# Chapter 1 • Basic SQL Elements

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.

This guide describes the basic language elements that are used in statements in Butler SQL's SQL dialect.

## Conventions

The following conventions are used throughout this and the following chapter.

| | |
|---|---|
| [ ] | Terms placed between square brackets indicate optional syntax items unless the syntax description indicates that you should include the brackets as literal characters in the statement. For example, you use brackets to enclose a column index. |
| < > | Terms placed between angle brackets are noise keywords. (See "Keywords" on page 8.) |
| { } | Terms placed between braces indicate required syntax items unless the syntax description tells you to include the braces as literal characters in the statement. For example, you use braces to enclose compound statements. |
| . . . | An ellipsis indicates a variable that may be repeated one or more times. |
| \| | Vertical lines separate alternate syntax items. |

| | |
|---|---|
| := | This symbol is used to separate the term that is defined (left) from the definition (right). It should be read as "is defined as." |
| *italics* | Terms that appear in italics indicate variables that you supply. |
| UPPERCASE | Terms that appear in UPPERCASE LETTERS identify SQL keywords. You may use either uppercase or lowercase letters in actual usage. |
| **boldface** | Terms that appear in bold identify API functions, and system variables. |

In a statement you should include any other punctuation symbols that appear in the syntax. Note that spaces have been inserted in syntax examples in these chapters for legibility and are not significant in actual syntax unless otherwise noted.

## Statement structure

The basic SQL unit of execution is the statement. Each SQL statement performs a single, well-defined task, such as opening a host database, updating a database record, or testing the value of a SQL variable. All SQL statements have the same basic structure. A statement begins with a verb that identifies the statement and describes its action. The statement then contains zero or more clauses that specify additional actions, objects to be acted upon, and how the action is to be carried out. Most SQL statements end in a semicolon, marking the end of the last clause.

SQL statement structure can be seen in the following example:

```
open dbms;
open database "accounting" as user "joe" ;
select dept nr, budget, actual from expenses into x ;
fetch next of x ;
if (budget > actual)
  print "Dept ", dept_nr, " over budget by ", (actual - budget);
```

A SQL verb always consists of one or two words. In the previous example, the verb OPEN DATABASE causes a database to be opened for processing. The verb IF causes testing of a condition. The PRINT verb generates output from the SQL program.

Clauses in a statement identify the SQL object to be acted upon by the verb or describe how the action is to be carried out. In the example, the AS USER clause in the OPEN DATABASE statement specifies how the "accounting" database is to be opened. The NEXT OF X clause in the FETCH statement tells SQL to retrieve the next row out of cursor X. Clauses are usually a mixture of keywords (such as NEXT) and identifiers (such as EXPENSES). In many cases, SQL clauses are optional, with SQL providing reasonable defaults.

White space between the tokens (words or symbols) in a SQL statement is not significant except as a delimiter for the tokens. You can freely insert spaces, tabs, end-of-line characters, and comments between tokens to improve readability.

## Program structure

To perform a complete connectivity task, SQL statements are combined to form a program, which is simply a sequence of one or more statements. Execution of a program begins with its first statement and normally continues sequentially, statement by statement, until the sequence is completed. The execution of SQL program control statements, such as IF or GOTO, can alter or interrupt the linear flow of the program's execution.

An unusual feature of the Butler runtime environment is that a SQL program is usually executed in a series of fragments, rather than all at once. The client application passes a sequence of statements, requests their execution, passes another statement sequence, requests their execution, and so on. The sequence of statements passed between execution requests is a program fragment. A program fragment always contains an integral number of SQL statements. A statement cannot be split across two fragments.

The following example (using the DAL API) illustrates how two program fragments are sent and executed. Assume that a SQL session has already been initiated by a **CLInit( )** function call and that data declaration statements have been sent and executed in previous program fragments.

1. **CLSend( ) call text: DESCRIBE DATABASES. . .**
2. **CLSend( ) call text: PRINTALL. . .**
3. **Execute first fragment through CLExec( ) call.**
4. **Check status through CLState( ) and when state is ready, get descriptive data through calls to the CLGetRow( ) function.**
5. **CLSend( ) call text: OPEN INGRES DATABASE . . .**
6. **CLSend( ) call text: SELECT . . .**
7. **Execute second fragment through CLExec( ) call.**

The following example illustrates how two program fragments are sent and executed. Assume that a SQL session has already been initiated by a DBInit( ) function call and that data declaration statements have been sent and executed in previous program fragments.

