the server. The backup is really a database file copy to the specified destination folder with the intent that this folder can be used for tape archival or simply on-line backup. Only valid database files can be duplicated in this manner.

Category Database-access statements.

Syntax

```
BACKUP DATABASE <database name>
[IN LOCATION <folder path>]
TO <folder path>
[WITH {JOURNAL, ACTIVITYLOG}]
[TIMEOUT <timeout seconds>]
```

Parameters

database name	name of the database.
folder path	any valid Macintosh folder path string expression specified. The folder path must end with a colon. If the specified destination folder doesn't exist, it will be created by the server.
timeout seconds	default timeout period is 0, which represents an infinite timeout period. Other timeout periods are specified as an integral expression in seconds.

Examples

```
backup database dal_demo in location "hdgash:data-
bases:" to "hdGash:backup:" with journal, activitylog
timeout 200000;
```

Notes Outstanding BACKUP commands are cancelled by the server if the client issuing the command is logged off the server either through a communications disconnect or through the server's "Terminate User" menu command.

The user must ensure that there is enough free space available on the server to perform the backup; otherwise, the backup will generate an error.

If transaction processing is enabled, the backup will not begin until all clients currently accessing the database have closed all transactions. All clients who have issued insert, update, or delete commands must perform a commit or rollback before the backup will begin.

If a timeout value is specified and open transactions exist at the completion of the timeout period, the backup will be cancelled and the server will return an error. Until the timeout occurs or the backup is completed, no new transactions will be permitted by any client.

However, clients may continue to read the database until held. Note that activity logs which are detached include the notice of the backup and detach, in addition to a backup notice being inserted to the new activity log.

If access control is enabled, only the database administrator can issue the command to perform a backup. If no database administrator is assigned, only clients with full access to the database will be able to issue the backup command.

The WITH clause can be used to detach and save database transaction journals and activity logs. This performs a detach of the current journal/log, then moves the file from the Database Journals/ Activity Logs folder to the same destination folder as the database backup.

If this option is used, a record of the backup and detach is added to the activity log.

BREAK

Use the BREAK statement to jump out of a loop structure such as a WHILE, DO, FOR, FOR EACH, or SWITCH statement. Execution continues with the first statement immediately following the body of the loop structure in which the BREAK statement appears.

Category Iteration statements.

Syntax `BREAK;`

Parameters None.

Example Add year-to-date sales to the sum of the current region:

```
switch (region_num) {
case 1 :
  set east_sum = east_sum + sales_ytd;
  break;
case 2 :
  set mid_sum = mid_sum + sales_ytd;
  break;
case 3 :
  set west_sum = west_sum + sales_ytd;
  break;
default :
  print " Bad Region ", region;
}
(after a break, execution begins here)
```

In this example, if BREAK statements were not used, all statements after the current case would be executed through the end of the SWITCH statement.

Notes When BREAK is used within a nested loop or SWITCH statement, only the processing of the innermost loop (or SWITCH) is exited. There is no way to cause a multilevel break using a single statement.

CALL

The CALL statement calls a SQL procedure. Supporting the use of modular SQL procedures, the CALL statement provides both a procedure call and a function call facility. The parameters of the statement specify the procedure to be called, the values passed to the procedure as its arguments, and the variables that receive the return values from the procedure.

Category Procedure call statements.

Syntax

```
[ CALL ] procname (argument_list);
```

If the procedure is a function (returning a single value), then the following function call form of procedure call can be used anywhere an expression is allowed:

```
procname (argument_list)
```

Parameters

procname	The name of the procedure to be called.
argument_list	A comma-separated list of scalar expressions whose values are passed as arguments. The number and type of arguments must agree with the procedure declaration.
variable_list	A comma-separated list of variables that receive the return values from the procedure. The number and type of variables must agree with the procedure declaration.

Example Assume a procedure "maxsales" which, given an employee number, returns the maximum sales by that employee for the last four quarters. To determine the sum of the maximum sales for all employees:

```
declare money sum, maxval;
set sum = 0;
select rep_nr from staff;
for each {
```

```
  call maxsales(rep_nr) returning maxval;
  set sum = sum + maxval;
}
```

Note that because “maxsales” returns a single MONEY value, the call statement could also be expressed as

```
set maxval = maxsales(rep_nr);
```

Notes

The keyword CALL is optional.

Procedure calls may be nested deeply. Depth of nesting is subject only to resource constraints on the SQL run-time environment.

Recursive procedure calls are not supported.

CLOSE DATABASE

The CLOSE DATABASE statement closes access to a database opened previously with the OPEN DATABASE statement.

Category Database-access statements.

Syntax `CLOSE DATABASE [`*dbalias*` ] ;`

Parameter

dbalias	The alias of the database to be closed. If *dbalias* is omitted, the default database is closed.

Examples Close the default database:

```
close database;
```

Close a database using an implicit alias:

```
open database "shakespeare";
. . .
close database shakespeare;
```

Close a database opened with an alias:

```
open database "shakespeare" alias bill;
. . .
close database bill;
```

Notes Closing a database ends any open transactions. Whether transactions are committed or rolled back depends on the server's auto-commit/auto-rollback setting.

CLOSE TABLE

The CLOSE TABLE statement closes access to a table opened previously with the OPEN TABLE statement.

Category Database-access statements.

Syntax `CLOSE TABLE` *tblref* `;`

Parameters

tblref — A reference to a table previously opened by an OPEN TABLE statement.

Examples Close a table in the default database:

```
close table orders;
```

Close a table in the database named "joe":

```
close table joe!orders;
```

Notes Closing a table with the CLOSE TABLE statement does not prevent subsequent access by a SELECT, INSERT, DELETE, or UPDATE statement. The automatic SQL table opening/closing behavior described in the OPEN TABLE statement provides access to the table, but with the possibility of much higher overhead.

Closing a table previously opened with EXCLUSIVE access effectively unlocks the table, permitting access by other users. If the table was updated, it may remain locked until a COMMIT or ROLLBACK statement is executed.

COMMIT

The COMMIT statement signals successful completion of the current transaction. Modifications to the host database made during the transaction are committed, and a new transaction begins.

Category Transaction-management statements.

Syntax `COMMIT [ WORK ] [ <FOR>` *dbbrand* `] ;`

Parameter

dbbrand The DBMS brand whose current transaction is to be committed. If *dbbrand* is omitted, the default brand is assumed.

Example Update the database to reflect the transfer of "Jones" to office number 301, and commit the change to the database:

```
update staff set office = 301 where rep_name =
"Jones";
commit;
```

Notes A transaction begins implicitly when a database is first opened; a new transaction begins upon completion of each COMMIT or ROLLBACK statement.

Butler SQL does not provide transaction integrity across multiple DBMS brands or host systems. The COMMIT/ROLLBACK mechanism can be used only to guarantee the integrity of modifications made to a single DBMS brand during a transaction.

The dbbrand parameter is not applicable when using ODBC to communicate with Butler SQL. The parameter is optional when using DAL/DAM.

The Butler server can be set to automatically COMMIT or ROLLBACK a transaction when a client application disconnects or closes the database before issuing the COMMIT or ROLLBACK command. See the Butler SQL User's Guide for more information on setting preferences for Butler SQL.

CONTINUE

The CONTINUE statement interrupts the flow of the body of a WHILE, DO, FOR, or FOR EACH statement and causes the remainder of the current iteration of the WHILE, DO, FOR, or FOR EACH statement to be skipped. Execution continues with the loop control expression (for WHILE or DO), with the loop re-initializer statement (for FOR), or with fetching of the next row (for FOR EACH).

Category Iteration statements.

Syntax

```
CONTINUE;
```

Parameters None.

Example Print all customers owing $50,000 or more; skip over customers owing less than $50,000:

```
declare cursor x;
select cust_num, cust_name, receivable from customer
into x for scrolling;
for each x {
  if (receivable < 50000)
    continue;
  else
    print cust_num, cust_name, receivable;
}
```

Notes When CONTINUE appears within nested loops or SWITCH statements, the remainder of the processing in the loop (or SWITCH) continuing the SWITCH statement is skipped. There is no way to use a single statement to skip out of multiple loops.

CREATE DATABASE

The CREATE DATABASE statement is used to create databases.

Category Entity-creation statements.

Syntax

```
CREATE [ dbbrand ] DATABASE "dbname"
[ < IN > LOCATION "dbloc" ]
[ < AS > USER "user" [ < WITH > PASSWORD "password" ] ] ;
```

Parameters

dbbrand	An identifier for the DBMS brand in which to create the database. If *dbbrand* is omitted, the default brand is assumed.
dbname	A string literal specifying the name for the new database. This parameter can include letters, numbers, and the underscore character ("_"), and can be up to 31 characters in length.
dbloc	A string literal specifying a pathname indicating where to save the new database *dbname*. If this parameter is omitted, the database is created in the default location (the server's Public Databases folder, if no other has been specified with a USE LOCATION statement).
user	A string literal specifying the username with which to create the database. If this parameter is omitted, the current username is used.
password	A string literal specifying the password for the username specified, if any.

Example Create a database called "contacts" in the "HD100:Fred's Databases" folder:

```
CREATE DATABASE "contacts" IN LOCATION "HD100:Fred's
Databases";
```

Notes The CREATE DATABASE statement, along with other entity creation statements, is not recorded in the database journal. The command is not affected by COMMIT or ROLLBACK statements.

CREATE INDEX

The CREATE INDEX statement is used to create an index for a table.

Category Entity-creation statements.

Syntax

```
CREATE [ UNIQUE ] INDEX "indexname" ON "tablename"
( colname [ < ASC > | < DESC > ] , … ) ;
```

Parameters

tablename	A string expression specifying the table from a currently open database to index. If there is no current database, or if *tablename* does not exist in it, this statement returns an error.
colname	An identifier specifying one or more column names from *tablename* to include in the new index.
indexname	A string literal specifying the name of the index to create. It can include letters, numbers, and the underscore character, and can be up to 31 characters in length. An error is returned if *indexname* already exists in the database.
UNIQUE	Specifies that the index will enforce unique keys; that is, attempts to insert or update the table with values which result in a duplicate key will result in an error.
ASC | DESC	ASC specifies that *colname* will be ordered from lowest to highest values in the index (ascending). DESC specifies that *colname* will be ordered from highest to lowest values in the index (descending). If this parameter is omitted, ASC is assumed.

Example Create an index called “names_by_last” for the “names” table including the “first_name” and “last_name” columns, enforcing unique keys:

```
CREATE UNIQUE INDEX "names_by_last" ON "names"
( last_name , first_name);
```

Notes The CREATE INDEX statement, along with other entity creation statements, is not recorded in the database journal. The command is not affected by COMMIT or ROLLBACK statements

CREATE TABLE

The CREATE TABLE statement is used to create tables within a database.

Category Entity-creation statements.

Syntax

```
CREATE TABLE "tablename"
( colname coltype [ ( length [ , numdecs ) ] , … ) ;
```

Parameters

tablename	A string literal specifying the name for the new table. It can include letters, numbers, and the underscore character, and can be up to 31 characters in length. The table is added to the currently open database; if none exists, this statement returns an error.
colname	An identifier specifying the name of a column name to be added to the new table. It can include letters, numbers, and the underscore character, and can be up to 31 characters in length.
coltype	An identifier specifying the data type for the column *colname.* This parameter can be any one of the following: BOOLEAN, SMINT, INTEGER, SMFLOAT, FLOAT, MONEY, DECIMAL, NUMBER, DATE, TIME, TIMESTAMP, CHAR, VARCHAR, VARBIN, LONGCHAR, PICTURE, SOUND, ICON, MOVIE, or DOCUMENT. If NUMBER is specified in *coltype*, the column is assigned an appropriate type (integer, smfloat, float, etc.) dependent on the values specified for *length* and *numdecs*.

length	An integer literal specifying the length of the column *colname*. This parameter is ignored if specified for a SMINT, INTEGER, SMFLOAT, FLOAT, MONEY, BOOLEAN, DATE, TIME, or TIMESTAMP *coltype*.
numdecs	An integer literal specifying the number of decimal places for the column *colname*. This parameter is ignored for a BOOLEAN, SMINT, INTEGER, MONEY, DATE, CHAR, VARCHAR, VARBIN, LONGCHAR, PICTURE, SOUND, ICON, MOVIE, DOCUMENT, FLOAT, SHORT FLOAT, BIGINT, TINYINT *coltype*.

Example Create a table called “names” in the “contacts” database, with columns titled “first_name” and “last_name”:

```
OPEN DATABASE "contacts"
CREATE TABLE "names" ( last_name CHAR ( 20 ) ,
      first_name CHAR ( 30 ) );
```

Notes The CREATE TABLE statement, along with other entity creation statements, is not recorded in the database journal. The command is not affected by COMMIT or ROLLBACK statements

DECLARE

The DECLARE statement is used to specify the data type of one or more SQL variables. Variables must be declared before they can be referenced by a SQL statement. A DECLARE statement appearing within a procedure definition specifies for that procedure a local variable, whose scope lies within the procedure definition. A DECLARE statement appearing in the outer block specifies an outer block variable, whose scope is limited to the outer block.

Category Program-control statements.

Syntax `[ DECLARE ]` *datatype variable_name* `[ =` *initial_val* `] ;`

Parameters

datatype	A keyword of a valid SQL data type.
variable_name	The name of the declared variable.
initial_val	An expression specifying an initial value for the variable.

Example Declare two money variables, a variable-length character string variable, and an integer:

```
declare money quota, sales;
declare varchar city;
integer staffcount = 7;
```

Notes The keyword DECLARE is optional.

Outer-block variable names and procedure names share the same name space. A DECLARE statement for an outer block variable redeclares automatically any procedure or outer-block variable with the same name.

If an initial value is present, an outer-block variable is initialized once at declaration; a local variable is re-initialized on each procedure entry.

Local variables lose their value on return from a procedure; in particular, rowsets associated with local CURSOR variables are deselected.

If an initial value is not specified, and used in an expression before it is assigned a value, the variable is uninitialized and produces an error.

A DECLARE statement supersedes automatically any previous declaration for a variable of the same name within the scope of the block.

DELETE (positioned)

The positioned DELETE statement deletes the current row of a cursor from the table referenced by the cursor.

Category Data-update statements.

Syntax `DELETE FROM` *tblref* `WHERE CURRENT OF` *cursor* `;`

Parameters

tblref	The table containing the rows to be deleted.
cursor	The cursor whose current row is to be deleted.

Example Remove the salesperson indicated by the current row of cursor "x":

```
delete from staff where current of x;
```

Notes The cursor must specify an 'updatable' table or view, and must specify a rowset created by a SELECT statement with a FOR UPDATE clause.

DELETE (searched)

The searched DELETE statement deletes rows from a table. The rows deleted are determined by the statement's WHERE clause.

Category Data-update statements.

Syntax `DELETE FROM` *tblref* `[ WHERE` *where_condition* `] ;`

Parameters

tblref	The table containing the rows to be deleted.
where_condition	The search condition that identifies the rows to be deleted. If *where_ condition* is omitted, all rows are deleted.

Examples Drop all staff in office 102:

```
delete from staff where office_nr = 102;
```

Purge all the data from the "orders" table:

```
delete from orders;
```

Notes WHERE conditions are described in the section "Search Conditions" in the Butler SQL Programmer's Guide.

DESCRIBE COLUMNS

The DESCRIBE COLUMNS statement retrieves column information for the designated table from the database. The information is returned in the form of a table, with each row describing the attributes of a column.

Category Information statements.

Syntax `DESCRIBE COLUMNS <OF> tblref [ INTO cursor ] ;`

Parameters

tblref A table reference identifying the table whose columns are to be described.

cursor A CURSOR variable to receive the rowset.

Results The resulting rowset has one row for each column in the specified table. Each row contains the 14 columns of information shown in Table 1.

Table 1: DESCRIBE COLUMNS rowset

Col #	Data type	Name	Information
1	SMINT	colnr	Column number
2	SMINT	level	Column level number
3	VARCHAR (31)	name	Column name
4	SMINT	type	Column data type (see Chapter 2 • Data Types in the Programmer's Guide for more information)
5	SMINT	len	Column length in bytes
6	SMINT	places	Column scale
7	BOOLEAN	nullsok	Nulls allowed

Table 1: DESCRIBE COLUMNS rowset

Col #	Data type	Name	Information
8	BOOLEAN	groupcol	Group column
9	SMINT	parentnr	Parent column number
10	SMINT	occurs	Number of occurrences
11	SMINT	occdep	Occurs depending on column
12	BOOLEAN	updtok	If the column can be updated?
13	VARCHAR (31)	title	Column title
14	VARCHAR (255)	remarks	Column remarks

Example Describe all the columns of table "joe.sales":

```
describe columns of joe.sales;
```

Notes The OPEN DATABASE statement must open the database containing the table specified by *tblref* before the columns in any of its tables can be described.

It is not necessary for the table specified by *tblref* to have been opened with an OPEN TABLE statement.

DESCRIBE DATABASES

The DESCRIBE DATABASES statement creates a rowset describing the available databases for a particular DBMS brand.

Category Information statements.

Syntax

```
DESCRIBE [dbbrand] DATABASES [ <IN> LOCATION dbloc ]
[ INTO [ cursor ];
```

Parameters

dbbrand	An identifier naming the DBMS brand whose databases are to be described. If *dbbrand* is omitted, Butler SQL's brand (RMR) is assumed.
dbloc	The pathname to a folder on the server continuing the databases you want to describe. If *dbloc* is omitted, the default database location, the Public Database folder, is assumed.
cursor	A CURSOR variable to receive the rowset.

Result The resulting rowset has one row for each database of the specified brand in the specified location. Each row contains the column of information shown in Table 2.

Table 2: DESCRIBE DATABASES rowset

Col #	Data type	Name	Information
1	VARCHAR[255]	name	Database name

Example Create a rowset that describes all databases for the DBMS:

```
describe  databases;
printall;
```

Notes The dbbrand parameter is not applicable when using ODBC to connect to Butler SQL.

DESCRIBE INDEXES

The DESCRIBE INDEXES statement creates a rowset describing a table's indices.

Category Information statements.

Syntax `DESCRIBE INDEXES OF` *tblref* `[ INTO` *cursor* `];`

Parameters

tblref	A table reference identifying the table whose indices are to be described.
cursor	A CURSOR variable to receive the rowset.

Results The resulting rowset has one row for each index associated with the specified table. Each row contains the five columns of information shown in Table 3.

Table 3: DESCRIBE INDICES rowset

Col #	Data type	Name	Information
1	VARCHAR[31]	indname	Index name
2	BOOLEAN	unique	Unique keys
3	INTEGER	numkeyed	Number of columns keyed
4	INTEGER	numkeys	Number of keys
5	VARCHAR[31]	primekey	Primary keyed column name

Example Describe the indexes of the "customer" table:

```
DESCRIBE INDEXES OF customer;
```

Notes The OPEN DATABASE statement must open the database containing the table specified by *tblref* before the table's indexes can be described.

It is not necessary for the table specified by *tableref* to have been opened with an OPEN TABLE statement.

The keyword OF is mandatory.

An alternate form of this statement is :

```
DESCRIBE INDICES OF tblref  [ INTO cursor ];
```

DESCRIBE KEYS

The DESCRIBE KEYS statement creates a rowset describing the keys comprising the specified index.

Category Information statements.

Syntax `DESCRIBE KEYS OF` *indexName* `[INTO` *cursor*`];`

Parameters

indexName	An identifier specifying the the index whose keys are to be described.
cursor	A CURSOR variable to receive the rowset.

Results The resulting rowset has one row for each key component of the specified index. Each row contains the seven columns of information shown in Table 4.

Table 4: DESCRIBE KEYES rowset

Col #	Data type	Name	Information
1	VARCHAR[31]	keyname	Keyed column name
2	VARCHAR[31]	keytitle	Keyed column title
3	INTEGER	size	Key length (bytes)
4	SMINT	keytype	Keyed column data type

Table 4: DESCRIBE KEYES rowset

Col #	Data type	Name	Information
5	CHAR[4]	sort_order	Sort order (ASC or DESC)
6	BOOLEAN	case_sens	Case sensitive ordering
7	CHAR[1]	comparetype	(A)SCII or (L)ocalized (international) ordering

Example

```
DESCRIBE KEYS OF CustomersByCust_Num;
```

Notes

The OPEN DATABASE statement must open the database containing the index specified by indexName before the index's keys can be described.