1. **DBSend( ) call text: DESCRIBE DATABASES . . .**
2. **DBSend( ) call text: PRINTALL . . .**
3. **Execute first fragment through DBExec( ) call.**
4. **Check status through DBState( ) and when state is ready, get descriptive data through calls to the DBGetItem( ) function.**
5. **DBSend( ) call text: OPEN DATABASE . . .**
6. **DBSend( ) call text: SELECT . . .**
7. **Execute second fragment through DBExec( ) call.**

For the most part, two SQL program fragments executed in sequence behave exactly as if they were executed as a single fragment. From one fragment execution to the next

- all variables maintain their values
- procedure definitions persist
- open database management systems and databases remain open
- active cursors remain active

Backward references to SQL statements in a previous program fragment are not permitted. For example, SQL does not permit a GOTO statement that attempts to jump backwards into a previous program fragment.

## Compound statements

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.

Compound statements are often used in the body of an IF statement, as shown in the following example:

```
/* Calculate total orders on cycle # 4 */
if (cycle = 4) {
   select sum(ytd_sales) as sales, sum(ytd_orders) as
   orders from staff ;
   fetch ;
   print sales, orders, (sales/orders) ;
}
```

Compound statements often make up the body of a WHILE, DO, SWITCH, or PROCEDURE statement.

## Procedures

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, as in the following example, which opens a database, performs a query, returns the results, and closes the database:

```
/* A SQL program with only an outer block */
open dbms ;
open database "receivables" ;
select cust_nr, cust_name, rcvbl_amount from customers ;
printall ;

close database ;
close dbms ;
```

Butler SQL supports structured programs through the use of SQL procedures. A procedure is a sequence of SQL statements that is assigned a name. It can take zero or more arguments as its input and produce zero or more results as its output. Once the procedure is defined, it can be invoked repeatedly by name, which eliminates the need to specify the statements of the procedure at each invocation. For example, if a client application frequently needs to obtain a customer's name and receivable amount, given a customer number, a procedure can be defined to perform that function, as the following example demonstrates:

```
/* Define a SQL procedure "getinfo" */
procedure getinfo(custnum)
returns varchar, money ;
argument integer custnum ;
{
  select cust_name, rcvbl_amount from customers
    where customers.cust_nr = custnum ;
  fetch ;
  return cust_name, rcvbl_amount ;
}
end procedure getinfo ;
```

To obtain information for customer number 12345 later in the SQL session, the client application simply calls the procedure and uses the resulting data:

```
call getinfo(12345) returning name, amount ;
if (amount > 10000)
  print " Customer ", name, " balance: ", amount ;
```

The appearance of the PROCEDURE statement in a SQL program's outer block does not cause execution of the procedure; it merely defines the procedure to Butler SQL. The statements that appear as the body of the PROCEDURE statement—those between the PROCEDURE line and the END PROCEDURE line (the SELECT and RETURN statements between the braces in the example)—form the procedure definition. They are not part of the outer block. These statements only execute when the CALL statement invokes the procedure.

## Comments

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

The following example shows a SELECT statement with embedded comments:

```
/* A query with comments in some strange places */
select ord_amount, ord_date, cust_nr /* select three columns */
  from orders
  where /* select only large orders */ ord_amount > 20000
  order by ord_date /* most recent first */ desc ;
```

## Keywords

The fixed words in SQL statements that identify the statement and its various clauses are called keywords. For example, in the statement

```
open database "accounting" as user "joe" ;
```

the words OPEN, DATABASE, AS, and USER are keywords. Some keywords (such as the verb OPEN DATABASE in this example) are required; the SQL statement must include them.

Other keywords (such as the word AS in this example) are optional. These keywords, called *noise keywords,* are prepositional or pointing words such as ON, IN, and TO. They make the SQL statement easier to read and understand. You may omit noise keywords without changing the meaning of the statement. You may also omit them if the default meaning of the statement is used.

## Identifiers

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, can be created dynamically by a SQL program.

A SQL identifier:

- is a name of up to 31 characters,
- must begin with a letter,
- may contain letters, numbers, and underscore characters only, and
- is case-insensitive (for example, “MyData” and “mydata” are the same identifier).

SQL identifiers that begin with a dollar sign ($) are used to distinguish SQL system variables from other identifiers. These variables are described in the section "System variables" on page 11.