The keyword OF is mandatory.

DESCRIBE OPEN DATABASES

The DESCRIBE OPEN DATABASES statement creates a rowset describing the current open database(s).

Category Information statements.

Syntax `DESCRIBE OPEN DATABASES [ INTO` *cursor* `] ;`

Parameter

cursor	A CURSOR variable to receive the rowset.

Results The resulting rowset has one row for each database that is currently open. The first database described in the rowset is the current default database. Each row contains the six columns of information shown in Table 5.

Table 5: DESCRIBE OPEN DATABASES rowset

Col #	Data type	Name	Information
1	SMINT	order	Sequence number
2	VARCHAR[31]	alias	Database alias
3	VARCHAR[31]	brand	DBMS brand (For Butler SQL, this is always “RMR”)
4	SMINT	shrmode	Sharing mode
5	SMINT	updmode	Update mode
6	VARCHAR[31]	owner	current owner

Example Create a rowset called “opendb” that describes all currently open databases:

```
describe open databases into opendb;
printall;
```

Notes Even though the data resulting from this command does not come from a table in the database, Butler SQL creates an Extract cursor for the rowset. The number of rows in the rowset can be determined by checking the **$rowcnt** system variable.

The sharing mode (shrmode) is reported as 1 for SHARED, 2 for PROTECTED, and 3 for EXCLUSIVE mode.

The update mode (updmode) is reported as 1 for READONLY and 2 for UPDATE.

DESCRIBE OPEN TABLES

The DESCRIBE OPEN TABLES statement creates a rowset describing the tables that have been opened explicitly through the OPEN TABLE statement and implicitly with a SELECT statement.

Category Information statements.

Syntax `DESCRIBE OPEN TABLES [ INTO` *cursor* `] ;`

Parameters

cursor	A CURSOR variable to hold the resulting rowset.

Results The rowset returned contains the following information:

Col #	Name	Information
1	order	Numerical order of open tables (1 means most recent).
2	name	The name of the open table.
3	dbalias	The alias for the database.
4	shrmode	The shared mode specified in an OPEN TABLE statement.
5	updmode	The update mode specified in an OPEN TABLE statement.
6	autoopen	$TRUE if table was opened implicitly during a SELECT statement; $FALSE if the table was opened with a OPEN TABLE statement.

Example

```
describe open tables;
for each print ->name;/*instead of printall; */
```

This example creates a rowset that describes all the currently opened tables in the default database and prints their names only.

Notes For most relational (SQL) DBMS's, the DESCRIBE OPEN TABLES and OPEN TABLE statements are not very useful. It is included for compatibility with the DAL/DAM SQL syntax.

DESCRIBE TABLES

The DESCRIBE TABLES statement creates a rowset describing the tables of the specified database.

Category Information statements.

Syntax `DESCRIBE TABLES [ <OF> dbalias ] [ INTO cursor ] ;`

Parameters

dbalias The alias of the database whose tables are to be described. If *dbalias* is omitted, the default database is assumed.

cursor A CURSOR variable to receive the rowset.

Results The resulting rowset has one row for each client-accessible table and view in the specified database. Each row contains the ten columns of information shown in Table 6.

Table 6: DESCRIBE TABLES rowset

Col #	Data type	Name	Information
1	VARCHAR[255]	name	Table name
2	VARCHAR[1]	type	Table (T) or view (V)
3	BOOLEAN	ordered	Is table ordered?
4	SMINT	colcnt	Column count
5	INTEGER	rowcnt	Row count
6	SMINT	parentcnt	Parent count
7	SMINT	childcnt	Child count

Table 6: DESCRIBE TABLES rowset

Col #	Data type	Name	Information
8	VARCHAR[31]	title	Table title
9	VARCHAR[255]	remarks	Remarks
10	VARCHAR[255]	owner	Table owner

Example Create a rowset that describes all the tables in the "flywheel_db" database:

```
describe tables of flywheel_db;
```

Notes The OPEN DATABASE statement must open the database specified by *dbalias* (or the default database if no *dbalias* is specified) before its tables can be described.

The following code fragment is useful if you want only part, but not all, of the information from a DESCRIBE TABLES statement. This code fragment has faster data-transmission times.

```
describe tables;
  for each print —> name;  /* instead of printall; */
```

In this example, instead of using PRINTALL, which will print all the information returned in the rowset resulting from the first statement, you can use the combination of statements described previously to select the field of interest to you, and print only the contents of that field.

Since DESCRIBE TABLES returns "owner table" in the name field, often you will not need the rest of the information about the tables. Also note, the "owner" portion is omitted in the returned name field when it is not necessary to access the table, such as when the current user is the table owner.

In addition, users who use the DESCRIBE TABLES statement (or any of the DESCRIBE statements) followed by PRINTALL should code their programs to allow for additional columns of data that may be added in the future, being sure to receive

data items (and possibly discard them) until the end of each row is reached. This will ensure maximum compatibility with future releases.

DESELECT

The DESELECT statement ends access to a rowset. Butler SQL discards the specified rowset and reclaims the rowset's resources. The cursor associated with the rowset becomes invalid.

Category Data-retrieval statements.

Syntax `DESELECT [` *cursor* `] ;`

Parameter

cursor — The name of a CURSOR variable containing the cursor created by the SELECT statement that created the rowset. If *cursor* is omitted, the default cursor (**$cursor**) is assumed.

Example Find the sales representative with the highest year-to-date sales:

```
select last_name, ytd_sales from staff order by
ytd_sales;
fetch;
print last_name, ytd_sales;
deselect;
```

Note Butler SQL supports scroll cursors which allow arbitrary motion through a rowset. Due to this support, the rowset cannot be discarded automatically when a FETCH statement moves beyond its last row. The rowset must be deselected explicitly, or it will consume resources (memory and disk space) for the remainder of the user's session.

If the cursor variable is a local variable, the rowset is deselected automatically when the procedure containing the variable completes.

DO

The DO statement performs repetitive execution with a post-test for loop termination. On each iteration, the DO statement executes the specified statement or statement block, then evaluates the specified expression. If the expression evaluates as TRUE, the cycle of statement execution followed by expression evaluation continues. If the expression eventually produces a FALSE result, flow passes to the statement following the DO statement.

Category Iteration statements.

Syntax `DO` *statement* `WHILE (`*boolean_expression*`) ;`

Parameters

statement	The statement to be executed for each repetition (it can be a compound statement).
boolean_expression	A scalar BOOLEAN expression evaluated after each repetition.

Example Select the sales office with the highest quota:

```
/* Find minimum offices needed for 100,000 quota */
select city, ytd_quota from offices
  order by ytd_quota desc;
total = 0;
count = 0;
do {
  fetch next;
  if ($sqlcode == $sqlnotfound)
    goto not_found;
  total = total + ytd_quota;
  count++;
} while (total < 100000);
print count, "offices needed for", total, "quota";
```

Notes The *statement* is often a compound statement.

The *statement* is always executed once, even if the boolean expression is FALSE the first time it is evaluated.
An expression that evaluates to NULL is FALSE.

DROP DATABASE

The DROP DATABASE statement is used to delete databases.

Category Database entity-deletion statements.

Syntax

```
DROP DATABASE "dbname" [ <IN>
LOCATION "dbloc" ];
```

Parameters

dbname A string literal specifying the name of the database to be deleted.

dbloc A string literal specifying a path name and indicating the location of the database *dbname*. If omitted, the database is assumed to be in the default location (the server's Public Databases folder, if no other has been specified with the USE LOCATION statement).

Example Delete the database "contacts":

```
DROP DATABASE "contacts";
```

Note

There is no way to retrieve deleted databases. Use this statement with caution.

Notes If the database is in use (i.e. another user has the database open) the DROP DATABASE statement will fail.

The DROP DATABASE statement, along with other entity creation statements, is not recorded in the database journal. The command is not affected by COMMIT or ROLLBACK statements.

DROP INDEX

The DROP INDEX statement is used to delete indexes from a database.

Category Database entity-deletion statements.

Syntax `DROP INDEX "indexName";`

Parameter

indexName	A string literal specifying the name of the index to be deleted in the current, open database. If there is no current database, or, if the index *indexName* does not exist in it, this statement generates an error.

Examples Delete the "names_by_last" index from the "contacts" database:

```
OPEN DATABASE "contacts";
DROP INDEX "names_by_last";
```

Note

There is no way to retrieve deleted indices. Use this statement with caution.

Notes If the index's table is open, the DROP INDEX statement will fail.

After dropping an index, the space occupied by the index will remain unused. Compact the database using ButlerTools to regain this space.

The DROP INDEX statement, along with other entity creation statements, is not recorded in the database journal. The command is not affected by COMMIT or ROLLBACK statements.

DROP TABLE

The DROP TABLE statement is used to delete tables from a database.

Category Database entity-deletion statements.

Syntax `DROP TABLE "tablename";`

Parameter

tablename	A string literal specifying the name of the table to be deleted in the current, open database. If there is no current database, or, if the table *tablename* does not exist in it, this statement generates an error.

Example Delete the "names" table from the "contacts" database:

```
OPEN DATABASE "contacts";
DROP TABLE "names";
```

Note

There is no way to retrieve deleted tables. Use this statement with caution. This statement also deletes any indexes associated with the table.

Notes If the table is in use the DROP TABLE statement will fail.

After dropping a table, the space used by the table will remain unused. Compact the database using ButlerTools to regain this space.

The DROP TABLE statement, along with other entity creation statements, is not recorded in the database journal. The command is not affected by COMMIT or ROLLBACK statements.

ERROR

The ERROR statement exists for those who process their own warnings and errors and want to generate a SQL error.

Those who want to use their own error processing in order to catch one or two errors, while passing other errors back, can use this statement to pass the errors back.

Category Output-control statements.

Syntax `ERROR` *major_code* `[,` *minor_code*`,` *message_text*`,` *item1*`,` *item2* `];`

Parameters

major_code	An integer expression containing the major error code value.
minor_code	An integer expression containing the minor error code value.
message_text	A string expression containing the primary message text.
item1	A string expression containing the secondary message text.
item2	A string expression containing the third message text.

Example Issue the following statements:

```
errorctl 1;
warnctl 0;
error $sqlcode;
```

These statements would cause termination of the SQL program with an error completion status (if errorctl is set to 1), as though the error associated with **$sqlcode** had occurred.

Note Unspecified, 'optional' parameters retain the settings of the last true error.

You should use the upper statement when ERRORCTL is set and when you encounter an error your SQL error handling routines do not handle.

ERRORCTL

The ERRORCTL statement controls the handling of data-management errors during execution of a SQL procedure. If the specified expression evaluates as zero, an error in a SQL data-management statement causes immediate termination of the SQL program with the error-completion status. The client application must handle the error and any required recovery.

If the expression evaluates as non-zero, then the system variable **$sqlcode** is set to reflect the error, and SQL program execution continues. In this case, the SQL procedure itself must handle the error and any required recovery.

Category Output-control statements.

Syntax `ERRORCTL` *expression* `;`

Parameter

expression An integer expression whose value governs error handling.

Examples Ask SQL to abort on data-retrieval errors:

```
errorctl 0;
select city, region from offices;
for each {
  . . . process the row . . .
}
```

Handle data-retrieval errors within the SQL program:

```
errorctl 1;
select city, region from offices;
if ($sqlcode != 0)
  goto handle_error;
for each {
  . . . process the row . . .
}
if ($sqlcode != 100)
  go to handle_error;
```

Note The ERRORCTL statement determines error processing until another ERRORCTL statement supersedes it.

EXECUTE FILE statement

The EXECUTE FILE statement loads and executes Butler SQL statements from a file. Procedure definitions contained in this file are compiled and loaded. Statements of other types are executed directly. All SQL statements are executed in the order in which they appear in the file. The procedure file can be used recursively.

Category Program execution statements.

Syntax `EXECUTE FILE` *filename_strlit* `[<IN> LOCATION` *location_strlit*`];`

Parameters

filename_strlit — A string expression containing the filename of the file that contains the SQL statements.

location_strlit — A string expression containing the location of the file that contains the SQL statements.

If no path is specified, the Procedures folder is assumed.

Notes When Butler SQL is launched, it executes the following statement implicitly:

```
execute file "msad$procedures";
```

Note that this feature should be used only in custom client applications and not in generic commercial applications. A generic SQL-based application should not require the user to specify a "procedure" file on the host system. Instead, the application should use the API to send to the host server any specific SQL statements and procedures needed.

Additionally, when a user logs on to Butler SQL, it executes the following implicity:

```
execute file "<user>$procedures"
```

where user is the user name used to log on to Butler SQL.

EXECUTE FROM

The EXECUTE FROM statement executes SQL statements contained in a string literal, variable, or column.

Category Program execution statements.

Syntax `EXECUTE FROM` *statement_strlit*`;`

Parameter

statement_strlit	A string expression containing a SQL statement.

Example

```
execute from "print 'hello world';";
```

prints out:

```
hello world
```

executing the following statements:

```
varchar v = "print ";
v = v + "city, "
v = v + "office_nr";
v = v + ";";
execute from v;
```

prints out:

```
New York    105
```

FETCH

The FETCH statement requests a single row from a rowset. SQL selects the requested row as the current row of the specified rowset. Cursor-based column references to columns of the rowset can subsequently be used in SQL expressions as identifiers that compare values of the specified column in the current row.

In its default form, the FETCH statement simply retrieves the next row of the current rowset. The optional parameters support other motions and permit concurrent processing of multiple rowsets.

Category Data-retrieval statements.

Syntax

```
FETCH [ motion ] [ OF cursor ] ;
```

where *motion* is:

```
{ FIRST
| LAST
| ABSOLUTE absrow
| NEXT
| PREVIOUS
| RELATIVE relrow }
```

Parameters

motion	Specifies which record to retrieve. If *motion* is omitted, FETCH NEXT is assumed.
cursor	The cursor returned by the SELECT statement that created the rowset. If *cursor* is omitted, the default cursor (**$cursor**) is assumed.
absrow	An integer scalar expression that specifies the absolute number of the row to be retrieved.

relrow — An integer scalar expression that specifies the row number to be retrieved relative to the current row (positive or negative).

Examples Simple sequential processing of query results:

```
select * from staff;
while ($true) {
  fetch;
  if ($sqlcode = $sqlnotfound)
     break;
 . . . other processing . . .
}
```

In this example, the other processing may be another statement testing the value of a column in the current row, or a PRINTROW statement that places the current row in the output stream.

Nested processing of staff within each office:

```
declare cursor offs, reps;
declare money office_sales = 0, office_quota = 0;
select office_nr, off_name from office into offs;
while ($true) {

/* Fetch the next office from the database */
    fetch next of offs;
    if ($sqlcode = $sqlnotfound)
  break;

/* Find all the sales reps from that office */
    select * from staff
     where staff.office_nr = offs—> office_nr
     into reps;
   while ($true) {

/* Fetch and process each staff member from the office
*/
    fetch next of reps;
      if ($sqlcode = $sqlnotfound)
      break;
     set office_sales = office_sales + reps—>
ytd_sales;
```

```
        set office_quota = office_quota + reps—> quota;
    }
    . . . update totals for this office . . .
}
```

Notes

The rowset has an unspecified order unless *sort_list* is specified in the SELECT statement which created the rowset.

When a rowset is initially generated, the current row is prior to the first row; therefore, a FETCH NEXT statement retrieves the first row.
There is no current row of the rowset:

- between the execution of SELECT and the first FETCH,
- when FETCH advances past the last row of the rowset, or
- if the current row is deleted.

Only the FETCH NEXT motion is supported if the update mode of the SELECT statement that created the rowset was READONLY or UPDATE.

Attempts to fetch past the end of a rowset with FETCH FIRST, LAST, NEXT, or PREVIOUS cause an **$sqlcode** value of **$sqlnotfound;** for FETCH ABSOLUTE or RELATIVE, an error code is returned.

FOR

The FOR statement performs repetitive execution with specified loop initialization, control, and reinitialization. This statement provides looping, which is convenient for numeric iteration.

Execution begins with the loop-initializer assignment (initial_asg). The loop-control expression is then evaluated (test_expr). If it produces a TRUE result, the statement that forms the body of the FOR statement is executed, then the loop-reinitialization assignment is executed (reinit_asg). The cycle of loop-control evaluation, statement execution, and loop reinitialization occurs until the loop-control expression produces a FALSE result. Next, flow passes to the statement following the FOR statement.

Category Iteration statements.

Syntax `FOR (`*initial_asg*`;` *test_expr*`;` *reinit_asg*`)` *statement*

where *initial_asg* has the form

varname = expr

Parameters

initial_asg	The loop-initializer assignment.
test_expr	A scalar BOOLEAN loop-control expression.
reinit_asg	The loop-reinitializer assignment.
statement	The statement to be executed on each iteration. This is often a compound statement (multiple statements grouped inside braces).

Example Process sales representatives by their sales performance, in quartiles:

```
for (i = 0; i < 1.0; i = i + 0.25) {
  select * from staff where (ytd_sales / ytd_quota
  between i and (i + 25));
  . . . process this quartile . . .
}
```

Note The *initial_asg* is always executed once. If the loop-control expression (*test_expr*) produces a FALSE result the first time it is evaluated, then *statement* and *reinit_asg* will never be evaluated.

To increment the loop variable by 1 each time, the statement block is executed, use the following syntax in the reinit_asg parameter:

```
varname++
```

To decrement the loop variable by 1:

```
varname--
```

FOR EACH

The FOR EACH statement performs an iteration over each row in a rowset. This statement provides a more convenient alternative to a WHILE loop containing a FETCH statement and **$sqlcode** testing.

The specified statement is executed once for each row of the rowset identified by the specified cursor. Cursor-based column references in the statement can be used for the iteration to obtain the values of the columns in the current row.

Category Iteration statements.

Syntax `FOR EACH [` *cursor* `]` *statement*

Parameters

cursor	A CURSOR variable identifying the target rowset. If *cursor* is omitted, the default cursor (**$cursor**) is implied.
statement	The statement to be repeated for each row of the rowset.

Example Accumulate the sales figures for each sales representative into the "offices" table:

```
declare cursors;
/* Form a row set from the staff table */
select rep_office, ytd_sales from staff into s;

/* For each staff member, update the correct office
table row */
for each s {
  update office set ytd_sales = ytd_sales + s –>
ytd_sales
    where office_nr = s–>rep_office;

}
```

Notes Failure during retrieval of any row in the rowset terminates the FOR EACH iteration with an error. A BREAK statement can also prematurely end the FOR EACH loop.

The CONTINUE statement can be used to jump from any point in the statement block to the beginning of the next loop iteration.

GOTO/LABEL

The GOTO/LABEL statements unconditionally change the flow of control in a SQL program. The GOTO statement causes an immediate branch to the statement labeled by the specified statement label. The LABEL statement associates a statement label with a particular statement, allowing it to serve as a branch destination.

These statements cause an immediate jump to a separate section of the program allowing for processing of any type (for example, print formatting). Overuse of the GOTO statement can cause a program to be extremely difficult to understand and debug, and it is not considered a good structured programming technique. Use the SWITCH statement instead whenever possible.

Category Conditional execution and branching statements.

Syntax

```
GOTO statement_label ;
LABEL statement_label : statement
```

Parameters

statement_label	A SQL identifier that becomes a label for the following *statement.*
statement	The statement(s) to be executed upon branching.

Example Detect a query execution error and branch to handle it:

```
select * from orders where ord_amount > 10000;
if ($sqlcode < 0)
  goto handle_error;
  . . . processing for a successful query . . .
return;
label handle_error:
. . . processing for a query error . . .
```

Notes The GOTO and LABEL statements may appear only within a procedure definition; they are not permitted in the outer block.

The scope of a statement label is the procedure that contains it.

A statement label and local variable may share the same name without ambiguity.

A GOTO may 'jump out' of a FOR, WHILE, DO, or SWITCH statement to break the flow of control. 'Jumping' into the middle of a FOR, WHILE, or DO loop, or a SWITCH statement is not recommended.

The GOTO statement is best suited for error handling in procedures. It allows you to jump from any point, even if you are nested deeply in looping or control statements.

IF

The IF statement conditionally executes SQL statements, depending on a test condition. When the statement is executed, the specified expression is evaluated. If it produces a TRUE value, then the conditioned statement is executed. Otherwise, the statement conditioned by the ELSE clause is executed if it is present.

Category Conditional execution and branching statements.

Syntax `IF (`*boolean_expression*`)` *true_statement* `[ELSE` *false_statement* `]`

Parameters

boolean_expression	A scalar BOOLEAN expression evaluated as the test condition.
true_statement	The SQL statement or compound statement executed if *boolean_expression* is evaluated as TRUE.
false_statement	The SQL statement executed if *boolean_expression* is evaluated as FALSE.

Examples Check to see if a preceding query returned more than 100 rows:

```
if ($rowcnt > 100)
  print total1;
else
  print total2;
```

Put large orders into a separate table on processing cycle number 7:

```
if (cycle = 7) {
  insert into bigorders
    (select * from orders where ord_amount > 20000);
  if ($sqlcode < 0)
    goto handle_failure;
  else {
```

```
    delete from orders where ord_amount > 20000;
    print "big orders deleted";
  }
}
```

Note An expression that evaluates to NULL is FALSE.

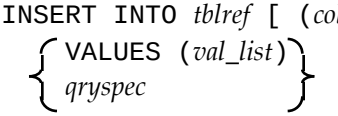
INSERT

The INSERT statement inserts new rows into a database table. SQL adds new rows to the database from one of two specified sources:

- A list of data values specified in the VALUES clause. In this case, the statement adds a single new row to the table.
- A query specified as part of the INSERT statement. In this case, the number of rows added corresponds to the number of rows in the query results.

Category Data-update statements.

Syntax



Parameters

tblref	The table to receive the new row(s).
colref_list	A comma-separated list of the columns to receive the new data, displayed in the same order as the columns from the data source. If *colref_list* is omitted, an ordered list of all columns in the table is assumed.
val_list	A comma-separated list of values for the columns specified in *colref_list*. Each item in *val_list* must be a constant or a SQL variable name. The number and data types of the items must agree with *colref_list*.
qryspec	If present, indicates that the specified query should be carried out and that its results should be added to *tblref*. The *qryspec* must be a query specification as

described in the section "Query Specification" of the Butler SQL Programmer's guide.

Examples This example inserts a row for New York City into the offices table:

```
insert into offices (office_nr, city) values (100,
"New York City");
```

This INSERT statement can be used to copy rows to the table:

```
insert into test_table (number, string) select
office_nr, city from offices;
```

Note that the number of SELECT values must match the number of INSERT columns. The following statement is not valid because the "offices" table has more than two columns:

```
insert into test_table(number, string) select * from
offices;
```

SQL supports automatic data type conversion, but the data must be of the same type class. The following statement is invalid because it tries to assign the VARCHAR field "city" to an INTEGER field "number."

```
insert into test_table (number,string) select city,
name from offices;
```

Notes Columns in the table reference that are not named in *colref_list* are either set to their default value or NULL in the added rows.

The database may restrict the values that can be assumed by a column. Attempts to insert a row in violation of these restrictions will result in an error.

Omission of *colref_list* is discouraged, since changes in the table definition may change the meaning of the INSERT verb; thus invalidating any INSERT statements in your application.

Generally, an INSERT statement with an embedded subquery will outperform separate SELECT and INSERT statements because Butler SQL supports both retrieval and insertion as a single operation.

OPEN DATABASE

The OPEN DATABASE statement opens a database for subsequent processing. The OPEN DATABASE statement must be executed before any other access to the database is permitted.

Category Database-access statements.

Syntax

```
OPEN [ dbbrand ] DATABASE [ "dbname" ]
 [ ALIAS dbalias ]
 [ <IN> LOCATION "dbloc "   ] [ <AS> USER "user"
 [ <WITH> PASSWORD "passwd" ] ]
[ FOR [ SHARED     ]  { READONLY }  ] ;
      [ PROTECTED  ]  { UPDATE   }
      [ EXCLUSIVE  ]
```

Parameters

dbbrand An identifier specifying a SQL DBMS brand. If *dbbrand* is omitted, Butler SQL (RMR) is assumed.

dbname A string literal specifying the database name. It is required by Butler SQL, but it may be omitted for DBMS brands not supporting named databases.

dbalias A SQL identifier that becomes the alias for the open database. If *dbalias* is omitted, dbname is used and must be a legal identifier.

dbloc A string literal specifying the location of the database. If this is omitted, the database is assumed to be in the Public Databases folder on the server Macintosh. If included, it is the full pathname to the database file server.

user — A string literal specifying the user name to be used for database access. If *user* is omitted, the user name specified in the OPEN DBMS statement is used.

passwd — A string literal specifying the password associated with the user name–if any password is associated.

`SHARED` — Specifies that database access may be shared with other users who concurrently update the database, and whose operations may cause interference with this user.

Note

In order to be able to specify SHARED, PROTECTED, or EXCLUSIVE, you must also specify a mode (i.e. READONLY or UPDATE).

`PROTECTED` — Specifies that database access may be shared with other users but that their operations may not cause interference with this user. This is the default sharing mode if neither SHARED nor EXCLUSIVE is specified.

`EXCLUSIVE` — Specifies that exclusive access to the database is reqired by this user.

`READONLY` — Specifies that the user may read, but not modify, the contents of the database being opened.

`UPDATE` — Specifies that the user may read and modify the contents of the database being opened. This is the default update mode if READONLY is not specified.

Examples

Open database “orders” on the default DBMS:

```
open database "orders";
```

Open a database with user name “rick” and password “casablanca” on the current host system:

```
open database as user "rick" with password
"casablanca";
```

Open the database named “mydb” in the “[many.accts]” directory:

```
open database “mydb” in location “many: acct”;
```

Notes

When connecting using ODBC, the database referred to in the data source is opened automatically, so use of the OPEN DATABASE command is not required.

The newly opened database becomes the default database for the session. Unqualified table references implicitly refer to this database.

The host system is the system where the SQL server opening the DBMS executes its statements. For DBMS brands with distributed database capabilities, databases on other hosts may also be available via the access point and the DBMS software on the named host. This parameter is applicable to DAL/DAM connections only. It is not applicable to ODBC connections.

Butler SQL supports the opening of multiple concurrent databases by executing two or more OPEN DATABASE statements.

The names of available databases for a given DBMS brand are available through the DESCRIBE DATABASES statement.

The sharing modes (SHARED/PROTECTED/EXCLUSIVE) offer more explicit control over the way in which the database may be shared with other users. For most ‘true’ database management systems, the SHARED and PROTECTED modes will be identical, and EXCLUSIVE access may not be possible at all.

If the ALIAS parameter is omitted, the database name is used as the database alias. This name must be a legal identifier within Butler SQL or must be enclosed in quotes.

For 'quasi-database' management systems that are actually data-dictionary/file-system combinations, the distinctions become more meaningful. In this case, SHARED access permits the broadest concurrent file access but at the risk of uncoordinated updates (since most such programs do not support transactions).

Access granted in the PROTECTED mode typically permits only one writer, but multiple concurrent readers of the database. Access permitted in the EXCLUSIVE mode locks out all other users of the database.

The update modes (READONLY/UPDATE) provide Butler SQL with information about operations to be performed against the database, which may be used to optimize access. For most 'true' database management systems, the update mode will not affect the actual process of opening the database for access.

The sharing and update modes are advisory statements passed on to Butler in order to allow the server to process queries efficiently. These modes are not enforced by Butler SQL.

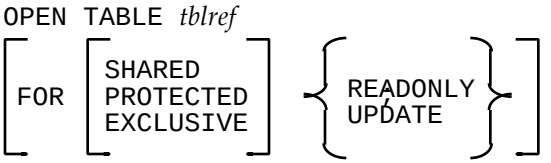
OPEN TABLE

The OPEN TABLE statement opens a table within a database for subsequent processing. The OPEN TABLE statement provides a way to inform Butler SQL that a particular table will be used for a period of time during the session in progress. It also provides a way to gain exclusive access to a table, or to specify non-default ways of sharing the table with other concurrent users.

Unlike the OPEN DATABASE statement for databases, an OPEN TABLE statement is not required before reference to a table in subsequent SQL statements can be made. The use of the OPEN TABLE/CLOSE TABLE statements is advised, however, if the client application can anticipate the pattern of table access during the SQL session. This optimizes access to these tables.

Category Database-access statements.

Syntax



Parameters

tblref A reference to a table in a database previously opened by an OPEN DATABASE statement. The reference may be qualified or unqualified. If it is unqualified, the statement refers to a table in the default database.

SHARED Specifies that table access may be shared with other users who concurrently update the database and whose operations may cause interference with this user.

PROTECTED	Specifies that table access may be shared with other users who concurrently update the database and whose operations may cause interference with this user.
EXCLUSIVE	Specifies that exclusive access to the table is required by this user.
READONLY	Specifies that the user may read, but not modify, the contents of the table being opened.
UPDATE	Specifies that the user may read and modify the contents of the table being opened.

Examples

Open a table in the default database in the default mode:

```
open table orders;
```

Open a table in the default database for exclusive update:

```
open table orders for exclusive update;
```

Open a table in the database named "rob" for shared reading:

```
open table rob!orders for shared readonly;
```

Notes

The default-sharing modes (SHARED/PROTECTED/EXCLUSIVE) and update modes (READONLY/UPDATE) for the table are the modes in which the database containing the table was opened. The OPEN TABLE statement can specify less stringent modes than the corresponding OPEN DATABASE statement, but a table cannot be opened with more stringent modes.

The OPEN TABLE statement is not mandatory for accessing a table in subsequent SELECT, INSERT, DELETE, or UPDATE data-manipulation statements. If an OPEN TABLE statement has not been issued, the SQL host server automatically opens the tables prior to executing the statements and closes them afterward.

Note

Butler SQL does *not* automatically close a table after executing the statements which opened it automatically. Once a table is open, it remains open until a CLOSE TABLE or CLOSE DATABASE statement is executed.

For most relational database management systems, the OPEN TABLE statement has no effect unless an EXCLUSIVE mode is specified, in which case it causes the SQL server to lock the specified table for exclusive access.

Note

The first Butler SQL user to open a particular table may experience a noticable delay. Subsequent 'opens' of the same table by other users will be instantaneous. In environments where all users access the same database tables, it would be a good idea to have each user open the tables upon connection. By doing this, you ensure that after the first user is connected, all others will experience no delay when accessing a table.

For more information about the meaning of the various sharing and update modes, see "OPEN DATABASE" on page 72.

PRINT

The PRINT statement outputs a series of data values as a single row of output.

Category Output-control statements.

Syntax `PRINT` *expression_1* `[ , ` *expression_2* `] . . . ;`

Parameter

expression — A scalar expression whose value is to become a column of the output row. The *n*th expression specifies the value for column number *n*.

Results The statement creates a single output row with a number of columns equal to the number of expressions in the statement. The data type of each column is the data type of the corresponding expression in the expression list, subject to any output data type mapping specified by the PRINTCTL statement.

Examples Print the values of three SQL variables:

```
print total1, total2, total3;
```

Print a total and a calculated quantity:

```
print tot_count, (tot_sales / tot_quota) * 100;
```

Print column contents from the current row of the cursor named "data":

```
print data –> item_nr, data –> item_price, data –>
item_qty;
```

Note A maximum of 255 expressions may appear in the expression list.

Note

All PRINT statements are not valid in ODBC, except when used in stored procedure.

PRINTALL

The PRINTALL statement prints the entire contents of a SQL rowset.

Category Output-control statements.

Syntax `PRINTALL [ cursor ];`

Parameter

cursor	The cursor identifying the rowset whose contents are to be printed. If cursor is omitted, the default cursor (**$cursor**) is assumed.

Results The statement creates zero or more output rows, one for each row of the rowset. The number of columns in each row is equal to the number of columns in the corresponding row of the rowset. The data type of each column is the type of the corresponding column of the rowset, subject to output data mapping.

Examples Perform a database query and output its results:

```
select office_nr, city, sales_ytd
  where sales_ytd < quota_ytd;
printall;
```

Print the contents of a rowset named "data":

```
printall data;
```

Notes The PRINTINFO statement can be used to obtain information about the names and data types of the columns of a rowset, as well as other related information.

Butler SQL does *not* impose any restriction on the amount of data printed; however, some DAL/DAM client applications may not be able to handle results that exceed 2000 characters per row or 255 characters per column. See "Appendix A • Butler–DAM Specifics" in the Butler SQL Programmer's Guide for information on writing client applications which can take advantage of Butler SQL's long data items.

For some operations, rowsets are formed for the duration of a single statement and then destroyed. For typical data-retrieval operations, rowsets persist and can be manipulated by a SQL program for row-by-row processing. The following FOR EACH statement will loop through each row of the rowset and print each item:

```
/* Using default cursor specification */
select * from offices;
for each {
   /* Now loop through all the columns */
   for (i=1; i < $ (); i++) {
     print –> i;
   }
}
```

You can return data items of any size by breaking each item into separate items and recombining them on the client. In the following example, the previous FOR EACH statement is modified to determine if the size of the data item is larger than 255 characters before it is returned:

```
/* Using default cursor specification */
select * from staff where dept = "sales";
for each
  { /* Now loop through all the columns */
  for (i = 1; i < cols(); i++)
    { /* column width */
    if $len(–> i) > 255
      for (sentchars = 1; sentchars < $len(–> i);
sentchars+200)
        { print $substr(–> i, sentchars, 200);
        }
    }
  }
```

Note

For strict conformance to ODBC standards, PRINTALL should not be used after a select.

PRINTCTL

The PRINTCTL statement controls data type mapping of SQL output rows. Using this statement, you can specify the manner in which each SQL data type is to be returned to the client application. This ensures that the client application can process all returned data.

The statement has three forms. In the first form, the statement specifies data type conversions by position. The second form allows a specific conversion to be specified: from a source data type to a destination data type. The third form of the statement lets you specify a SQL procedure that handles data-type conversion for a particular source data type.

Category Output-control statements.

Syntax

```
PRINTCTL typespec1, typespec2, ... typespec15 ;
PRINTCTL srctypespec = dsttypespec, ... ;
PRINTCTL srctypespec = procname ;
```

Parameter

typespec	An integer expression, or the name of a previously declared SQL procedure that governs conversion of SQL output data. The specification for *typspec1* governs output mapping of SQL data type codes 1 (BOOLEAN), *typspec2* governs output mapping of SQL data type code 2 (SMINT), and so on. Any *typespec* can be omitted from the list, causing the SQL data type to be unmapped on output.
srctypespec	The source data type to be converted.
dsttypespec	The destination data type to which the source data type will be converted.
procname	The name of the procedure whose return value replaces the source data type.

SQL data types can be referred to by either data type code or by their SQL system-constant name. SQL data types, codes, and names are shown in Table 7.

Table 7: SQL data type codes

Type	Code	System constant name
BOOLEAN	1	$boolean
SMINT	2	$smint
INTEGER	3	$integer
SMFLOAT	4	$smfloat
FLOAT	5	$float
DATE	6	$date
TIME	7	$time
TIMESTAMP	8	$timestamp
CHAR	9	$char
DECIMAL	10	$decimal
MONEY	11	$money
VARCHAR	12	$varchar
VARBIN	13	$varbin
LONGCHAR	14	$longchar
❖ The following are Butler-only data types.		
PICTURE	24	$picture
SOUND	25	$sound

Table 7: SQL data type codes

Type	Code	System constant name
ICON	26	$icon
MOVIE	27	$movie
DOCUMENT	28	$document

If an integer expression appears as *typespec,* the PRINTCTL statement directs Butler SQL to convert data of that type to the target data type (whose type code is the value of the integer expression), before sending the data back to the SQL client application. For example, if the integer expression for *typespec*2 produces the value 3, the SQL server is directed to convert SMINT (type code 2) data to INTEGER (type code 3) data on output.

The special code 0 (zero) can be used in the *typespec* list to request mapping to VARCHAR (type code 12) data, with NULL values returned as the text string "$NULL." As a convenience, this statement:

```
printctl 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0;
```

can be abbreviated as

```
printctl 0;
```

If the name of a SQL procedure appears as *typespec,* the PRINTCTL statement directs Butler SQL to instruct the named procedure to convert data of that type before sending it back to the client application. The specified procedure should be declared to take a single argument of the appropriate type and produce a single return value. Butler sends the return value from the procedure back to the client application.

Note

There is no way to find out what the current PRINTCTL settings are.

Result The statement governs output data mapping until another PRINTCTL statement supersedes it.

Examples The following example maps floating-point data types to the DECIMAL data type without changing the mapping of any other data types:

```
printctl  , , , 10, 10,0;
```

The same conversion can be specified using the second form of the statement:

```
printctl $smfloat = $decimal;
printctl $float = $decimal;
```

The following statement also specifies the same conversion:

```
printctl $smfloat = $decimal, $float = $decimal;
```

You can ask Butler SQL to perform very flexible data-type conversion by specifying a conversion procedure in the PRINTCTL statement. For example, the following SQL defines a procedure and PRINTCTL statement for mapping MONEY data to character strings with a leading dollar sign:

```
procedure domoney(inval) returns varchar;
argument money inval;
{
    return "$ " + varchar inval;
} end procedure domoney;

printctl $money = domoney;
```

As the above example shows, the conversion procedure must accept a single argument of the source data type and produce a single return value of the destination data type. The PRINT-

CTL statement causes the procedure to be called every time Butler SQL attempts to print data of the specified source data type.

The following example shows a procedure declaration and PRINTCTL statement that map TIMESTAMP data into the data type DATE:

```
procedure makedate(ts) returns date;
/* converts ts variable to date */
argument timestamp ts;
{
    return date ts;
} end procedure makedate;

printctl $timestamp = makedate;
```

Notes The PRINTCTL statement affects the data actually returned to the client application. The API functions will accurately report the mapped type and return mapped data.

The PRINTINFO statement reports the actual SQL data type of each column in a rowset –that is, the unmapped types. However, if the contents of a rowset are output with a PRINTALL or PRINTROW statement, the client application will map the received data.

Note

All PRINT statements are not strickly valid in ODBC, except as a stored procedure.

PRINTF

The PRINTF statement combines data items into an output-formatted string.

Category Output-control statements.

Syntax `PRINTF ( format_expression, [ expression_list ] ) ;`

Parameters

format_expression	A scalar string expression whose value is a format string that controls formatting.
expression_list	A comma-separated list of scalar expressions whose values are to be included in the output string, subject to the specified format.

Result The statement creates a single output row containing a single output column of type VARCHAR, subject to output data mapping.

Examples Output a text sentence containing variable data:

```
select * from staff for extract;
printf ("Row set has %d columns and %d rows", $colcnt,
$rowcnt);
```

Format a table containing query results for the client application:

```
select cust_name, rcvbl_balance from customers;
printf ("Customer | Balance");
for each {
  printf ("%20s | $%10.2p", cust_name, rcvbl_balance);
}
```

Notes The PRINTF statement is especially useful for text-oriented client applications and have little or no tabular-processing capability.

The **$format( )** function provides the same formatting features as PRINTF, but produces a VARCHAR return value rather than generating output.

Note

All PRINT statements are not strickly valid in ODBC, except as a stored procedure.

PRINTINFO

The PRINTINFO statement outputs a description of a SQL rowset, making it available to the client application.

Category Output-control statements.

Syntax `PRINTINFO` [*cursor*] ;

Parameter

cursor The name of the cursor variable identifying the rowset to be described. If *cursor* is omitted, the default cursor (**$cursor**) is assumed.

Results The PRINTINFO statement generates SQL output to the client application in the same manner other PRINT statements generate output. The statement generates a single row for each column of the specified rowset. The row contains the six columns of information shown in Table 8.

Table 8: PRINTINFO rowset

Col #	Data type	Information
1	SMINT	Column number
2	VARCHAR[255]	Column name
3	SMINT	Column data type (see Chapter 2 • Data Types in the Butler SQL Programmer's Guide for information on data types)

Table 8: PRINTINFO rowset

Col #	Data type	Information
4	SMINT	Column length
5	SMINT	Column scale
6	SMINT	Column display width

Examples Describe the default rowset:

```
printinfo;
```

Describe the rowset identified by cursor "myset":

```
printinfo myset;
```

Note All of the information available in the output messages generated by the PRINTINFO statement is also available through system functions (for example, **$colname( ), $coltype( ), $rows( )**, and **$cols( )**).