The following are some examples of valid SQL identifiers:

```
customers
i1
dept_total
$sqlcode
```

A variable of data type OBJNAME can take the place of an identifier (brand identifier, table name identifier, or column name identifier). To substitute a variable for an identifier, precede the variable name with a colon. For example, when this statement

```
select <column> from <table>
```

is used within a procedure where the column and table name are not known, it can be specified as

```
procedure sel(col, tab)
argument objname col, tab;
{
  select :col from :tab;
  printall;
} end procedure sel;
```

The following statement calls this procedure:

```
sel("city", "offices");
```

## Qualified identifiers

Some SQL entities are compound objects consisting of component parts. For example, a database is composed of tables, which are composed of individual columns. 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. Data-

base names and table names are separated by an exclamation point (!) while column, table names, and table owner are separated by a period (.) as in the following:

```
dbname!owner.table.column
```

For example, the qualified identifier

```
accounting!midwest.customers.rcvbl_amount
```

identifies a column named "rcvbl_amount" in the table named "customers," owned by the user "midwest," in the "accounting" database. Compound identifiers for each SQL object are described later in this chapter.

## Variables

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,
- has an associated SQL data type, and
- must be explicitly declared before it is used (the exception is CURSOR variables).

The DECLARE statement, described in the SQL Reference guide, is used to declare a SQL variable. When initially declared, a variable is uninitialized. Attempting to evaluate an uninitialized variable (for example, using it in an expression) causes a runtime error. A variable receives a value through a SQL assignment statement or by receiving a return value from a SQL procedure.

Only certain SQL data types can take on the special value NULL, which represents an unknown value. NULL values are completely legal in SQL expressions and do not produce runtime errors. The ability of SQL variables to hold NULL values eliminates the need for separate indicator variables, which are often found in SQL implementations.

Butler SQL supports three types of variables, each defining its own scope:

- **External variables** are declared outside of a SQL procedure definition in the outer block. The scope of an external variable begins with its declaration and extends through the outer block for the remainder of the SQL session or until its declaration is explicitly revoked. External variables are not visible within SQL procedures.
- **Local variables** are declared within a SQL procedure definition. The scope of a local variable begins with its declaration and ends at the end of the procedure definition or when its declaration is explicitly revoked. It is not visible inside other procedures or in the outer block.
- **System variables** are maintained by Butler SQL. The value of a system variable may be set as a result of SQL statement execution, or it may govern optional statement-execution features. System variable names always begin with a dollar sign ($). (This is not the case with ODBC variables, as they do not start with a dollar sign.) Their scope is global; they are visible throughout the outer block and within all SQL procedures. System variables are discussed in the following section.

## System variables

The system variables shown in Table 1 provide status information on the execution of SQL data-manipulation commands, specify limits and processing options for those commands, define date and time data-type formats, or provide current date and time information.

**Table 1: SystemVariables**

| Variable | Data type | Information |
|---|---|---|
| $colcnt | INTEGER | The number of columns in the rowset created by the most recent SELECT statement |
| $cursor | CURSOR | The default cursor, automatically set to identify the rowset created by the most recent SELECT statement |
| $datefmt | VARCHAR | Sets the date literal for input and output |
| $maxrows | INTEGER | The maximum number of rows to be retrieved by a SELECT statement |
| $rowcnt | INTEGER | The number of rows selected by the most recent SELECT statement (if known; i.e., if the EXTRACT update mode was used) |
| $sqlcode | INTEGER | The result code set by execution of each SQL data manipulation statement |
| $timefmt | VARCHAR | Sets the time literal for input and output |
| $tsfmt | VARCHAR | Sets the timestamp literal for input and output |
| $procid | INTEGER | The process ID of the SQL process executing in the environment. The value of $procid is set to NULL if there is no unique process associated with the current SQL session |
| $currentDate | DATE | The current system date of the Butler server Macintosh |
| $currentTime | TIME | The current system time of the Butler server |
| $currentTimeStamp | TIMESTAMP | The combination of the current system time and date of the Butler server Macintosh |

The **$maxrows** system variable controls detailed processing of the SELECT and/or FETCH verbs. When **$maxrows** is set to a non-zero value, the SELECT and FETCH statements will stop fetching data from the database once the number of rows specified by **$maxrows** is equaled.