Note

All PRINT statements are not valid in ODBC, except as a stored procedure.

PRINTROW

The PRINTROW statement outputs the current row of a SQL cursor.

Category Output-control statements.

Syntax `PRINTROW [ cursor ] ;`

Parameter

cursor The name of the cursor identifying the rowset whose current row is to be printed. If *cursor* is omitted, the default cursor ($cursor) is assumed.

Results The statement creates a single output row with a number of columns equal to the number of columns in the current row of the rowset. The data type of each column is the type of the corresponding column of the rowset, subject to output data mapping.

Examples Print the current row of the rowset identified by the cursor named "data":

```
printrow data;
```

Print the current row of the default rowset:

```
printrow;
```

Note The PRINTINFO statement can be used to obtain information about the names and data types of the columns of a rowset, as well as other related information.

Note

All PRINT statements are not valid in ODBC, except as a stored procedure.

PROCEDURE

The PROCEDURE statement defines a SQL procedure. A procedure can be called by other SQL procedures using the CALL statement. A SQL procedure takes zero or more arguments, which are passed on by value. The procedure returns zero or more return values. In addition, local variables can be declared for use within the procedure.

Category Procedure call statements.

Syntax

```
PROCEDURE procname [ ( [ argname_list ] ) ]
    [RETURNS datatype_list ; ]
    [ argdecl_list ]
    { statement }
END PROCEDURE procname ;
```

where *argdecl_list* is a list of argument declarations, each having the form

```
ARGUMENT datatype argspec_list ;
```

and *argspec_list* is a comma-separated list of argument names with optional initialization, each having the form

argname [= *const_expr*]

If *argname* is initialized, it is not required when calling the procedure.

Parameters

procname	A SQL identifier (up to 31 characters) that names the procedure.

Example Given an employee number, this procedure returns the employee's quota and year-to-date sales:

```
declare procedure getinfo(repnr)
returns money, money;
argument integer repnr;
{
  select quota, ytd_sales from staff where
staff.rep_nr = repnr;
```

```
  fetch;
  if ($sqlcode != 0)
    return $NULL;
  else
    return quota, ytd_sales;
}
end procedure getinfo;
```

Notes

Generally, execution of a procedure ends when a RETURN statement is executed. If no RETURN statement is encountered before the end of *statement* is reached, an implied RETURN statement (with no return value) is executed. Note that this results in an error if the procedure head specifies a return data type.

Procedure names and outer block variables share the same name space. Appearance of a PROCEDURE statement automatically redeclares any previously declared procedure or outer block variable with the same name.

All data flowing in or out of a procedure is passed along by means of arguments and return values. There are no provisions for user-defined global variables or a pass-by-reference mechanism.

There are two ways to execute an included procedure. Note the following example:

```
declare cursor y;
procedure find (me)
returns cursor;
argument varchar me;
{
   declare cursor x;
   select office, req_date, ord_amount from orders
   where orders.terms = me into x;
   return x;
}
end procedure find;
```

You can execute this included procedure using one of the following methods:

1. Use the valued-procedure call form:

```
y = find ("COD");
```

2. Use the function call form:

```
find ("COD") returning y;
```

You can initialize SQL procedure arguments in the procedure declaration. When a procedure is defined in this manner, the argument parameter is optional with the initialized value used as the default. Also, you do not need to enclose the argument parameters in parentheses. Here is an example:

```
procedure test (it)
argument varchar it = "the default value";
{
     print it;
}
end procedure test;
test;

the default value

test("something original");

something original

test "look no parens";

look no parens
```

In this example, the argument in the procedure test is initialized with the value "the default value". When no argument is passed, as in the first call of the procedure, the default value is used. However, when an argument is passed as in the second and third calls, then the default value is ignored and the argument is used.

RESTORE DATABASE

The RESTORE DATABASE statement is used to recover a database using a backup copy of the database as well as a current journal file. There are two forms of the RESTORE DATABASE statement:

Category Database-access statements.

- the RESTORE ALL DATABASES command will restore all completed transactions for every database in the specified transaction journal;
- the RESTORE DATABASE statement will restore all completed transactions for a specific database, ignoring journal entries for other databases.

Syntax

```
RESTORE <ALL> DATABASES FROM jrnl
  [ <IN> LOCATION path ]
  [ AFTER  afterTimeStamp ]
  [ BEFORE beforeTimeStamp ];

RESTORE  DATABASE dbname FROM jrnl
  [ <IN> LOCATION path ]
  [ AFTER  afterTimeStamp ]
  [ BEFORE beforeTimeStamp ];
```

Parameters

dbname	A string literal which specifies the name of the database whose transactions are to be restored.
jrnl	A string literal which specifies the file name of the journal to use.
path	A string literal which specifies the path to *jrnl*. This must be in the form of a standard Macintosh path specification. E.g.: “Mac HD:Butler SQL:My Journals:”. Note that since you are specifying a folder, the path must end in a colon (“:”).

If the IN LOCATION clause is omitted, *jrnl* is expected to be in the default journals folder, i.e., the Database Journals folder in the Butler Preferences Folder.

afterTimeStamp — An expression evaluating to a TIMESTAMP value specifying the earliest transaction to restore. If the AFTER clause is omitted, the restore begins with the first entry in the journal.

beforeTimeStamp — An expression evaluating to a TIMESTAMP value specifying the latest transaction to restore. If the BEFORE clause is omitted, the restore ends with the last entry in the journal.

Notes

To use the RESTORE statement effectively, you must make frequent backups of your databases, detach the journal regularly, and keep track of the time periods covered by a journal. There is currenly no way to view a Butler SQL journal file in text form.

It is very important to make backups of both the databases being restored and the journal before using this statement.

The databases being restored must be in the Public Databases folder in the Butler Preferences Folder and must have the same names they did when the transactions in the journal occurred.

The journal *jrnl* must be detached.

The RESTORE DATABASE command will suspend all other clients' processes until it is complete.

Examples

```
RESTORE DATABASE FROM 'Butler Journal_01/14/1994_1';
```

Restores all completed transactions found in the journal file "Butler Journal_01/14/1994_1" to all databases found in the Public Databases folder.

```
RESTORE DATABASE 'SQL_Demo' FROM 'Butler Journal_01/
14/1994_1'
```

```
IN LOCATION 'Hobbes:Saved Journals:'
AFTER  '07/28/1994 15:00:00'
BEFORE '07/28/1994 17:00:00';
```

Restores all completed transactions made between 3 and 5 PM on 28 July 1994 found in the journal file “Butler Journal_01/14/1994_1”, which is in the “Saved Journals” folder on the “Hobbes” volume, and which applies to the database “dal_demo” located in the Public Databases folder.

RETURN

The RETURN statement is used to return from a SQL procedure. The RETURN statement ends execution of a called SQL procedure; execution continues with the statement immediately following the CALL statement in the calling procedure.

One or more return values can optionally be returned to the calling procedure. Evaluating the expression list in the RETURN statement obtains the return values.

Category Procedure call statements.

Syntax `RETURN [` *expr_list* `] ;`

Parameter

expr_list — A comma-separated list of scalar expressions that are evaluated to obtain the procedure's return values. The resulting values must agree in number and data type with the return-value data-type list in the PROCEDURE statement header.

Example This procedure returns the quota and current quota percentage for a sales representative, given his or her employee number:

```
declare procedure getdata(repnr)
returns money, scientific;
argument integer repnr;
{
  select ytd_sales, ytd_quota from staff
    where staff.rep_nr = repnr;
  fetch;
  if ($sqlcode != 0)
    return $null,$null;
  else
    return ytd_quota, (ytd_sales/ytd_quota);
}
end procedure getdata;
```

Note Items in *expr_list* may evaluate to NULL, producing a NULL return value.

ROLLBACK

The ROLLBACK statement aborts the current transaction. Modifications to the database made during the transaction are undone, and a new transaction is begun.

Category Transaction management statements.

Syntax `ROLLBACK [ WORK ] [ <FOR>` *dbbrand* `] ;`

Parameter

dbbrand — The DBMS brand whose current transaction is to be committed. If *dbbrand* is omitted, the default brand is assumed.

Example Update the database to reflect the transfer of "Jones" to office number 301, and rollback the change to the database if there is an error condition:

```
update staff set office = 301 where rep_name =
"Jones";
. . . other processing . . .
if {error condition} rollback;
```

Note Butler SQL does not provide transaction integrity across multiple DBMS brands or host systems. The COMMIT/ROLLBACK mechanism can be used only to guarantee the integrity of modifications made to a single database during a transaction.

The dbbrand parameter is not applicable when using ODBC to communicate with Butler SQL. The parameter is optional when using DAL/DAM.

SELECT

The SELECT statement performs a database query, creating an rowset of data and an associated cursor. The cursor can be used for subsequent processing of the rowset by other data-manipulation statements. The various clauses of the SELECT statement specify how the rowset is to be formed and optionally specify a CURSOR variable to receive the cursor. Individual rows from the rowset can subsequently be retrieved for row-at-a-time processing with the FETCH statement, or the entire rowset can be reiterated with the FOR EACH statement. For example, the application can retrieve the next row and write it to the output stream with a PRINTROW statement.

For descriptions of rowsets and cursors, see "Rowsets" and "Cursors" in the Programmer's guide.

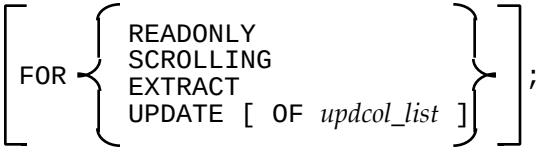
Category Data-retrieval statements.

Syntax

```
SELECT [ ALL | DISTINCT ] select_list
FROM from_list
[ WHERE where_condition ]
[  GROUP BY group_list ]
[ HAVING have_condition ]
```

queryspec

```
[ ORDER BY sort_list ]
[ INTO cursor ]
```



Parameters The *qryspec* parameters are described in detail in the section "Query specification" in the Programmer's guide. Briefly, the queryspec includes the following parameters:

ALL | DISTINCT	Specifies whether duplicate rows are to be eliminated from the result set.
select_list	A comma-separated list of *select_items*, which specifies the columns to appear in the rowset.
from_list	A comma-separated list of table specifications, identifying the tables and views from which the data in the rowset is drawn. At least one item must appear in the list.
where_condition	A search condition that determines which rows are included in the rowset. If *where_condition* is omitted, all candidate rows are included in the rowset.
group_list	A comma-separated list of the columns that are used to group the rowset and form summary rows. If *group_list* is omitted, no grouping is performed.
have_condition	A search condition that determines which summary rows (generated by the GROUP BY clause) are included in the rowset. If *have_condition* is omitted, all summary rows are included.

Duplicate-row elimination

Depending upon the columns specified in *select_list* and the data in those columns, the rowset generated by the query specification may include rows that are exact duplicates of one another. If the DISTINCT keyword is specified, these duplicate rows are eliminated from the rowset. If the ALL keyword is specified or neither keyword is specified, the duplicate rows are not eliminated.

Select list

The *select_list* is a comma-separated list of *select_items* that specify the columns to be included in the rowset. Each *select_item* can be one of the following:

- a column from a source table
- a calculated column specified by an expression
- an all-columns-of-table specification
- an all-columns specification

The columns of the resulting rowset correspond, in left-to-right sequence, to *select_items* in *select_list*.

A *select_item* of the following form specifies a column of a source table:

colref [<AS> *colalias*]

Here *colref* specifies a column of a source table, and *colalias* is a valid Butler identifier. The value of the column in each row is the value of the column in the corresponding row of the source table. The specified alias becomes the name of the column in the resulting rowset. If *colalias* is not specified, *colref* is used.

A *select_item* of the following form specifies a calculated column:

expr [<AS> *colalias*]

Here *expr* is a scalar expression and *colalias* is a valid Butler identifier. The *expr* may include references to columns of source table. The specified alias becomes the name of the column in the resulting rowset. If *colalias* is not specified, Butler assigns the name *expr* concatenated with

the column number. For example, a calculated column in position 6 receives the name *expr6*.

A *select_item* of the following form is shorthand for all columns of a table:

tblalias.* Here *tblalias* identifies one of the source tables. Butler expands the asterisk into a sequential list of all columns in the table.

A *select_item* consisting of an asterisk (*) is shorthand for all columns of all source tables. Butler expands the asterisk into a list of the columns, organized in left-to-right sequence within each table, as the tables appear in the FROM list.

The following example shows a select list that includes the first three types of *select_item:*

```
select offices.*, rep_name, (ytd_sales/
ytd_quota)
from offices, staff
where staff.office_nr = offices.office_nr ;
```

If all columns of both tables are required, the example could be specified as

```
select *, (ytd_sales/ytd_quota)
from offices, staff
where staff.office_nr = offices.office_nr ;
```

In general, you should not hard-code all-column (*) references of either type into Butler client applications because the meaning of a select expression may change if the structure of the referenced tables changes.

FROM clause

The FROM clause in a query specification identifies the tables and views from which the rowset is constructed. Each table specification in the *from_list* has the form

tblref [<AS> *tblalias*]

where *tblref* specifies a table or view in a currently open database and *tblalias* is a Butler identifier that becomes the alias of the table reference within this statement. If *tblalias* is omitted, the table reference is used as the alias. The aliases (whether explicit or implicitly specified) in *from_list* must be distinct from one another.

The tables and views identified in the FROM clause are referred to as source tables in the following sections. Each row in the rowset draws its data from a single row in each of the source tables.

WHERE clause

The WHERE clause in the query specification restricts the rows that are used to form the rowset. For each candidate row of the rowset, *where_condition* is evaluated to produce a BOOLEAN result. If the result is TRUE, the row continues as a candidate row of the rowset; otherwise, it is discarded and is not included in the rowset. If the WHERE clause is omitted, all candidate rows continue as candidate rows.

The search condition may select a single row from the database, for example:

```
select * from offices where office_nr =
103 ;
```

Or the search condition may select multiple rows, for example:

```
select * from offices where
ytd_sales > quota ;
```

Valid search conditions are described in the section "Search Conditions" in the Programmer's guide.

GROUP BY clause

The GROUP BY clause controls consolidation of the candidate rows of the rowset into summary rows. If the clause appears, the rows in the rowset are divided into groups of rows that have identical data values in one or more specified columns. Each group is then summarized in a single, summary row for the group. The *group_list* is a comma-separated list of column references, each identifying a grouping column. If the GROUP BY clause is omitted, no grouping is performed.

When the GROUP BY clause is specified, each item in *select_list* must be constant-valued for a group. That is, *select-item* must be either a grouping column (which by definition has a constant value for all rows in the group) or an expression containing a statistical function (which summarizes the values from all rows in the group as a single value). Statistical functions are specified by aggregate functions, described later in this chapter.

The following example specifies a rowset of summary rows from the "staff" table, where each row summarizes data for a single office:

```
select office_nr, sum(ytd_sales),
sum(ytd_quota)
from staff
group by office_nr ;
```

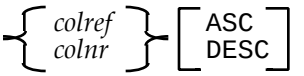
In this example, SUM( ) is an aggregate function (see the section "Aggregate Functions" later in this chapter for more information).

When the GROUP BY clause is not specified, then either (1) no aggregate functions can appear in *select_list* or (2) every column reference in *select_list* must be an argument to an aggregate function. In the latter case, the query specification produces a single row, summarizing the entire rowset.

When NULL values occur in the grouping column, the results of the query may differ from one DBMS brand to another. Some DBMS brands, such as ORACLE, consider two NULL values to be identical when used with the GROUP BY clause and collect all rows with a NULL grouping column value into a single row group. Other DBMS brands, such as INFORMIX, treat two NULL values as different values when used with the GROUP BY clause and generate a separate row group for each row with a NULL grouping column value.

Butler treats two NULL values as being identical when used with the GROUP BY clause, and therefore generates one summary row in this case.

HAVING clause	The HAVING clause is used to restrict the summary rows created by the GROUP BY clause, just as the WHERE clause restricts the candidate rows that are subject to grouping. For each candidate summary row of the rowset, *have_condition* in the HAVING clause is evaluated to produce a BOOLEAN result. If the result is TRUE, the summary row continues as a candidate row of the rowset; otherwise it is discarded and is not included in the rowset. The search conditions that can be specified are described in the following section, "Search conditions." Any column reference within the HAVING clause search condition must reference a column that has a constant value for all rows in a group, or it must be an argument to an aggregate function that summarizes column values in the group. If the HAVING clause is omitted, all candidate summary rows continue as candidate rows. If the HAVING clause is specified without a GROUP BY clause, the entire rowset is treated as a single group for purposes of the HAVING clause.
colref	Specifies sorting based on the value of the named column, which must be a column in the *select_list* of *qryspec*.
colnr	Specifies sorting based on a column in the *select_list*, identified by its ordinal position in the list.
ASC	Specifies an ascending collating sequence for the column, which is the default.

DESC	Specifies a descending collating sequence for the column.
cursor	A CURSOR variable that receives the cursor created by the SELECT statement. If *cursor* is omitted, only the system variable **$cursor** is set.
READONLY	Specifies that the rowset resulting from the query will be read sequentially, using FETCH NEXT statements only, and that the cursor will not be used to update the database. READONLY is the default if no other update mode is specified.
SCROLLING	Specifies that the rowset resulting from the query will be processed by means of FETCH motions (NEXT, PREVIOUS, FIRST, LAST, and so forth) and that the cursor will not be used to update the database.
sort_list	where *sort_list* is a comma-separated list of sort specifications, each having the form 
EXTRACT	Specifies that the rowset resulting from the query will be processed by means of any of the possible FETCH motions and that the cursor will not be used to update the database. With this mode the number of rows in the rowset can be determined after the SELECT statement is complete.
UPDATE	Specifies that the rowset resulting from the query will be read sequentially, using FETCH NEXT statements only,

and that the cursor may be used to update the database in subsequent positioned UPDATE or positioned DELETE statements. If UPDATE is specified, the FROM clause of the query specification must name a single table.

updcol_list — Specifies that only specific columns may be updated by subsequent positioned UPDATE statements. The parameter is a comma-separated list of column references, each naming a column of the table in the FROM clause of the query specification.

Results — The system variable **$cursor** receives the cursor created by the SELECT statement. Unqualified cursor-based column references in SQL data-manipulation statements reference this cursor implicitly.

If *cursor* is specified, the named SQL CURSOR variable also receives the cursor created by the SELECT statement.

The system variable **$colcnt** is set to the number of columns in the rowset.

The system variable **$rowcnt** is set to the number of rows in the resulting rowset, if the EXTRACT parameter is specified. Otherwise the **$rowcnt** variable is set to NULL.

Update modes. The SELECT statement update modes (READONLY, UPDATE, SCROLLING, and EXTRACT) control the detailed operation of the SELECT statement and the subsequent FETCH and FOR EACH statements that reference the rowset it creates.

The simplest update mode is READONLY. In READONLY mode, Butler obtains data from the database row by row, as directed by subsequent FETCH or FOR EACH statements. Only forward sequential FETCH motion is supported; previously retrieved rows are discarded by Butler and are no longer available without performing another query.

Because the READONLY mode retrieves rows from the database only on request, the number of rows in the rowset cannot be predicted until all rows have been obtained with the FETCH statement. The **$rowcnt** system variable is set to NULL immediately following the SELECT statement. This mode is therefore appropriate for most queries in which the data is read into the client application once and then processed exclusively within the client application.

The SCROLLING update mode is only slightly more complex than READONLY. In SCROLLING as in READONLY mode, Butler obtains data from the database as directed by subsequent FETCH or FOR EACH statements. But in addition to making the row available to Butler for processing, Butler adds the row to a shadow copy of the rowset, which it maintains. Butler supports FETCH motions for previously retrieved (or skipped) rows by reading them from the shadow copy.

The size of a rowset created in the SCROLLING mode, as in READONLY, cannot be predicted until all rows have been retrieved. The **$rowcnt** system variable is set to NULL immediately following the SELECT statement. This mode is appropriate for queries in which the client application anticipates a large rowset and uses the SQL server to support large-scale scrolling through the data. The cost of this flexibility is additional mass storage requirements and I/O overhead within the SQL server.

The EXTRACT update mode is a variation on the SCROLLING mode. Instead of waiting for subsequent FETCH requests, the SQL server retrieves all rows of the rowset from the database as part of the SELECT statement, filling its shadow copy of the rowset. The SQL server releases all locks against the actual rows of the database upon completion of the SELECT statement. All subsequent FETCH and FOR EACH statements as well as all FETCH motions are supported from the shadow copy of the rowset. The EXTRACT mode thus effectively takes a snapshot of the rowset, and subsequent operations take place against this snapshot.

With the EXTRACT mode, the number of rows in the rowset is known when the SELECT statement is complete, and the **$rowcnt** system variable is set accordingly. This mode is appropriate for queries in which the client must know the number of rows in advance or when prolonged processing against a snapshot of the database is anticipated. The cost of this flexibility is additional mass storage and I/O overhead for the SQL server as well as a potentially long execution time for the original SELECT statement.

The UPDATE mode supports cursor-based (positioned) modifications to the database. Rowsets created in this mode can be referenced in positioned DELETE (DELETE ... WHERE CURRENT OF) and positioned UPDATE (UPDATE ... WHERE CURRENT OF) statements. This mode is most similar to the READONLY mode because the SQL server retrieves rows of data from the database only on demand and discards previously retrieved rows. In UPDATE as in READONLY modes, the size of the rowset cannot be determined before all rows have been retrieved and the **$rowcnt** variable is NULL following the SELECT statement.

The UPDATE mode is appropriate when the SQL client application provides positioned update of data, such as allowing a user to update data form by form. Although SQL supports such applications, care must be taken in designing them, since, at a minimum, the current row of the rowset will be locked from access by other users, causing potential performance and lockout problems.

Examples

Select all rows and all columns from the "orders" table:

```
select * from orders;
```

Select only orders for more than $20,000:

```
select * from orders where ord_amount > 20000;
```

Select only the order amount and order date for those orders:

```
select ord_amount, ord_date from orders
  where ord_amount > 20000;
```

Select the order amount and customer name for each order:

```
select ord_amount, cust_name from orders, customer
  where orders.cust_nr = customer.cust_num;
```

Select offices that are over quota and sort them by year-to-date sales:

```
select * from offices where ytd_sales > quota
  order by ytd_sales desc;
```

Select offices that are under quota, for possible revision of their quotas:

```
select * from offices where ytd_sales < quota
into q1
for update of quota;
```

Sort the offices by quota performance and use the CURSOR variable "q1" to identify the resulting rowset:

```
declare cursor q1;
select city, (ytd_sales / quota), ytd_sales, quota
  from offices
  order by 2 desc
  into q1;
```

Notes

The **$maxrows** system variable can be set to restrict the number of rows retrieved by the SELECT statement.

As a convenience, SQL automatically declares the variable named by *cursor* if it was not previously declared.

If FOR UPDATE is specified, the ORDER BY, GROUP BY, and HAVING clauses cannot be present.

SET

The SET statement assigns a value to a SQL variable. The statement has three forms. In the first form, the specified expression is evaluated, and its value is assigned to the specified SQL variable. The second form (++) and third form (– –) increase and decrease, respectively, the specified variable.

Category Variable-support statements.

Syntax

```
[ SET ] variable_name = expression ;
[ SET ] variable_name + + ;
[ SET ] variable_name – – ;
```

Parameters

variable_name	A variable previously declared in the current scope. The variable must have a numeric data type for the increment and decrement forms of the statement.
expression	A scalar expression.

Examples Store the current region name in the variable named "reg":

```
set reg = "Eastern";
```

Calculate a percentage; note that the verb SET is optional and is not used here:

```
pct = (ytd_sales/ytd_quota) * 100;
```

Notes The keyword SET is optional.

The expression may include literals, variables, cursor-based column references, and arithmetic and comparison operators. It may not include subqueries or aggregate functions.

Butler SQL automatically converts the data type of *expression* to that of *variable_name,* if possible, according to the data-type conversion rules explained in the section "Data-type conversion rules" of the Butler SQL Programmer's Guide.

A NULL expression value assigns the NULL value to the variable.

The increment (++) and decrement (--) syntax is applicable to numeric-based variables only.

SWITCH

The SWITCH statement provides a multipath branch in a SQL program's flow of control, based on the value of a specified expression. The SWITCH statement offers a multiple-choice version of the IF statement that is based on numeric or string values rather than a simple TRUE/FALSE test.

The head of the SWITCH statement specifies an expression that governs conditional execution of the statement body. The SWITCH statement body contains one or more case prefixes, each specifying a constant integer or string expression and serving as a label for the statement that follows it. Other SQL statements follow each case prefix.

When the SWITCH statement is executed, the expression is evaluated. Then the resulting value is used to select the case prefix whose *comparison_expression* has the same value, and execution continues with the first statement following the case prefix. If no such case prefix is present, execution continues at the DEFAULT case prefix, if one is present. Otherwise, none of the statements in the body of the SWITCH statement are executed.

Category Conditional execution and branching statements.

Syntax

```
SWITCH (expression) {
  [ CASE comparison_expression :  statement_list ] . . .
  [ DEFAULT: statement_list ]
}
```

Parameters

expression	An expression that is evaluated to control the switch.
comparison_expression	A constant scalar expression that specifies the matching value for this case prefix.
statement_list	The sequence of SQL statements to be executed if the value of *expression* matches the case prefix.

Example Calculate sales by region:

```
declare cursor c;
select region_nr, ytd_sales from staff into c;
for each c {
  switch(region_nr/100) {
  case 1:
    /* Eastern Region */
    set east_sum = east_sum + ytd_sales;
    break;

  case 2:
    /* Central Region */
    set cent_sum = cent_sum + ytd_sales;
    break;

  case 3:
    /* Western Region */
    set west_sum = west_sum + ytd_sales;
    break;

  default:
    /* Should never happen */
    print "Bad region", region_nr/100;
    return;
  }
}
```

Notes The DEFAULT case prefix may appear only once in the statement, but it can be placed anywhere relative to case prefixes.

The case prefixes do not alter the flow of execution within the SWITCH statement body. After initial selection of a case prefix, flow continues sequentially until all statements in the body are executed, unless interrupted by a BREAK, RETURN, GOTO, or similar statement, as shown in the example. The most common use is for each case to end with a BREAK statement.

SQL converts data types for comparison just as it would for the comparison expression (*expression* == *comparison_expression*).

The system variable **$switch** is set to the value of *expression* for the last SWITCH statement executed, when the body of the SWITCH statement is entered.

UNDECLARE

The UNDECLARE statement removes the current declaration of one or more SQL variables. After the UNDECLARE statement is executed, subsequent references to the variable are illegal unless a new declaration is encountered. An UNDECLARE statement appearing within a procedure definition effectively removes a local variable declaration. An UNDECLARE statement appearing in the outer block removes an outer-block variable declaration.

Category Variable-support statements.

Syntax `UNDECLARE` *variable_list* `;`

Parameter

variable_list	A comma-separated list of previously declared variables whose declarations are to be revoked.

Example Declare variables to hold a loop counter and computer total, perform the computation, print the total, and then remove the declaration:

```
declare money total = 0;
declare integer qtr;
for (qtr = 1; qtr <= 4; qtr++) {
total = total + getresults(qtr);
}
print total;
undeclare total, qtr;
```

Notes Outer block variables and procedure names share the same name space, so the UNDECLARE statement can be used to remove procedure declarations as well as outer-block variable declarations.

The UNDECLARE statement is most useful in the outer block, where it can be used to prevent variable declarations from lingering and colliding with column names and other identifiers. Declarations of local variables within a procedure effectively end at the end of the procedure definition, so they seldom need to be undeclared explicitly.

UPDATE (positioned)

The positioned UPDATE statement modifies data in the current row of a cursor. The specified columns are updated according to expressions specified in the SET clause.

Category Data-update statements.

Syntax `UPDATE` *tableref* `SET` *assign_list* `WHERE CURRENT OF` *cursor* `;`

where *assign_list* is a comma-separated list of assignments, each having the form

colref = *expr*

Parameters

tableref	The table containing the rows to be modified.
colref	The name of the column in *tableref* to receive a new value.
expr	A scalar expression whose value is assigned to *colref*. The expression may include references to columns of *tableref*.
cursor	An active cursor with a valid current row that is modified by the statement.

Example Raise the quota for the salesperson indicated by the current row of cursor "x":

```
update staff set quota = quota * 1.2 where current of
x;
```

Notes The table reference must specify an table which can be updated. The cursor must specify a rowset created by a SELECT statement with a FOR UPDATE clause. If the FOR UPDATE clause specifies a list of updatable columns, then every *colref* in *assign_list* must be present in the FOR UPDATE list.

If an expression in *assign_list* includes a reference to a column being updated, the value used in computing the expression is the value of the column before any updates are performed on the row.

UPDATE (searched)

The searched UPDATE statement modifies data in one or more rows of a table based on row-selection criteria. The where condition specified in the statement identifies the rows to be updated. The specified columns of those rows are updated according to the instructions specified in the assignment list.

Category Data-update statements.

Syntax `UPDATE` *tableref* `SET` *assign_list* `[ WHERE` *where_condition* `] ;`

where *assign_list* is a comma-separated list of assignments, each having the form

colref = *expr*

Parameters

tableref	The table containing the rows to be modified.
colref	The name of the column in *tableref* to receive a new value.
expr	A scalar expression whose value is assigned to *colref*. The expression may include references to columns of *tableref*.
where_condition	The search condition that identifies the rows to be modified. If *where_condition* is omitted, all rows of *tableref* are modified.

Result The data in the target table is updated.

Examples Transfer all staff from office 102 to office 103:

```
update staff set office_nr = 103 where office_nr =
102;
```

Reset quota and year-to-date sales to zero for all staff:

```
update staff set quota = 0.00, ytd_sales = 0.00;
```

Notes If an expression in *assign_list* includes a reference to a column being updated, the value used in computing the expression is the value of the column before any updates are performed on the row.

USE DATABASE

The USE DATABASE statement establishes a previously opened database as the current default database for the SQL session. Subsequent unqualified table references implicitly refer to this database.

Category Database-access statements.

Syntax `USE DATABASE` *dbalias* `;`

Parameter

dbalias	The alias of the database that is to become the new default database. The OPEN DATABASE statement must have previously opened the database.

Example Open two databases in sequence and then re-establish the first one as the default database:

```
open  database "joe" as db1;
select * from offices;
printall; /* data comes from database "joe" */
open  database "sam" as db2;
select * from orders;
printall; /* data comes from database "sam" */
```

```
use database db1;
select * from customers;
printall; /* data comes from database "joe" */
```

Note Data in all currently open databases is accessible if a *dbalias* qualifier is used on table references in data-manipulation statements referencing the databases.

You are only required to use this command if you have more than one database open and if you use unqualified table references in other statements.

USE LOCATION statement

The USE LOCATION statement sets the default location (folder) Butler uses for database files. Unless changed, Butler always uses the Public Databases folder inside the Butler Preferences Folder.

Category Database-access statements.

Syntax

```
USE LOCATION "loc" ;
```

Parameter

loc — A string expression specifying the pathname to the folder containing the Butler database files on the server Macintosh.

Examples Set the default location on a Butler server:

```
use location "HD100:Personal Databases";
```

Notes When you open a database, it is preferable to use the IN LOCATION clause of the OPEN DATABASE statement rather than USE LOCATION.

If you do use the USE LOCATION statement, follow these rules:

1. Use it only once per session.
2. Use it before a OPEN DATABASE statement is executed.

WARNCTL

The WARNCTL statement functions just like ERRORCTL, except that it controls Butler's operation concerning warnings instead of errors. When WARNCTL evaluates to 0, the default setting, warnings cause immediate termination of the SQL program in the same way that errors do. With WARNCTL set to a nonzero value, the system variable **$sqlcode** is set to reflect the warning, and execution of the SQL program continues. In this case, the SQL program itself must manage the warning and any required recovery.

Category Output-control statements.

Syntax `WARNCTL` *expression;*

Parameter

expression	An integer expression whose value governs error processing.

Examples Ask Butler to stop running when a data-retrieval warning is found:

```
warnctl 0;
select city, region from offices;
for each {
  . . . process the row . . .
}
```

Respond to data-retrieval warnings within the SQL program:

```
warnctl 1;
select city, region from offices;
if ($sqlcode != 0)
  goto handle_warning;
for each {
  . . . process the row . . .
}
if ($sqlcode != 100)
  goto handle_warning;
```

Note The WARNCTL statement governs warning processing until another warning statement supersedes it.

WHILE

The WHILE statement performs repetitive execution with a pretest for loop termination. Before each iteration, the WHILE statement evaluates the specified test expression. If the expression is evaluated as TRUE, then the specified statement is executed and a cycle of expression evaluation followed by statement execution commences. When the expression eventually produces a FALSE result, flow passes to the next statement after the WHILE statement.

Category Iteration statements.

Syntax `WHILE (`*expression*`)` *statement*

Parameters

expression	A scalar expression whose value can be converted to BOOLEAN type (TRUE or FALSE), which is evaluated before each repetition loop.
statement	The statement or statement block to execute for each repetition.

Example Loop process each row from the "offices" table:

```
declare cursor x;
declare boolean not_done = $true;
select * from offices giving x;
while (not_done) {
  fetch x;
  if (sqlcode < 0) {
    not_done = $false;
    continue;
  }
. . . process this office . . .
}
```

Notes The *statement* is usually compound.

If an expression is evaluated as FALSE the first time, then *statement* will not be executed at all.

An expression that evaluates to NULL is FALSE.

The WHILE statement is similar to the DO statement, except that the DO statement is always executed at least once. In a WHILE loop, if the expression evaluates to false, the statement block in the WHILE loop are never executed.

Appendix A • SQL statement summary

This chapter summarizes alphabetically the syntax of every (SQL) statement supported by Butler SQL.

ALTER TABLE

```
ALTER TABLE base-table base-table-alteration

                base-table        a string or alphanumeric literal

                base-table-alteration

                                  column-alteration-action

                column-alteration-action

                                  add-column-action | alter-column-
                                  action drop-column-action

                add-column-action

                                  ADD [ COLUMN ] column-definition

                alter-column-action{ALTER | MODIFY}  [COLUMN] column

                                  { data-type | [SET] default-defini-
                                  tion | [SET] name "newcolumnname" |
                                  DROP DEFAULT }

                drop-column-action

                                  DROP [COLUMN] column {RESTRICT |CAS-
                                  CADE}

                column-definition

                                  column { data-type | domain }

                                  [ default-definition ]

                                  [ column-constraint-definition-list]

                default-defintion DEFAULT { literal | niladic-func-
                                  tion| NULL}

                niladic-function  USER

                                  CURRENT_USER
```

```
                                    SESSION_USER
                                    SYSTEM_USER
                                    CURRENT_DATE
                                    CURRENT_TIME
                                    CURRENT_TIMESTAMP
```

BACKUP DATABASE

```
BACKUP DATABASE <database name>
```

BREAK

```
BREAK;
```

CALL

```
[ CALL ] procname (argument_list) RETURNING variable_list ;
```

CLOSE DATABASE

```
CLOSE DATABASE [ dbalias ] ;
```

CLOSE TABLE

```
CLOSE TABLE tblref ;
```

COMMIT

```
COMMIT [ WORK ] [ <FOR>  dbbrand ] ;
```

CONTINUE

```
CONTINUE;
```

CREATE DATABASE

```
CREATE [ dbbrand ] DATABASE "dbname"
[ < IN > LOCATION "dbloc" ]
[ < AS > USER "user" [ < WITH > PASSWORD "password" ] ] ;
```

CREATE INDEX

```
CREATE [ UNIQUE ] INDEX "indexname" ON "tablename"
( fieldname [ < ASC > | < DESC > ] , … ) ;
```

CREATE TABLE

```
CREATE TABLE "tablename"
( fieldname fieldtype [ ( length [ , numdecs ) ] , … ) ;
```

DECLARE

```
[ DECLARE ] datatype  variable_name [ = initial_val ] ;
```

DELETE (positioned)

```
DELETE FROM tblref  WHERE CURRENT OF cursor ;
```

DELETE (searched)

```
DELETE FROM tblref [ WHERE where_condition ] ;
```

DESCRIBE COLUMNS

```
DESCRIBE COLUMNS <OF> tblref [ INTO cursor ] ;
```

DESCRIBE DATABASES

```
DESCRIBE [dbbrand] DATABASES [ <IN>  LOCATION dbloc ]
[ INTO [ cursor ];
```

DESCRIBE INDEXES

```
DESCRIBE INDEXES OF tblref  [ INTO cursor ];
```

DESCRIBE KEYS

```
DESCRIBE KEYS OF indexName  [INTO cursor];
```

DESCRIBE OPEN DATABASES

```
DESCRIBE OPEN DATABASES [ INTO cursor ] ;
```

DESCRIBE OPEN TABLES

```
DESCRIBE OPEN TABLES [ INTO cursor ] ;
```

DESCRIBE TABLES

```
DESCRIBE TABLES [ <OF>  dbalias ] [ INTO cursor ] ;
```

DESELECT

```
DESELECT [ cursor ] ;
```

DO

```
DO statement WHILE ( boolean_expression ) ;
```

DROP DATABASE

```
DROP DATABASE "dbname" [ <IN> LOCATION "dbloc" ];
```

DROP INDEX

```
DROP INDEX "indexName";
```

DROP TABLE

```
DROP TABLE "tablename";
```

ERROR

```
ERROR major_code [, minor_code, message_text, item1, item2 ];
```

ERRORCTL

```
ERRORCTL expression ;
```

EXECUTE FILE

```
EXECUTE FILE filename_strlit [<IN> LOCATION location_strlit];
```

EXECUTE FROM

```
EXECUTE FROM statement_strlit;
```

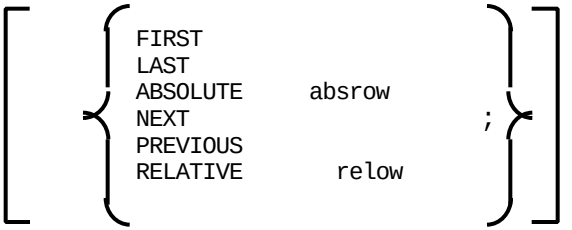
EXECUTE IMMEDIATE

```
EXECUTE IMMEDIATE "cmdstring" [ <IN> dbbrand ] ;
```

FETCH

```
FETCH [ motion ] [ OF cursor ] ;
```

where motion is:



FOR

```
FOR (initial_asg; test_expr; reinit_asg) statement
where initial_asg and reinit_asg have the form
 varname = expr
where reinit_asg has the form:
     varname ++
     varname --
```

FOR EACH

```
FOR EACH [ cursor ] statement
```

GOTO/LABEL

```
GOTO statement_label ;
LABEL statement_label : statement
```

IF

```
IF (boolean_expression) true_statement [ ELSE false_statement ]
         INSERT
INSERT INTO (colref_list)
[{ VALUES val_list
     querspec}]
   [ pos_spec ]
```

OPEN DATBASE

```
OPEN [ dbbrand ] DATABASE “dbname” ] [ ALIAS dbalias ]
[ <IN> LOCATION “dbloc” ] [ <AS> USER “user” [ <WITH>
PASSWORD “passwd” ] ]
[For [SHARED, PROTECTED, EXCLUSIVE]{READONLY, UPDATE}]
```

OPEN TABLE

```
[For [SHARED, PROTECTED, EXCLUSIVE]{READONLY, UPDATE}]
```

PRINT

```
PRINT expression_1 [ , expression_2 ] . . . ;
```

PRINTALL

```
PRINTALL [ cursor ] ;
```

PRINTCTL

```
PRINTCTL typespec1, typespec2, ... typespec15 ;
PRINTCTL srctypespec = dsttypespec, ...  ;
PRINTCTL srctypespec = procname ;
```

PRINTF

```
PRINTF ( format_expression, [ expression_list ] ) ;
```

PRINTINFO

```
PRINTINFO [ cursor ] ;
```

PRINTROW

```
PRINTROW [ cursor ] ;
```

PROCEDURE

```
PROCEDURE procname [ ( [ argname_list ] ) ]
   [RETURNS datatype_list ; ]
   [ argdecl_list ]
   { statement }
END PROCEDURE procname ;
```

RESTORE DATABASE

```
RESTORE <ALL> DATABASES FROM jrnl
<IN> LOCATION path ]
[ AFTER  afterTimeStamp ]
[ BEFORE beforeTimeStamp ];
RESTORE  DATABASE dbname FROM jrnl
[ <IN> LOCATION path ]
```

```
[ AFTER  afterTimeStamp ]
[ BEFORE beforeTimeStamp ];
```

RETURN

```
RETURN [ expr_list ] ;
```

ROLLBACK

```
ROLLBACK [ WORK ] [ <FOR> dbbrand ] ;
```

SELECT

```
  queryspec
[ ORDER BY sort_list ]
 [INTO cursor ]
[For{ READONLY, SCROLLING, EXTRACT, UPDATE [OF updcol_list]]
The queryspec format is:
SELECT [ ALL | DISTINCT ] select_list
                  FROM from_list
                  [ WHERE where_condition ]
                  [ GROUP BY group_list ]
                  [ HAVING have_condition ]
```

SET

```
[ SET ] variable_name = expression ;
[ SET ] variable_name + + ;
[ SET ] variable_name – – ;
```

SWITCH

```
SWITCH (expression) {
[ CASE comparison_expression :  statement_list ] . . .
[ DEFAULT: statement_list ]
             }
```

UNDECLARE

```
UNDECLARE variable_list ;
```

UPDATE (positioned)

```
UPDATE tableref SET assign_list WHERE CURRENT OF cursor ;
```

UPDATE (searched)

```
UPDATE tableref SET assign_list [ WHERE where_condition ] ;
```

USE DATABASE

```
USE DATABASE dbalias ;
```

USE LOCATION

```
USE LOCATION "loc" ;
```

WARNCTL

```
WARNCTL expression;
```

WHILE

```
WHILE (expression) statement
```

Appendix B • System variables, constants, and functions

This appendix presents a summary of Standard Query Language (SQL) and Butler system variables, constants, and functions. The Butler additions to the variables and functions are listed at the end of each category. The Butler SQL Programmer's Guide provides more information on these functions.