Setting **$maxrows** to 24, for example, provides a way to return the first 24 rows of data resulting from the query without waiting for the entire query to complete.

**Note**

If the query specifies sorting, duplicate row elimination, or grouping, the DBMS will effectively perform the entire query before beginning to produce retrievable results.

## Literals

SQL literals represent constant values in a SQL program. Literal representations are provided for most of the SQL data types.

A boolean literal represents a BOOLEAN constant. There are two boolean literals, $TRUE and $FALSE (representing the TRUE and FALSE values, respectively).

An integer literal represents an INTEGER constant. It is specified as a sequence of digits with an optional leading plus or minus sign, for example:

```
1
-756
+812364734
```

A decimal literal represents a DECIMAL constant. It is specified as a sequence of digits with a decimal point and an optional leading plus or minus sign, for example:

```
1.1
-756.87
+812364734.0
.995
```

A money literal represents a MONEY constant. It has the same form as a decimal literal, with a leading dollar sign, for example:

```
$1.1
$-756.87
$+812364734.0
$.995
```

A floating-point literal represents a FLOAT constant. It is specified in scientific notation, for example:

```
1.234E10
-3.45E-2
+8.22222E+1
1e
```

A string literal represents a CHAR, VARCHAR, or LONGCHAR constant. It is specified as a sequence of printable characters enclosed in either single or double quotes, for example:

```
"Hello"
'Joe Smith'
"I'm convinced!"
```

A double quote or single quote character can be included in a string literal by using the opposite quote character as the delimiting quote, as shown in the last string literal of the previous example. The delimiting quote character itself can be included in the body of a string literal by specifying two quote characters in succession. For example, the string

```
"I'm convinced!", he said.
```

can be represented as a string literal enclosed in either single or double quotes by repeating the single or double quote characters, respectively, that are contained in the string. The single-quoted and double-quoted literals for the string are as follows:

```
"'I'm convinced!", he said."

""""I'm convinced!"", he said."
```

String literals can also represent DATE, TIME, or TIMESTAMP values using *mm/dd/yyyy* and *hh:mm:ss.ss* notation, as shown in the following examples:

```
"04/15/1987 13:57:03"
'04/01/1988'
"00:03:45.30"
```

**Note**

String literals representing LONGCHAR values must be 255 characters or less.

Hexadecimal literals are used to assign values to variables or columns of Butler's binary types (for example, VARBIN, DOCUMENT, and PICTURE). A hexadecimal literal is a string of valid hexadecimal digits (0-9, A-F, and a-f) surrounded by quotes and beginning with '$', '0X', or '0x'. (The first character of each of the last two items is a zero.)

You can also use ordinary text literals to assign binary values. In this case, the value stored is the binary representation of the text.

Here are examples showing assignment of values to VARBIN variables:

```
DECLARE VARBIN myBinVarA = '$41';
DECLARE VARBIN myBinVarB = '0X42';
DECLARE VARBIN myBinVarC = '0x43';

DECLARE VARCHAR hexString = '$ff0b45';
DECLARE VARBIN myBinData = hexString;

/* The next two statements assign the same value to the binText variable */

DECLARE VARBIN binText = 'A';
DECLARE VARBIN binText = '$41';
```

Typically, these literals are converted automatically to the appropriate type by their use in context. SQL also provides explicit data-type conversion operators, described in the section "Data-Type Conversion" later in this chapter.

A special literal, $NULL, is used to indicate that a SQL value is NULL. The $NULL literal can appear in assignment statements, for example:

```
integer x;
x = $NULL;
```

SQL statements and expressions can also contain the $NULL literal:

```
insert into mytable values ( 1, "ABC", $NULL );
```

In the strictest sense, $NULL is not a value, but an indication that a value is not known. In most cases, however, you can use the symbolic constant $NULL as a SQL literal in expressions.

A variable of data type VARCHAR can take the place of a string literal. To substitute a variable for a string literal, precede the variable name with a colon. For example, the following statement opens a database using a variable instead of hard coding the database name as a string literal:

```
varchar dbname = "daldemo"
open database :dbname
```

The colon may be omitted if the specified variable causes no ambiguity.

SQL data-type conversion operators, described in the section “Data-Type Conversion” later in this chapter, can be used to construct literals with SMINT, SMFLOAT, DATE, TIME, and TIMESTAMP data types from other literal types. There is no way to specify literal CURSOR, OBJNAME, or GENERIC data.

## Operators

Literals, variables, and other identifiers (such as column references) can be combined in expressions to calculate parameters that are used when statements are executed. An expression can be:

- a literal,
- an identifier,

- an operator applied to one or more expressions.

Butler SQL has unary, binary, and ternary operators that use prefix, infix, and postfix notation. The following sections describe the general-purpose operators that are available throughout the language. Specialized operators and functions that appear only in specific contexts are described with the appropriate statements.

## Numeric operators

Butler SQL supports the five arithmetic operators shown Table 2. These operators can be applied to any of the numeric data types.

**Table 2: Numeric Operators**

| Operator | Type | Notation |
|---|---|---|
| Addition | Infix binary | a + b |
| Subtraction | Infix binary | a - b |
| Multiplication | Infix binary | a * b |
| Division | Infix binary | a / b |
| Negation | Prefix unary | – a |

If any argument of an arithmetic operator is NULL, the result is NULL.

## String-concatenation operator

The infix string concatenation operator (+) can be applied to any string data type (CHAR, VARCHAR, or LONGCHAR), as in the following example:

```
"abc" + "xyz"
```

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.

**Note**

The string concatenation operator can appear in any SQL program structure statement; however, it cannot be used in SQL data-manipulation statements.

The string concatenation operator can also be used with Butler's binary data types.

Example:

```
DECLARE VARBIN myBinary1 = '$00001BF83FFC3FFE6F';
DECLARE VARBIN myBinary2 = '$6E7B6E7FFC3CF41EF8';
DECLARE VARBIN myBinary = myBinary1 + myBinary2;
```

## Comparison operators

SQL includes six comparison operators, shown in Table 3, that perform comparisons among data values to produce a BOOLEAN result.

**Table 3: Comparison Operators**

| Operator | Type | Notation |
|---|---|---|
| Equality | Infix binary | a = b or a == b |
| Inequality | Infix binary | a != b or a <> b |
| Less than | Infix binary | a < b |
| Greater than | Infix binary | a > b |
| Less than or equal | Infix binary | a <= b |
| Greater than or equal | Infix binary | a >= b |

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.

**Note**

All of the comparison operators shown in Table 3 may be used to compare binary values; however, the equality operator ('=' or '==') is probably the only useful one for these.

Example:

```
VARBIN myVarBinA = "$0A";
VARBIN myVarBinB = "$0B";
PRINT (myVarBinA = myVarBinB); /* prints FALSE */
```

## Logical operators

SQL supports three logical operators, shown in Table 4, that operate on BOOLEAN values to produce a BOOLEAN result.

Each logical operator produces a NULL result if any of its operands are NULL.

**Table 4: Logical Operators**

| Operator | Type | Description |
|---|---|---|
| AND | Infix binary | TRUE if both operands are TRUE |
| OR | Infix binary | TRUE if either operand is TRUE |
| NOT | Prefix unary | TRUE if operand is FALSE |

## 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, as shown in these examples:

```
quota is null
(sales / quota) is null
```

Since the other SQL operators usually produce a NULL result if any of their arguments are NULL, the IS NULL operator is frequently used in conditional tests, as in this example:

```
/* Combine first name, optional initial, and last name */
if (initial is null)
  name = first + " " + last ;
else
  name = first + " " + initial + ". " + last ;
print name ;
```

Because of the IS NULL operator, the name variable always receives a non-NULL value.

## Data-type conversion operators

Butler SQL supports automatic data-type conversion when data of different types are combined in an expression. For conversion purposes, data types are grouped into the hierarchy of type classes shown in Table 5.

**Table 5: Hierarchy of data type classes**

| Type class | Data types | Precedence |
|---|---|---|
| DECIMAL | DECIMAL, MONEY | Highest |
| FLOAT | FLOAT, SMFLOAT | |
| INTEGER | INTEGER, SMINT, BIGINT, TINYINT | |
| BOOLEAN | BOOLEAN | |
| TIMESTAMP | DATE, TIME, TIMESTAMP | |
| VARCHAR | VARCHAR, LONGCHAR, CHAR | |
| BINARY | VARBIN, PICTURE, SOUND, ICON, MOVIE, DOCUMENT | Lowest |

When an operator has operands of different data types within the same class, the operands are automatically converted to the appropriate type class before applying the operator. When operands from different type classes appear, the operand from the lower type class is automatically converted to the type of the higher class, if possible. For example, SQL will successfully evaluate the expression

```
"123" + 7 - 6.5
```

to produce a decimal value 123.5 by promoting the VARCHAR “123” to an INTEGER 123, adding, and promoting the result to DECIMAL type before subtracting. If the data cannot be converted, a runtime error occurs.