DAM System variables

$colcnt After each successful SELECT statement, automatically set by the SQL server to the number of columns in the rowset just created.

$cursor After each successful SELECT statement, automatically set by the SQL server to identify the rowset just created.

$datefmt Sets the date literal for input and output.

$maxrows Initially set to NULL and may be modified by a SQL SET statement. If set, it limits the maximum number of rows in a rowset created by a SELECT statement.

$month Allows redefinition of the text used for a month when a string MMM is specified in **$datefmt.**

$rowcnt After each successful SELECT statement, automatically set by the SQL server to the number of rows in the rowset just created. If the number of rows is unknown (that is, if the update mode of the SELECT statement is not EXTRACT), the variable is set to NULL.

$sqlcode Automatically set by the SQL server after execution of each data manipulation statement. Its value is zero (0) if no error occurred, positive for warning conditions (such as **$sqlnotfound**), and negative for errors.

$switch	When a SWITCH statement is executed, automatically set by the SQL server to the value of the "switch expression." The variable is set to this value before execution of the body of the SWITCH statement.
$timefmt	Sets the time literal for input and output.

Butler system variable extensions

$AutoResolveAlias	Governs the printing and saving (using the **$SaveFile( )** system function) of DOCUMENT alias values. If TRUE (the initial setting), the original document is printed or saved; if false, a Finder alias file referring to the original is printed or saved.
$currentDate	Contains the current server date.
$currentTime	Contains the current server time.
$currentTimeStamp	Contains the current server time and date.

DAM System constants

$FALSE	A SQL BOOLEAN FALSE value.
$NULL	A SQL NULL value.
$sqlnotfound	The value of $sqlcode that indicates a FETCH past the end of a rowset.
$TRUE	A SQL BOOLEAN TRUE value.
$version	The version number of the currently executing SQL server, stated as a VARCHAR string.
	In addition, each data type name (except GENERIC, OBJNAME, and CURSOR), prefaced by a dollar sign ($), is a system constant evaluating to the code of the type. For example **$boolean** evaluates to 1, and **$document** (on Butler servers only) evaluates to 28.

DAM System functions

$collen( )	Returns the length of a rowset column.
$colname( )	Returns the name of a rowset column.
$colplaces( )	Returns the number of decimal places in a rowset column.
$cols( )	Returns the number of columns in a rowset.
$coltype( )	Returns the data type code of a rowset column
$colwidth( )	Returns the display width of a rowset column
$format( )	Returns a formatted string built from its arguments according to a format specification.
$left( )	Returns a leading substring of its argument.
$len( )	Returns the length of its argument.
$locate( )	Finds the position of a pattern within a string.
$ltrim( )	Trims leading blanks from a string.
$right( )	Returns a trailing substring of its argument.
$rows( )	Returns the number of rows in a rowset.
$rtrim( )	Trims trailing blanks from a string.
$substr( )	Returns a substring of its argument.
$trim( )	Trims leading and trailing blanks from its string argument.
$typeof( )	Returns the SQL data type code of its argument.

Butler system function extensions

$freemem( )	Returns the amount of free server application memory, in bytes.
$LoadFileAliased( )	Returns a DOCUMENT value corresponding to the original file specified by its DOCUMENT alias argument.

$LoadFileNamed( )	Returns a DOCUMENT value corresponding to the file specified by its server file path argument.
$MakeAlias( )	Returns a DOCUMENT value corresponding to an alias of the file specified by its server file path argument.
$ResolveAlias( )	Returns a VARCHAR value which is the path of the original file specified by its DOCUMENT alias argument.
$SaveFile( )	Saves a DOCUMENT variable or column as a file in the server's file system. Returns a VARCHAR value which is the path of the saved file.
$ticks( )	Returns the number of ticks (one tick is approximately 1/60 of a second) that have elapsed since the server Macintosh was started.

Scalar functions

ODBC supports five categories of scaler functions: numeric, string, time and date, system, and conversion. The following sections will deal with each scalar function.

String functions

String functions include the exact substrings, perform case conversion, and determine string length. The following table describes the complete string functions.

ASCII(string_exp)	Returns the ASCII code value of the leftmost character of string_exp as an integer.
CHAR(code)	Returns the character that has the ASCII code value specified by code. The value of code should be between 0 and 255; otherwise, the return value is data source–dependent.

CONCAT(string_exp1, string_exp2)	Returns a character string that is the result of concatenating string_exp2 to string_exp1. The resulting string is DBMS dependent. For example, if the column represented by string_exp1 contained a NULL value, DB2 would return NULL, but SQL Server would return the non-NULL string.
INSERT(string_exp1, start,length, string_exp2)	Returns a character string where length characters have been deleted from string_exp1 beginning at start and where string_exp2 has been inserted into string_exp, beginning at start.
LCASE(string_exp)	Converts all upper case characters in string_exp to lower case.
LEFT(string_exp, count)	Returns the leftmost count of characters of string_exp.
LENGTH(string_exp)	Returns the number of characters in string_exp, excluding trailing blanks and the string termination character.
LOCATE(string_exp1, string_exp2[, start])	Returns the starting position of the first occurrence of string_exp1 within string_exp2. The search for the first occurrence of string_exp1 begins with the first character position in string_exp2 unless the optional argu-ment, start, is specified. If start is specified, the search begins with the character position indicated by the value of start. The first character posi-tion in string_exp2 is indicated by the value 1. If string_exp1 is not found within string_exp2, the value 0 is returned.

LTRIM(string_exp)	Returns the characters of string_exp, with leading blanks removed.
REPEAT(string_exp, count)	Returns a character string composed of string_exp repeated count times.
REPLACE(string_exp1, string_exp2, string_exp3)	Replaces all occurrences of string_exp2 in string_exp1 with string_exp3.
RIGHT(string_exp, count)	Returns the rightmost count of characters of string_exp.
RTRIM(string_exp)	Returns the characters of string_exp with trailing blanks removed.
SPACE(count)	Returns a character string consisting of count spaces.
SUBSTRING(string_exp, start, length)	Returns a character string that is derived from string_exp beginning at the character position specified by start for length characters.
UCASE(string_exp)	Converts all lower case characters in string_exp to upper case.

Numeric functions

Numeric functions include those functions that determine square roots, sine and cosine, and logarithms. The following table describes the complete numeric functions.

ABS(numeric_exp)	Returns the absolute value of numeric_exp.
ACOS(float_exp)	Returns the arccosine of float_exp as an angle, expressed in radians.
ASIN(float_exp)	Returns the arcsine of float_exp as an angle, expressed in radians.

ATAN(float_exp)	Returns the arctangent of float_exp as an angle, expressed in radians.
ATAN2(float_exp1, float_exp2)	Returns the arctangent of the x and y coordinates, specified by float_exp1 and float_exp2, respectively, as an angle, expressed in radians.
CEILING(numeric_exp)	Returns the smallest integer greater than or equal to numeric_exp.
COS(float_exp)	Returns the cosine of float_exp, where float_exp is an angle expressed in radians.
COT(float_exp)	Returns the cotangent of float_exp, where float_exp is an angle expressed in radians.
EXP(float_exp)	Returns the exponential value of float_exp.
FLOOR(numeric_exp)	Returns largest integer less than or equal to numeric_exp.
LOG(float_exp)	Returns the natural logarithm of float_exp.
MOD(integer_exp1, integer_exp2)	Returns the remainder (modulus) of integer_exp1 divided by integer_exp2.
PI( )	Returns the constant value of pi as a floating point value.
POWER(numeric_exp, integer_exp)	Returns the value of numeric_exp to the power of integer_exp.
RAND([integer_exp])	Returns a random floating point value using integer_exp as the optional seed value.
SIGN(numeric_exp)	Returns an indicator or the sign of numeric_exp. If numeric_exp is less than zero, –1 is returned. If numeric_exp

	equals zero, 0 is returned. If numeric_exp is greater than zero, 1 is returned.
SIN(float_exp)	Returns the sine of float_exp, where float_exp is an angle expressed in radians.
SQRT(float_exp)	Returns the square root of float_exp.
TAN(float_exp)	Returns the tangent of float_exp, where float_exp is an angle expressed in radians.

Time and Date functions

Time and date functions include the functions that extract time and date elements from a column and do time-based calculations. The following table describes the complete time and date functions.

CURDATE( )	Returns the current date as a date value.
CURTIME( )	Returns the current local time as a time value.
DAYOFMONTH(date_exp)	Returns the day of the month in date_exp as an integer value in the range of 1–31.
DAYOFWEEK(date_exp)	Returns the day to the week in date_exp as an integer value in the range of 1–7, where 1 represents Sunday.
DAYOFYEAR(date_exp)	Returns the day of the year in date_exp as an integer value in the range of 1–366.
HOUR(time_exp)	Returns the hour in time_exp as an integer value in the range of 0 – 23.
MINUTE(time_exp)	Returns the minute in time_exp as an integer value in the range of 0 – 59.

MONTH(date_exp)	Returns the month in date_exp as an integer value in the range of 1–12.
NOW( )	Returns current date and time as a timestamp value.
QUARTER(date_exp)	Returns the quarter in date_exp as an integer value in the range of 1– 4, where 1 represents January 1 through March 31.
SECOND(time_exp)	Returns the second in time_exp as an integer value in the range of 0 – 59.

System functions

System functions include the functions that returns the current user.

DATABASE( )	Returns the name of the database corresponding to the connection handle (hdbc). (The name of the database is also available by calling SQLGetConnectOption with the SQL_CURRENT_QUALIFIER connection option.)
IFNULL(exp, value)	If exp is null, value is returned. If exp is not null, exp is returned. The possible data type(s) of value must be compatible with the data type of exp.
USER( )	Returns the user's authorization name. (The user's authorization name is also available via SQLGetInfo by specifying the information type: SQL_USER_NAME.)

Explicit Conversion function

The data type conversion function converts a data type to a different data type on the server.

Syntax `CONVERT`(*value_exp, data_type*)

Example `{ fn CONVERT( { fn CURDATE() }, SQL_CHAR) }`

This example converts the output of the CURDATE scalar function to a character string. The following can all be supplied as data types for this conversion:

SQL_BIGINT,

SQL_LONGVARBINARY,

SQL_BINARY,

SQL_LONGVARCHAR,

SQL_BIT,

SQL_REAL,

SQL_CHAR,

SQL_SMALLINT,

SQL_DATE,

SQL_TIME,

SQL_DECIMAL,

SQL_TIMESTAMP,

SQL_DOUBLE,

SQL_TINYINT,

SQL_FLOAT,

SQL_VARBINARY,

SQL_INTEGER,

SQL_VARCHAR.

Notes The function returns the value specified by value_exp converted to the specified data_type, where data_type is one of the following keywords:

The ODBC syntax for the explicit data type conversion function does not support specification of conversion format. If specification of explicit formats is supported by the underlying data source, a driver must specify a default value or implement format specification.

The argument value_exp can be a column name, the result of another scalar function, or a numeric or string literal. For example:

Appendix C • DAL error codes

The information in this appendix is targeted towards those programming an application using DAL. It describes the error messages returned to the application by the DAL driver.

Success and warning status codes

Name	Value	Meaning
C_OK	0	successful completion
C_NOMORE	100	no more data
C_ISNULL	1	NULL data

Error status codes compatible with DB2

Name	Value	Meaning
CECHAR	–7	illegal character in statement
CEQUOTE	– 10	unterminated string constant
CESYNTAX	– 84	unacceptable statement
CECOMPLEX	– 101	statement too long or complex
CENUMLIT	– 103	invalid numeric literal
CETOKEN	– 104	invalid token
CESTRLIT	– 105	invalid string literal
CENAMLEN	– 107	identifier name too long
CECLAUSE	– 109	clause not permitted
CEFCNARG	– 111	function without column name
CEINVARG	– 112	nested functions or other illegal arguments

CENAMINV	– 113	invalid identifier name
CEPRDINV	– 115	invalid predicate
CEINSCNT	– 117	insert columns/values count mismatch
CESRCDST	– 118	same table is source and destination in UPDATE
CEHAVCOL	– 119	HAVING column not in GROUP BY clause
CEINVREF	– 120	where/set references function or group column
CEDUPCOL	– 121	duplicate column name in insert/update list
CESELGRP	– 122	select/group/aggregate clause disagreement
CESRTNUM	– 125	illegal column number in ORDER BY clause
CEUPDSRT	– 126	UPDATE and ORDER BY clauses conflict
CEDISTINCT	– 127	too many DISTINCT keywords
CEUSENUL	– 128	invalid use of NULL in predicate
CETBLCNT	– 129	too many table names
CELIKE	– 131	LIKE with incompatible data types
CELIKARG	– 132	LIKE has wrong arguments
CECORHAV	– 133	function on correlated reference in HAVING clause
CEVARBIN	– 134	illegal operation on long unstructured data
CESRTLEN	– 136	too many/too long SORT/GROUP BY columns
CENUPVIEW	– 150	update of “non-updateable” view
CENUPCOL	– 151	update of “non-updateable” column
CEGRPVIEW	– 155	FROM clause includes grouped view

CEINVVIEW	– 161	update violates VIEW constraints
CEKEYWORD	– 199	invalid use of keyword
CEAMBIG	– 203	ambiguous column reference
CEUDFIDN	– 204	reference to undefined identifier
CECOLTBL	– 205	unknown column for table
CEUDFCOL	– 206	unknown column name for any table
CECOLSRT	– 208	unknown column reference in ORDER BY clause
CEVARTYP	– 303	invalid data type for variable
CEVARRNG	– 304	data out of range for data type
CEUDFVAR	– 312	undefined/unusable variable name
CEOPTYP	– 401	incompatible operands for operator
CENUMCHR	– 402	numeric operation on character data
CECOLLEN	– 404	string too long for column on update/insert
CELITRNG	– 405	numeric literal out of range
CEVALRNG	– 406	calculated value out of range
CENONNUL	– 407	NULL assigned to non-NULL column
CECOLTYP	– 408	update/insert data has invalid type for column
CECNTOP	– 409	invalid operand of COUNT( ) function
CEFLTLEN	– 410	floating-point literal too long
CEUSER	– 411	keyword USER improperly used
CESUBCOL	– 412	illegal multicolumn subquery
CECNVOFL	– 413	numeric overflow during type conversion
CELIKNUM	– 414	LIKE on numeric column
CESCALE	– 419	negative scale on decimal division
CECRSNOP	– 501	cursor (or other object) not open

CECRSOPN	– 502	cursor (or other object) already open
CEUPDLST	– 503	column not in FOR UPDATE list
CEUDFCRS	– 504	specified cursor is not defined for this operation
CENOT2OPEN	– 507	cursor (update/delete) not open
CECURROW	– 508	cursor not positioned on a row
CECRSNUP	– 510	"non-updateable" cursor
CETBLNUP	– 511	"non-updateable" table
CEOBJPRV	– 551	privilege violation on object
CEOPNPRV	– 552	privilege violation for operation
CEEXISTS	– 601	object of given name already exists
CEOFLOW	– 801	arithmetic overflow
CEDUPROW	– 805	insert/update would create duplicate row
CESUBCNT	– 811	subquery produces multirow result
CEGRPHAV	– 815	illegal GROUP/HAVING in subquery
CECOLCNT	– 840	too many items in select/insert list
CESYSTEM	– 901	nonfatal system error
CEFATAL	– 904	fatal system error
CETXROLL	– 911	transaction rolled back, deadlock/time-out
CETXFAIL	– 913	transaction failed, deadlock/timeout
CECONAUTH	– 922	database open authorization failure
CENOCON	– 923	database not opened
CEDB2CON	– 924	DB2 internal connection error

SQL syntax errors

Name	Value	Meaning

CECOMMENT	– 10001	unterminated comment
CEDATLIT	– 10002	invalid date/time literal
CENOTHERE	– 10003	language construct not allowed in that context
CEUNDEF	– 10004	undefined identifier
CEEXP	– 10005	token expected
CEUNEXP	– 10006	token unexpected
CESYN	– 10007	syntax error
CEMISNG	– 10008	token missing
CEINVID	– 10009	invalid SQL identifier
CEILLOP	– 10010	invalid SQL operator use
CEREDCL	– 10011	illegal redeclaration of identifier
CESEM	– 10012	semantic error, construct does not make sense
CEILLASN	– 10013	illegal assignment to SQL variable
CEILLPRT	– 10014	printing not allowed now
CEILLTYP	– 10015	illegal type for operation or type mismatch
CENOVAL	– 10016	variable has no value yet
CERECUR	– 10017	recursion not allowed here
CEPCNT	– 10018	function call parameter count is wrong
CERCNT	– 10019	function call return item count is wrong
CENOQRY	– 10020	no query associated with (default) cursor
CENOFTCH	– 10021	no fetch done yet (no current row)
CEMXFTCH	– 10022	fetch beyond last (no current row)
CEFTCHOP	– 10023	bad fetch operand for fetch mode
CENOTXNS	– 10024	database does not support transactions
CEPRECIS	– 10025	loss of precision on conversion

CEUFLOW	– 10026	conversion resulted in underflow
CECOLVAL	– 10027	insert/update value illegal for column

SQL operational errors—object manipulation

Name	Value	Meaning
CEOPEN	– 10101	error opening object
CECLOSE	– 10102	error closing object
CERENAME	– 10103	error renaming object
CEDELETE	– 10104	error deleting object
CELOCK	– 10105	object locked by another user
CEISOPEN	– 10106	object is already open
CENOTOPEN	– 10107	object is not open
CEFILEIO	– 10108	file I/O error on access
CEPRTIO	– 10109	printer I/O error
CENETIO	– 10110	network I/O error
CEPROTOCOL	– 10111	protocol not supported
CENOMEM	– 10112	insufficient memory for operation
CEBREAK	– 10113	break request interrupted operation
CEUNIMPL	– 10114	unimplemented feature/function
CETOOMANY	– 10115	object/item maximum exceeded
CECOPY	– 10116	error copying object
CECREATE	– 10117	error creating object
CELOGIN	– 10118	invalid login parameters
CETIMOUT	– 10119	timeout on request
CEPARM	– 10150	illegal parameter
CEFORK	– 10151	unable to run subprocess

SQL execution errors—data manipulation

Name	Value	Meaning
CEBRAND	– 10200	unknown DBMS brand specified
CEDBREV	– 10201	incompatible DBMS revision levels
CEROWNR	– 10202	bad row number specified
CESUBTBL	– 10203	main and subquery FROM names duplicate table
CEUDFTBL	– 10204	undefined table name in database
CEDUPTBL	– 10205	duplicate table/synonym name in FROM list
CECATALOG	– 10206	error accessing database catalogs
CEDBMS	– 10207	untranslated DBMS error
CEALIAS	– 10208	undefined alias specified
CEUDFALS	– 10209	table (or alias) not in FROM list
CEDBOPEN	– 10211	no database is open

SQL client errors

Name	Value	Meaning
CENETIOC	– 10628	network I/O error on client
CENETMODE	– 10629	network send/receive mode error
CEIOPORT	– 10630	error opening I/O port
CEMODEM	– 10631	error on modem
CECONNECT	– 10632	connection error
CEVERSION	– 10634	version incompatibility
CESHUTDOWN	– 10635	could not shut down host server
CENOSERVER	– 10636	no server

CEATTMDM	– 10637	too many modem attempts
CEINIMDM	– 10638	could not initialize modem
CELOGPRM	– 10640	invalid login parameters
CEPASPRM	– 10641	invalid password parameter
CEAUTLOG	– 10642	error during auto-login
CEINIHST	– 10643	error initializing host
CEHUPMDM	– 10644	error hanging up modem
CENETLD	– 10646	cannot load network adapter
CECTRLBRK	– 10647	modem control error

SQL configuration and internal errors

Name	Value	Meaning
CECONFIG	– 10901	SQL server configuration problem
CEINTID	– 10902	invalid object ID specified
CEINTCTX	– 10903	internal—invalid transaction for context
CEINTCNT	– 10904	internal—count mismatch
CEINTTYP	– 10905	unknown data type in DBMS
CEINTTXN	– 10906	operation at wrong time in transaction
CEINTPARM	– 10907	internal parameter error
CEINTDBIF	– 10908	DBMS interface error
CEINTNET	– 10909	network interface error
CEINTOSIF	– 10910	operating system interface error
CEUNSPEC	– 10911	unspecified error

Appendix D • ODBC errors codes

The information in this appendix is targeted towards those programming an application using ODBC. It describes the error messages returned to the application by the ODBC driver.