Butler SQL aggressively converts data types across an assignment regardless of the data type hierarchy. The program fragment

```
declare varchar x ;
set x = "123" + 7 - 6.5 ;
```

causes the DECIMAL result of the expression to be converted to the VARCHAR value "123.5", which is assigned to the variable *x*. If the data cannot be converted, a runtime error occurs.

For the rare cases in which the automatic data-type conversions do not produce the desired effect, Butler provides the data-type conversion operators shown in Table 6.

**Table 6: Data-type conversion operators**

| Operator | Description |
|---|---|
| BOOLEAN | Converts operand to a BOOLEAN value |
| CHAR | Converts operand to a CHAR value |
| DATE | Converts operand to a DATE value |
| DECIMAL | Converts operand to a DECIMAL value |
| FLOAT | Converts operand to a FLOAT value |
| INTEGER | Converts operand to an INTEGER value |
| MONEY | Converts operand to a MONEY value |
| SMFLOAT | Converts operand to a SMFLOAT value |
| SMINT | Converts operand to a SMINT value |
| TIME | Converts operand to a TIME value |
| TIMESTAMP | Converts operand to a TIMESTAMP value |
| VARCHAR | Converts operand to a VARCHAR value |
| VARBIN | Converts operand to a VARBIN value |
| LONGCHAR | Converts operand to a LONGCHAR value |
| PICTURE | Converts operand to a PICTURE value |

**Table 6: Data-type conversion operators**

| Operator | Description |
|---|---|
| SOUND | Converts operand to a SOUND value |
| ICON | Converts operand to an ICON value |
| MOVIE | Converts operand to a MOVIE value |
| DOCUMENT | Converts operand to a DOCUMENT value |
| BIGINT | Converts operand to a BIGINT value |
| TINYINT | Converts operand to a TINYINT value |

Each type conversion operator is a unary prefix operator whose operand can be any reasonable data type. For example, the following expression takes the integer value of the variable "i" and explicitly converts it to a character representation before concatenating it with another string:

```
("column #" + varchar i)
```

## Operator precedence

In an expression involving multiple operators, SQL applies the operator precedence hierarchy shown in Table 7.

Operators group from left to right within the same precedence level. For example, the expression

```
a - b + c
```

is interpreted as

```
(a - b) + c
```

You can use parentheses in the usual way to alter the standard precedence rules and to improve readability:

```
((a + b) * c < d) or (today = "10/01/1987"))
```

**Table 7: Operator precedence**

| Operator class | Precedence |
|---|---|
| Unary minus | Highest |
| Type conversion | |
| Multiply, divide | |
| Add, subtract | |
| Comparison operators | |
| NOT | |
| AND | |
| OR | Lowest |

## Null Processing

When a column or variable is said to be null, it means that it has no known value. This is different from an empty string which, by definition, is a known value.

Butler SQL currently treats NULLs and empty strings as the same because it has no mechanism to distinguish between the two. Although this limitation will be changed in a future version, in the mean time it is necessary to understand how a NULL column or variable can and should be used.

Let us say that we have two variables x & y that are said to be NULL, the question is does x=y. The answer is NO. Since a NULL column/variable is unknown in content then we don't know the value of either, so we don't know if they are equal. Also when you perform operations and comparisons with NULLs the result is NULL also. There is specific operator to test for NULLs. It should also be pointed out that char, varchar and date

datatypes are the only datatypes that can be NULLs. Numeric values will be set at zero instead of null (this may change in a future version of Butler SQL).

NULL values may be combined with other values in expressions. If any of the operands of an operator have a NULL value, the result is NULL. Thus, if the variable x has a NULL value, each of the following expressions produces a NULL value:

```
declare x=$null,y=$null;
x+3
x * 3x || "Hello"
x > y
(x = y
-x
```

The NULL test can be used to detect NULLs and circumvent this behavior when necessary. For example, this procedure concatenates two strings and explicitly handles NULL values:

```
procedure con(x,y) returns varchar;
argument varchar x, y;
 {
   varchar result = x + y;
   IF (result is null)
     { if ( x is null) result = y;
       else result = x;}
   return result;
   } end procedure con;
```

There is also a limitation here. Since the NULL test ordered with another test will yield a NULL, you can get some unexpected behaviour.