Defined as a layered architecture, ODBC is a connection between an application and a data source. The complexity of an ODBC connection can vary between a simple conection consisting of the Driver Manager and a driver, or a complex connection consisting of the Driver Manager, a number of drivers, and a number of Database Management Systems.

As the complexity increases, the importance of providing consistent and complete error messages to the application, its users, and technical support personnel, also increases. Error messages must contain the identity of the component in which an error occurs, which is of particular importance to support personnel when an application uses ODBC components from more than one vector. This information must be included in the error text as it is not returned by SQLError.

Function Return codes

When an ODBC function is called by an application, the driver executes the function and returns a predefined code. These return codes can indicate different status information. Definitions for the return codes are given in the following table.

Table 9: Function return codes

Return Code	Definition
SQL_NEED_DATA	While processing a statement, the driver determined that the application needs to send parameter data values.
SQL_STILL_EXECUTING	A function that was started asynchronously is still executing.
SQL_INVALID_HANDLE	Function failed due to an invalid environment handle, connection handle, or statement handle. This indicates a programming error. No additional information is available from SQLError.
SQL_ERROR	Function failed. The application can call SQLError to retrieve error information.
SQL_NO_DATA_FOUND	All rows from the result set have been fetched.
SQL_SUCCESS_WITH_INFO	Function completed successfully, possibly with a nonfatal error. The application can call SQLError to retrieve error information.
SQL_SUCCESS	Function completed successfully; no additional information is available.

ODBC error codes

The character string value returned for a SQLSTATE (values returned by SQLError) consists of a two character class value succeeded by a three character subclass value. A class value of “01” indicates a warning and is accompanied by a return code of

SQL_SUCCESS_WITH_INFO. With the exception of class “IM”, class values other than “01” indicate an error and are accompanied by a return code of SQL_ERROR. “IM” class errors derive from the implementation of ODBC itself. The subclas value “000” in any class is for implementation defined conditions within the class itself. The assignment of class and subclass values is defined by ANSI SQL-92.

The table beginning on the following page lists SQLSTATE values that a driver can return for SQLError.

Table 10: ODBC Error codes

Error	Returned From	SQL State
General Warning	All ODBC functions except: SQLAllocEnv SQLError	01000
Disconnect error	SQLDisconnect	01002
Data truncated	SQLBrowseConnect SQLColAtrributes SQLDataSources SQLDescribeCol SQLDriverConnect SQLDrivers SQLExecDirect SQLExecute SQLExtendedFetch SQLFetch SQLGetCursorName SQLGetData SQLGetInfo SQLPutData SQLSetPos	01004
Privilege not revoked	SQLExecDirect SQLExecute	01006
Invalid connection string attribute	SQLBrowseConnect SQLDriverConnect	01S00
Error in row	SQLExtendedFetch SQLSetPos	01S01
Option value changed	SQLSetConnectOption SQLSetStmtOption	01S02

Table 10: ODBC Error codes

Error	Returned From	SQL State
No rows updated or deleted	SQLExecDirect SQLExecute SQLSetPos	01S03
More than one row updated or deleted	SQLExecDirect SQLExecute SQLSetPos	01S04
Wrong numbers of parameters	SQLExecDirect SQLExecute	07001
Restricted data type attribute violation	SQLBlindParameter SQLExtendedFetch SQLFetch SQLGetData	07006
Unable to connect to data source	SQLBrowseConnect SQLConnect SQLDriverConnect	08001
Connection in use	SQLBrowseConnect SQLConnect SQLDriverConnect SQLSetConnectOption	08002
Connection not open	SQLAllocStmt SQLDisconnect SQLGetConnectOption SQLGetInfo SQLNativeSql SQLSetConnectOption SQLTransact	08003
Data source rejected establishment of connection	SQLBrowseConnect SQLConnect SQLDriverConnect	08004

Table 10: ODBC Error codes

Error	Returned From	SQL State
Connection failure during transaction	SQL Transact	08007
Communication link failure	SQL BrowseConnect SQLColumnPrivileges SQLColumns SQLConnect SQLDriverConnect SQLExecDirect SQLExecute SQLExtendedFetch SQLFetch SQLForeignKeyes SQLFreeConnect SQLGetData SQLGetTypeInfo SQLParamData SQLPrepare SQLPrimaryKeyes SQLProcedureColumns SQLProcedures SQLPutData SQLSetConnectOption SQLSetStmtOption SQLSpecialColumns SQLStatistics SQLTablesPrivileges SQLTables	08S01
Insert value list does not match column list	SQLExecDirect SQLPrepare	21S01
Degree of derived table does not match column list	SQLExecDirect SQLPrepare SQLSetPos	21S02

Table 10: ODBC Error codes

Error	Returned From	SQL State
String data right truncation	SQLPutData	22001
Numeric value out of range	SQLExecDirect SQLExecute SQLExtendedFetch SQLFetch SQLGetData SQLGetInfo SQLPutData SQLSetPos	22003
Error in assignment	SQLExecDirect SQLExecute SQLGetData SQLPrepare SQLPutData SQLSetPos	22005
Datetime field overflow	SQLExecDirect SQLExecute SQLGetData SQLPutData SQLSetPos	22008
Division by zero	SQLExecDirect SQLExecute SQLExtendedFetch SQLFetch	22012
String data, length mismatch	SQLParamData	22026
Integrity constraint violation	SQLExecDirect SQLExecute SQLSetPos	23000

Table 10: ODBC Error codes

Error	Returned From	SQL State
Invalid cursor state	SQLColAttributes SQLColumnPrivileges SQLColumns SQLDescribeCol SQLExecDirect SQLExecute SQLExtendedFetch SQLFetch SQLForeignKeyes SQLGetData SQLGetStmtOption SQLGetTypeInfo SQLPrepare SQLPrimaryKeyes SQLProcedureColumns SQLProcedures SQLSetCursorName SQLSetPos SQLSetStmtOption SQLSpecialColumns SQLStatistics SQLTablesPrivileges SQLTables	24000
Invalid transaction state	SQLDisconnect	25000
Invalid authorization specification	SQLBrowseConnect SQLConnect SQLDriverConnect	28000
Invalid cursor name	SQLExectDirect SQLPrepare SQLSetCursorName	34000

Table 10: ODBC Error codes

Error	Returned From	SQL State
Syntax error or access violation	SQLExecDirect SQLNativeSql SQLPrepare	37000
Duplicate cursor name	SQLSetCursorName	3C000
Serialization failure	SQLExecDirect SQLExecute SQLExtendedFetch SQLFetch	40001
Syntax error or access violation	SQLExecDirect SQLExecute SQLPrepare SQLSetPos	42000
Operation aborted	SQLCancel	70100
Driver does not support this function	All ODBC functions except: SQLAllocConnect SQLAllocEnv SQLDataSources SQLDrivers SQLError SQLFreeConnect SQLFreeEnv SQLGetFunctions	IM001
Data source name not found and no default driver specified	SQLBrowseConnect SQLConnect SQLDriverConnect	IM002
Specified driver could not be loaded	SQLBrowseConnect SQLConnect SQLDriverConnect	IM003

Table 10: ODBC Error codes

Error	Returned From	SQL State
Driver's **SQLAllocEnv** failed	SQLBrowseConnect SQLConnect SQLDriverConnect	IM004
Driver's **SQLAllocConnect** failed	SQLBrowseConnect SQLConnect SQLDriverConnect	IM005
Driver's **SQLSetConnectOption** failed	SQLBrowseConnect SQLConnect SQLDriverConnect	IM006
No data source or driver specified; dialog prohibited	SQLDriverConnect	IM007
Dialog failed	SQLDriverConnect	IM008
Unable to load translation DLL	SQLBrowseConnect SQLConnect SQLDriverConnect SQLSetConnectOption	IM009
Data source name too long	SQLBrowseConnect SQLDriverConnect	IM010
Driver name too long	SQLBrowseConnect SQLDriverConnect	1M011
DRIVER keyword syntax error	SQLBrowseConnect SQLDriverConnect	IM012
Trace file error	All ODBC functions.	IM013
Base table or view already exists	SQLExecDirect SQLPrepare	S0001

Table 10: ODBC Error codes

Error	Returned From	SQL State
Base table not found	SQLExecDirect SQLPrepare	S0002
Index already exists	SQLExecDirect SQLPrepare	S0011
Index not found	SQLExecDirect SQLPrepare	S0012
Column already exists	SQLExecDirect SQLPrepare	S0021
Column not found	SQLExecDirect SQLPrepare	S0022
No default for column	SQLSetPos	S0023
General error	All ODBC functions except: SQLAllocEnv SQLError	S1000
Memory allocation failure	All ODBC functions except: SQLAllocEnv SQLError SQLFreeConnect SQLFreeEnv	S1001
Invalid column number	SQLBlindCol SQLColAttributes SQLDescribeCol SQLExtendFetch SQLFetch SQLGetData	S1002

Table 10: ODBC Error codes

Error	Returned From	SQL State
Program type out of range	SQLBlindCol SQLBlindParameter SQLGetData	S1003
SQL data type out of range	SQLBlindParameter SQLGetTypeInfo	S1004
Operation canceled	All ODBC functions that can be processed asynchronously: SQLColAttributes SQLColumnPrivileges SQLColumns SQLDescribeCol SQLDescribeParam SQLExecDirect SQLExecute SQLExtendedFetch SQLFetch SQLForeignKeyes SQLGetData SQLGetTypeInfo SQLMoreResults SQLNumParams SQLNumResultCols SQLParamData SQLPrepare SQLPrimaryKeyes	S1008

Table 10: ODBC Error codes

Error	Returned From	SQL State
Operation cancelled - continued	SQLProcedureColumns SQLProcedures SQLPutData SQLSetPos SQLSpecialColumns SQLStatistics SQLTablesPrivileges SQLTables	S1008
Invalid argument value	SQLAllocConnect SQLAllocStmt SQLBlindCol SQLBlindParameter SQLExectDirect SQLForeignKeys SQLAllocConnect SQLGetData SQLGetInfo SQLNativeSql SQLPrepare SQLPutData SQLSetConnectOption SQLSetCursorName SQLSetPos SQLSetStmtOption	S1009

Table 10: ODBC Error codes

Error	Returned From	SQL State
Function sequence error	SQLBlindCol SQLBlindParameter SQLColAttributes SQLColumnPrivileges SQLColumns SQLDescribeCol SQLDescribeParam SQLDisconnect SQLExecDirect SQLExecute SQLExtendedFetch SQLFetch SQLForeignKeyes SQLFreeConnect SQLFreeEnv SQLFreeStmt SQLGetConnectOption SQLGetCursorName SQLGetData SQLGetFunctions SQLGetStmtOption SQLGetTypeInfo SQLMoreResults SQLNumParams SQLNumResultCols SQLParamData SQLParamOptions SQLPrepare SQLPrimaryKeyes SQLProcedureColumns SQLProcedures	S1010

Table 10: ODBC Error codes

Error	Returned From	SQL State
Function error sequence -continued	SQLPutData SQLRowCount SQLSetConnectOption SQLSetCursorName SQLSetPos SQLSetScrollOptions SQLSetStmtOption SQLSpecialColumns SQLStatistics SQLTablesPrivileges SQLTables SQLTransact	S1010
Operation invalid at this time	SQLGetStmtOption SQLSetConnectOption SQLSetStmtOption	S1011
Invalid transaction operation code specified	SQLTransact	S1012
No cursor name available	SQLGetCursorName	S1015

Table 10: ODBC Error codes

Error	Returned From	SQL State
Invalid string or cursor length	SQLBlindCol SQLBlindParameter SQLColAttributes SQLColumnPrivileges SQLColumns SQLDescribeCol SQLDriverConnect SQLDrivers SQLExecDirect SQLExecute SQLForeignKeys SQLGetCursorName SQLGetData SQLGetInfo SQLNativeSql SQLPrepare SQLPrimary Keyes SQLProcedureColumns SQLProcedures SQLPutData SQLSetCursorName SQLSetPos SQLSpecialColumns SQLStatistics SQLTablePrivileges SQLTables	S1090
Descriptor type out of range	SQLColAttributes	S1091

Table 10: ODBC Error codes

Error	Returned From	SQL State
Option type out of range	SQLFreeStmt SQLGetConnectOption SQLGetStmtOption SQLSetConnectOption SQLSetStmtOption	S1092
Invalid parameter number	SQLBlindParameter SQLDescribeParam	S1093
Invalid scale value	SQLBlindParameter	S1094
Function type out of range	SQLGetFunctions	S1095
Information type out of range	SQLGetInfo	S1096
Column type out of range	SQLSpecialColumns	S1097
Scope type out of range	SQLSpecialColumns	S1098
Nullable type out of range	SQLSpecialColumns	S1099
Uniqueness option type out of range	SQLStatistics	S1100
Accuracy option type out of range	SQLStatistics	S1101
Direction option out of range	SQLDataSources SQLDrivers	S1103
Invalid precision value	SQLBlindParameter	S1104

Table 10: ODBC Error codes

Error	Returned From	SQL State
Invalid parameter type	SQLExtendedFetch	S1105
Fetch type out of range	SQLExtendedFetch	S1106
Row value out of range	SQLExtendedFetch SQLParamOptions SQLSetPos SQLSetScrollOptions	S1107
Concurrency option out of range	SQLSetScrollOptions	S1108
Invalid cursor position	SQLExecute SQLExecDirect SQLGetData SQLGetStmtOption SQLSetPos	S1109
Invalid driver completion	SQLDriverConnect	S1110

Table 10: ODBC Error codes

Error	Returned From	SQL State
Invalid bookmark value	SQLExtendedFetch	S1111
Driver not capable	SQLBlindCol SQLBlindParameter SQLColAttributes SQLColumnPrivileges SQLColumns SQLExecDirect SQLExecute SQLExtendedFetch SQLFetch SQLForeignKeys SQLGetConnectOption SQLGetData SQLGetInfo SQLGetStmtOption SQLGetTypeInfo SQLPrimary Keyes SQLProcedureColumns SQLProcedures SQLSetConnectOption SQLSetPos SQLSetStmtOption SQLSetScrollOption SQLStatistics SQLTablePrivileges SQLTables SQLTransact	S1C00

Table 10: ODBC Error codes

Error	Returned From	SQL State
Timeout expired	SQLBrowseConnect SQLColAttributes SQLColumnPrivileges SQLColumns SQLConnect SQLDescribeCol SQLDescribeParam SQLDriverConnect SQLExecDirect SQLExecute SQLExtendedFetch SQLFetch SQLForeignKeys SQLGetData SQLGetInfo SQLGetTypeInfo SQLMoreResults SQLNumParams SQLParamData SQLPrepare SQLPrimary Keyes SQLProcedureColumns SQLProcedures SQLPutData SQLSetPos SQLSpecialColumns SQLStatistics SQLTablePrivileges SQLTables	S1T00

Clearing an error

When SQLError is called for a handle and returns an error, that error is removed from the structure associated with that handle. When a handle is used in a subsequent function call, all errors stored for the handle are removed. The call to a function using an associated handle of a different type results in removing the errors stored on a given handle.

Driver Manager error checking

The Driver Manager checks and records function calls. When requested, the Driver Manager records:

- each called function in a trace file (after checking the function call for errors),
- the name of each function that does not contain errors detectable by the Driver Manager, and
- the values of the imput arguments and the names of the output arguments (as listed in the function definitions).

Also, the Driver Manager checks for function arguments, state transitions, and other error conditions before a call is passed to the driver associated with the connection. This checking reduces the amount of error handling that a driver needs to perform.

The driver does not check all state transitions, arguments, or error conditions for a given function.

Appendix E • Reserved Words

The following list contains all the reserved words for Butler SQL. While it is possible to use some of these reserved words when naming databases, columns, tables, and indices, it is not recommended.

$AUTORESOLVEALIASES
$BOOLEAN
$CHAR
$CHARACTER
$COLCNT
$COLLEN
$COLNAME
$COLPLACES
$COLS
$COLTYPE
$COLWIDTH
$CURRENTDATE
$CURRENTTIME
$CURRENTTIMESTAMP
$CURSOR
$DATE
$DATEFMT
$DEC
$DECIMAL
$DOCUMENT
$FALSE
$FLOAT
$FORMAT
$FREEMEM
$GENERIC
$GETRUNTIME
$ICON
$INT
$INTEGER
$LEN
$LOADFILEALIASED
$LOADFILENAMED
$LONGBIN
$LONGCHAR
$MAKEALIAS
$MAXROWS
$MONEY
$MOVIE
$NULL
$OBJNAME
$PICTURE
$PROCID
$REAL
$RESOLVEALIAS
$ROWCNT
$ROWS
$SAVEFILE
$SETRUNTIME
$SMALLINT
$SMFLOAT
$SMINT
$SOUND
$SQLCODE
$SQLNOTFOUND
$SUBSTR
$SWITCH
$TICKS
$TIME
$TIMEFMT
$TIMESTAMP
$TRIM
$TRUE
$TSFMT
$TYPEOF
$VARBIN
$VARCHAR
$VERSION

ABS
ABSOLUTE
ACOS
ACTIVITYLOG
ADD
AFTER
ALIAS
ALL
ALTER
AND
ANY
ARGUMENT
AS
ASC
ASCII
ASIN

ATAN
ATAN
AVG

BACKUP
BEFORE
BETWEEN
BIGINT
BIT
BREAK
BY

CALL
CASCADE
CASE
CEILING
CHAR
CHECK
CLOSE
COLUMN
COLUMNS
COMMIT
COMPACT
CONCAT
CONTINUE
CONVERT
COS
COT
COUNT
CREATE
CURDATE
CURRENT
CURTIME

DATABASE
DATABASES
DATE
DAYNAME
DAYOFMONTH
DAYOFWEEK
DAYOFYEAR
DBMS
DECIMAL
DECLARE
DEFAULT
DEFINED
DEGREES
DELETE
DESC
DESCRIBE
DESELECT
DISTINCT
DO
DOCUMENT
DOUBLE
DROP

EACH
ELSE
END
ERROR
ERRORCTL
EXCLUSIVE
EXECUTE
EXISTS
EXP
EXTRACT

FALSE
FETCH
FILE
FIRST
FLOAT
FLOOR
FOR
FOREIGN
FORMAT
FROM

GENERIC
GOTO
GROUP

HAVING
HOUR

ICON
IF
IFNULL
IMMEDIATE
IN
INCREMENT
INDEX
INDEXES
INDICIES
INSERT
INT
INTEGER
INTO

IS
JOIN
JOURNAL
KEY
KEYS
LABEL
LAST
LCASE
LEFT
LENGTH
LIKE
LINKSETS
LOCATE
LOCATION
LOG
LONG
LONGBIN
LONGCHAR
LTRIM
MAX
MAXVALUE
MIN
MINUTE
MINVALUE
MOD
MODIFY
MONEY
MONTH
MONTHNAME
MOVIE
NAME
NATIONAL
NCHAR
NEXT
NOMAXVALUE
NOMINVALUE
NOT
NOW
NULL
NUMBER
NUMERIC
OBJNAME
OF
OJ
ON
OPEN
OPTION
OR
ORDER
OUTER
PASSWORD
PI
PICTURE
POWER
PRECISION
PREVIOUS
PRIMARY
PRINT
PRINTALL
PRINTCTL
PRINTF
PRINTINFO
PRINTROW
PROCEDURE
PROTECTED
QUARTER
RADIANS
RAND
READONLY
REAL
REFERENCES
RELATIVE
REPEAT
REPLACE
RESTORE
RESTRICT
RETURN
RETURNING
RETURNS
RIGHT
ROLLBACK
ROUND
RTRIM
SCROLLING
SECOND
SELECT
SEQUENCE
SESSION
SET

SHARED
SIGN
SIN
SMALLINT
SMFLOAT
SMINT
SOME
SOUND
SPACE
SQLCODE
SQLNOTFOUND
SQL_BIGINT
SQL_BINARY
SQL_BIT
SQL_CHAR
SQL_DATE
SQL_DECIMAL
SQL_DOUBLE
SQL_FLOAT
SQL_INTEGER
SQL_LONGVARBINARY
SQL_LONGVARCHAR
SQL_NUMERIC
SQL_REAL
SQL_SMALLINT
SQL_TIME
SQL_TIMESTAMP
SQL_TINYINT
SQL_TSI_DAY
SQL_TSI_HOUR
SQL_TSI_MINUTE
SQL_TSI_MONTH
SQL_TSI_QUARTER
SQL_TSI_SECOND
SQL_TSI_WEEK
SQL_TSI_YEAR
SQL_VARBINARY
SQL_VARCHAR
SQRT
START
SUBSTRING
SUM
SWITCH
SYSTEM

TABLE
TABLES
TAN
TIME
TIMEFMT
TIMEOUT
TIMESTAMP
TIMESTAMPADD
TINYINT
TO
TRUNCATE

UCASE
UNDECLARE
UNIQUE
UPDATE
USE
USER

VALUES
VARBIN
VARCHAR
VARIABLE
VARYING

WARNCTL
WEEK
WHERE
WHILE
WITH
WORK

YEAR

Glossary

access privileges

The permissions granted to a user to access information in a database. There are eight different access privileges assigned in ButlerTools.

activity log

A record of the operations of Butler SQL or ButlerTools. The Activity Log record, functions, errors, executed, and user activity, along with the date and time when each operation occurred.

administrator

A user selected to control and protect access to certain Butler SQL functions. When an administrator is defined, access to these functions is restricted to those users who provide the administrator's password.

aggregate function

An aggregate function is used to compute a statistical function of the values in a particular column over all of the rows in a row group.

alias

An alias is a SQL identifier that is used in subsequent references to identify a SQL object in place of its real name. SQL supports several different kinds of aliases: a database alias (with the OPEN DATABASE statement), a table alias (in the FROM clause of a SELECT statement), and a column alias (in the select list of a SELECT statement).

Apple Shared Library Manager (ASLM)

An extension of MacOS responsible for loading shared libraries of program code and providing access to the functions in the libraries. Shared libraries are used to extend the functionality of application.

API

Application Programming Interface. An API is the definition of the method to be used to extend the functionality of an application.

arithmetic (numeric) operators

SQL supports five arithmetic operators: addition (+), multiplication (*), division (/), and subtraction (-). All of these operators can be used with numeric columns.

ascii

ASCII is the acronym for American Standard Code for Information Interchange. Each character that can be produced on a standard keyboard is identified by its ascii value.

case sensitive

The need to use either upper or lowercase commands. Case sensitivity recognizes the case in which data exists when storing data. In Butler, commands are not case sensitive.

calculated column

A column produced by specifying an expression instead of a table column reference.

catalogue

A catalogue is a facility of a DBMS that describes how a data source is structured. Information about tables, columns, keys, stored procedures, and access privileges are contained in a catalogue.

column

Columns make up the structure of a table. Columns are made up of data items of a single type, i.e. a CHAR type column is made up of

data items containing character-based data.

comment

Comments can be used to improve a SQL procedure's readability. A comment can appear anywhere there's white space within a SQL procedure. A comment begins with a slash followed by an asterisk (/*) and ends with an asterisk and a slash (*/). Comments can be nested.

compacting

A process whereby the space formally used by a deleted data item, table, or index is freed, allowing it to be used for new data items.

comparison operator

SQL includes six comparison operators, which perform comparisons among data values to produce a BOOLEAN result. Each comparison operator produces a NULL result if any of its operands are NULL; otherwise the result is TRUE or FALSE. All of these operators support both string and numeric data types. For compatibility with both C and SQL programming styles, SQL supports alternate notations for equality and inequality. Comparison operators include: eqaluity (a = b or a == b), inequality (a= b or a<> b), less than (a < b), greater than (a > b), less than or equal (a <= b), and greater than or equal (a>=b).

compound statement

Compound statements provide a limited block structure for SQL programs. A compound statement is a sequence of SQL statements that are grouped together to function as a single statement. The sequence is enclosed in a left brace/right brace pair ({ }). When the compound statement executes, each statement in the sequence executes consecutively, as it appears.

cursor

A cursor is a pointer that uniquely identifies a rowset and designates

one of its rows as the current row. When the SQL SELECT statement forms a rowset, it automatically creates a cursor that identifies the rowset. Subsequent data-manipulation statements can access the rowset and its component rows and columns by referencing the cursor. Finally, the SQL program destroys the cursor, deactivating the associated rowset when it is no longer needed. Thus, the cursor acts as an identifier for the rowset and its current row.

cursor-based reference

A cursor-based row reference identifies the current row as a unit and has the form. A SQL program has access to the current row of a cursor and the values of its columns through a cursor-based reference that uses the name of the cursor variable that holds the cursor.

DAL

Data Access Language - Refers to three different things: the manager, SQL dialect, and the driver. DAL connections use the “Butler DAL” database extension to connect to Butler SQL or Apple's 'DAL' database extension to connect to other DAL servers. This connection is configured using the ButlerHosts application and by editing the DAL Preferences file.

DAM

Data Access Manager - Refers to the manager and driver components with support for the DAL/SQL dialect. This is more robust than pure DAL and is preferred over the pure DAL connection whenever possible. A DAM connection uses the Butler Access database extension to connect to Butler SQL. This connection is configured using the ButlerHosts application.

data item

A single piece of information that cannot be divided. A data item occupies one cell in a database table.

data tables

Butler SQL databases are made up of data tables. A data table is organized into rows and columns. Each row in a data table is the equivalent of a record in a text file, and each column element (or cell) is the equivalent of a single field. A database can have any number of data tables.

data type

A data type defines the type of data in a table column. Columns can be assigned data types of CHAR, VARCHAR, LONGCHAR, DATE, TIMESTAMP, TIME, DECIMAL, SMALL FLOAT, FLOAT, INTEGER, PICTURE, SOUND, ICON, MOVIE, DOCUMENT, BOOLEAN, and SMALL INTEGER.

database

A collection of related information organized in a logical manner for ease of retrieval and storage. This information can be accessed by applications such as ButlerClient. Butler SQL databases use a unified language for defining, querying, modifying and controlling the data in a database.

DBMS

Database Management System. A software application which stores, manages, and allows retrieval of data. When working with Butler SQL, using the DAL/DAM API, you need to specify Butler's DBMS, called RMR, in order to log on to the server. When using ODBC, you do not need to use any of the commands or parameters that refer to DBMS's.

duplicate keys

Keys containing the same value. When duplicate keys are allowed in an index, data items which are the same according to key length will be seen as equal. When duplicate keys are not allowed, the index will

prevent entry of duplicate data.

free-space

Empty space in a database caused by deleting data. Free-space can be used when importing, but such a requirement of the importing function slows it down to some extent.

groups

A collection of users who are arranged into groups, where the members of each group have a common characteristic and a common password.

identifier

A SQL identifier is a name used to identify a SQL entity (such as a host system, database, table, or column) or a SQL variable. Some identifiers, such as table and column names, are inherited from the SQL environment. Other identifiers, such as variable names and aliases, are created dynamically by a SQL procedure.

index

An ordering of the data in one or more of a table's columns used to speed access and sorting of the table's data.

index caching

Index caching improves performance of inserts, updates and deletes significantly when many records with indexes are being processed. Each index cache requires about 5K of Butler's RAM when turned on. This memory is freed when caching is turned off. Therefore a table with 5 indexes will require 25K of RAM. These settings are application wide, and will affect all users when in effect. Caches are in effect until the termination of Butler or explicitly turned off by any user.

initial value

The value specified as a default value for a data item when a new row is added to a table. For example, the initial value for a date column may be the current date.

IS NULL operator

The IS NULL operator is a unary operator that checks whether its operand has a NULL value. It accepts an operand of any valid SQL type and produces a BOOLEAN result. The operator is written with a postfix notation.

join conditions

Join definitions describe the access paths used to propagate from one table to another within the execution of the database access plan.

key

A column or set of columns whose contents can be used to identify a row in a database table. Key values are often used to locate specific rows in a table.

key size

The length of the key. Usually the same as the size of the columns that comprise the key, but may be customized so that only the partial contents of a column is contained in the key.

keywords

The fixed words in SQL statements that identify the statement and its various clauses.

keyed column

A column included in an index.

literals

SQL literals represent constant values in a SQL program. Literal representations are provided for most of the SQL data types. As an example, a boolean literal represents a BOOLEAN constant. There are two boolean literals, $TRUE and $FALSE (representing the TRUE and FALSE values, respectively).

localized

Localized refers to the language used by the current server. When data is exported in a specific, localized language, it must be received by a server with the same localized language.

logical operator

Operators that operate on values to produce BOOLEAN (true or false) results.

normalized (relation)

A relation, without repeating groups, such that the values of the data elements could be represented as a two-dimensional table.

ODBC

Open Database Connectivity. An extension to the computer's operating system that provides applications with access to database or other data sources independent of the database's platform or underlying protocols.

ODBC expression

An expression can be a literal, an identifier, or an operator applied to one or more expressions. ODBC literals, variables, and other identifiers (such as column references) can be combined in expressions to calculate parameters that are used when ODBC statements are executed.

outer block

The sequence of statements forming the main body of a SQL program is called the outer block. The simplest SQL programs contain *only* an outer block, which opens a database, performs a query, returns the results, and closes the database.

owner

The owner of a database table. The owner when the table is created using ButlerTools or using CREATE TABLE SQL statements.

password

A word which must be entered in order to connect a user to the Butler SQL database. The password is used to restrict access to the database.

pathname

A name specifying the location of a file in a directory.

ports

Butler allows communication with clients through "ports", or access channels into the server. There are three different ports supported by Butler SQL: program linking (or PPC) ports, Communications Toolbox (or CTB) ports, and TCP IP ports.

precision

Defines the number of digits to the right of the decimal place in a floating-point column.

primary key

A key which uniquely identifies a row in a table.

primary table

During the projection from the database, the primary table is the first table which is read. All joined tables' rows are derived from this table's content. There can be one and only one primary table within a query.

procedure

To perform a complete connectivity task, SQL statements are combined to form a procedure, which is simply a sequence of one or more statements. Execution of a procedure begins with its first statement and normally continues sequentially, statement by statement, until the sequence is completed. The execution of SQL procedure control statements, such as IF or GOTO, can alter or interrupt the linear flow of the procedure's execution.

projection

The process of extracting the data from relational tables into a result rowset as a part of the query specification.

qualified identifier

To identify a component part of an entity unambiguously, use a qualified identifier. A qualified identifier consists of a sequence of simple identifiers separated by punctuation characters. Database names and table names are separated by an exclamation point (!) while column and table names are separated by a period (.).

query

A request for information from the database based on specific conditions you specify.

query specification

A query specification *(qryspec)* specifies a SQL rowset and describes

its contents. A query specification forms the main body of the SQL SELECT statement, which creates rowsets for row-by-row manipulation. A query specification can also appear in a search condition of a DELETE or UPDATE statement, selecting rows to be deleted or modified. In addition, it can be used to specify a rowset that is inserted as a unit into another table by the SQL INSERT statement.

range

Defines a set of values that are to be returned by a query.

relational database

A database whose logical structure is based on one or more tables of data.

resource fork

The fork of a file that contains the file's resources.

resource ID

The identifier for a particular resource type in a resource file.

row

A row is a set of related columns that describe a single record in a table.

rowid

A pseudo-column that can be returned by Butler SQL queries. Rowid is only guaranteed to be unique and valid within a session. Rowid is generally used only by DAL/DAM client applications.

rowset

A rowset is a collection of rows and columns from a database that has

the same characteristics as a table. SQL supports set-level access to data in a host database through the concept of a rowset. All SELECT and DESCRIBE statements output data in rowsets.

scale

Defines the maximum number of digits to the right of the decimal point.

scaler functions

A function that allows the manipulation of values. ODBC supports five categories of scaler functions: numeric, string, time and date, system, and conversion.

schema

An outline of the format by which records are organized in a database table.

search arguments

The attribute value(s) which are used to retrieve some data from a data source, whether through an index, or by a search.

self-joins

Self-joins can be defined as table selections where the table referred to on either side of the join equation are to the same physical table. In other words, when the FROM clause of a query specification contains the same table more than once (with table aliases used to differentiate) then the query is said to contain a self-join.

server

A computer on a network that provides services and facilities to client applications.

SQL

Structured Query Language is a language used for the management of data in a Butler SQL database. These commands let the user add, update, delet, or retrieve data.

SQL entity

A SQL entity are items identified by SQL such as a host system, database, table, or column.

SQL statements

Butler SQL statements use a number of basic language elements. Some of these elements are the same as those found in programming languages such as C or Pascal. Other elements are related to Level 1 of the ANSI SQL standard, which defines data manipulation language statements. All SQL statements employ SQL style, with an initial verb, one or more English-like clauses, and a statement terminator.

stored procedure

A program residing on a data source. Stored procedures can be invoked to perform specialized DBMS operations on the data source.

string

A data type that refers to an imput item that is made up of characters, numbers, or words. An example of string data types are all alphanumeric data types.

string concatenation operator

This operator produces a concatenated string whose length is the sum of the lengths of the arguments. If either of the arguments of the concatenation operator is NULL, the result is NULL. The infix string concatenation operator (+) can be applied to any string data type (CHAR or VARCHAR).

table

A table is the basic storage structure of a Butler SQL database. It is a two dimensional structure made up of rows and columns.

table reference

Tables and views within a database are identified in SQL statements by a table reference *(tblref)* with the following syntax: [*dbalias*!] [*tblgrp*.] . . . *tblname*.

transaction

A transaction is a series of database commands that modify data in the database, and that has a specific termination point; when the transaction is terminated, all the modifications are either made permanent in the database, or they are reversed so that the database reverts to the same state it was in before the transaction was started. The key is that either all commands of a transaction are successfully performed on the database, or none of them are.

undo

Reverses the last action carried out by the user. It is used when editing or modifying data.

unique columns

Unique columns are a column or group of columns which uniquely identify each row of a table (e.g. "user_id").

unsuccessful verification

When the data stored in a database does not conform with the format of Butler SQL databases and the integrity of the database is faulted.

user

A person granted or denied access privileges to the database.

variable

A SQL variable contains data that is referenced and manipulated during the execution of a SQL program. A SQL variable has a variable name that is a valid SQL identifier, an associated SQL data type, and must be explicitly declared before it is used (the exception is CURSOR variables).

verifying

An evaluative process designed to determine if the data stored in a database is stored in a way which conforms with the format of Butler SQL databases.

Index

Symbols

A

B

C

D

E

F

T

U

V

W

Disclaimer of Warranty and Limited Warranty on Media

EveryWare will replace defective distribution media or manuals at no charge, provided you return the item to be replaced with proof of purchase to EveryWare during the 90-day period after purchase. ALL IMPLIED WARRANTIES ON THE MEDIA AND MANUALS, INCLUDING WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE, ARE LIMITED IN DURATION TO NINETY (90) DAYS FROM THE DATE OF THE ORIGINAL RETAIL PURCHASE OF THIS PRODUCT.

EVERYWARE MAKES NO WARRANTY OR REPRESENTATION, EITHER EXPRESS OR IMPLIED, WITH BUTLER, ITS QUALITY, PERFORMANCE, MERCHANTABILITY, OR FITNESS FOR A PARTICULAR PURPOSE. AS A RESULT, BUTLER IS LICENSED "AS IS"AND YOU THE PURCHASER ARE ASSUMING THE ENTIRE RISK AS TO ITS QUALITY AND PERFORMANCE.

EVERYWARE MAKES NO WARRANTY OR REPRESENTATION, EITHER EXPRESS OR IMPLIED, WITH BUTLER, ITS QUALITY, PERFORMANCE, MERCHANTABILITY, OR FITNESS FOR A PARTICULAR PURPOSE. AS A RESULT, BUTLER IS LICENSED "AS IS"AND YOU THE PURCHASER ARE ASSUMING THE ENTIRE RISK AS TO ITS QUALITY AND PERFORMANCE.

IN NO EVENT WILL EVERYWARE BE LIABLE FOR DIRECT, INDIRECT, SPECIAL, INCIDENTAL, OR CONSEQUENTIAL DAMAGES RESULTING FROM ANY DEFECT IN BUTLER OR ITS DOCUMENTION, even if advised of the possibility of such damages. In particular, EveryWare shall have no liability for any programs or data stored using Butler on data storage devices of any description, including the cost of recovering such programs or data.

THE WARRANTY AND REMIDIES SET FORTH ABOVE ARE EXCLUSIVE AND IN LIEU OF ALL OTHERS, ORAL OR WRITTEN, EXPRESS OR IMPLIED. No EveryWare Development distributor, dealer, agent, or employee is authorized to make any modification, extension, or addition to this warranty. Some states do not allow exclusion or limitation of implied warranties or liability for incidental or consequential damages, so the above limitation or exclusion may not apply to you. This warranty gives you specific legal rights, and you may also have other rights which vary from state to state.

Apple Documentation Disclaimer

IN NO EVENT WILL APPLE, ITS DIRECTORS, OFFICES, EMPLOYEES OR AGENTS BE LIABLE TO YOU FOR ANY CONSEQUENTIAL, INCIDENTAL DAMAGES (INCLUDING DAMAGES FOR LOSS OF BUSINESS PROFITS, BUSINESS INTERRUPTION, LOSS OF BUSINESS INFORMATION, AND THE LIKE) ARISING OUT OF THE USE OR INABILITY TO USE THE APPLE DOCUMENTATION EVEN IF APPLE HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES. BECAUSE SOME STATES DO NOT ALLOW THE EXCLUSION OR LIMITATION OF LIABILITY FOR CONSEQUENTIAL OR INCIDENTAL DAMAGES, THE ABOVE LIMITATIONS MAY NOT APPLY TO YOU.

Copyright Notice

The Butler manual, program design, and design concepts are copyrighted, with all rights being subject to the limitations and restrictions imposed by the copyright laws of the United States of America and Canada. Under the copyright laws, this manual may not be copied, in whole or part, including translation to another language or format, without the express written consent of EveryWare Development Corporation.

2.0 2-96

Publishing Tools

This manual was created on a Macintosh using FrameMaker from Frame Technology Corporation.