Take the following example;

```
 if (x is null or x = 0)
    print "true";
 else
    print "false";
```

One would expect that if x is 0 or NULL, this fragment would print true; however, due to the limitation mentioned above, it does not. To get this fragment to work correctly, you must write it as follows:

```
if (x is null)
   print "true";
else if  (x = 0)
   print "true";
 else
   print "false";
```

# References

## Column references

Columns are identified in SQL statements by a column reference *(colref)*, with the following syntax:

[ *tblqual*. ] *colname*

The components of the column reference are as follows:

| | |
|---|---|
| *tblqual* | A table reference or table alias identifying the table containing the column. If *tblqual* is omitted, SQL resolves the column reference among the candidate tables. |
| *colname* | The name of the column. |

A column reference appearing in a clause of a SQL statement can identify a column of one of the tables named in the FROM (or INTO) clause of that statement; or, in the case of a subquery, the FROM clauses of the SELECT statement containing the column. SQL binds the column reference to a specific column of a specific table as follows:

- If a period appears in the column reference and a leading series of one or more period-separated tokens exactly matches a table specification in the FROM clause(s), then the specified table becomes the only candidate table for the column reference. If the column reference is in a subquery, the search for a table specification match begins with the innermost FROM clause and proceeds sequentially through each table specification in the FROM clause before moving to the next

FROM clause, and so forth.

- If a period does not appear in the column reference or if the leading part of the reference cannot be interpreted as a table reference from the context, then all tables specified in the FROM clause(s) are candidate tables.

In a relational database, the absence of column hierarchies means that a column reference can always be unambiguously interpreted as its table reference and column name components. For example, the reference "a.b.c" can only be a reference to column "c" in table "a.b." In a database with hierarchical columns, the reference might also be to a column "b.c" in table "a."

For most cases, the FROM clauses will restrict the possible interpretations of the column reference to a single one. If both table "a" and table "a.b" appear in the FROM clause, SQL will bind the reference to the first table appearing in the FROM clause. The use of table aliases, described in the next section, is recommended heavily to avoid ambiguity.

## Table references

Tables and views within a database are identified in SQL statements by a table reference *(tblref)* with the following syntax:

[ *dbalias*! ] [ *tblgrp*. ] . . . *tblname*

The components of the table reference are as follows:

| | |
|---|---|
| *dbalias* | The alias of the database that contains the table, established by the OPEN DATABASE statement. If *dbalias* is omitted, the default database is assumed. |
| *tblgrp* | The table group containing the table. In some dbms's, the table group is used to identify the qwner of the table. If *tblgrp* is omitted, then the default table group is assumed. |

*tblname* The name of the table or view.

**Note**

Butler does not support table groups. Table references may contain group qualifiers, but they are ignored.

In the simplest case, a table in the default database can be referenced by its name. A reference to a table in another database requires a *dbalias* qualifier.

The following examples show valid table references that assume an ANSI SQL database organization, where user names form a single-level table group structure:

| | |
|---|---|
| `offices` | My "offices" table in the default database |
| `guido.orders` | Guido's "orders" table in the default database |
| `mfg!ethel.staff` | Ethel's "staff" table in the "mfg" database |
| `mfg!offices` | My "offices" table in the "mfg" database |

## Aliases

Fully qualified references can be long and unwieldy to use, especially in a complex database structure. To simplify these references, an alias, or alternate name, can be assigned. An alias is a SQL identifier that is used in subsequent references to identify a SQL object in place of its real name. Butler SQL supports several different kinds of aliases:

- A database alias is declared when a database is opened (with the OPEN DATABASE statement) and is used in subsequent references to that database while it remains open.
- A table alias is declared in the FROM clause of a SELECT statement and is used in subsequent references to that table within the SELECT statement. The alias is discarded when execution of the SELECT statement is complete.

- A column alias is declared in the select list of a SELECT statement. The alias becomes the name of the resulting column in the rowset created by the SELECT statement; it is discarded when the rowset is deselected.

In some cases, aliases are not just a convenience; they are required to create an unambiguous reference, as in the following instances:

- When joining a table to itself, an alias must be used to distinguish columns from the two instances of the table.
- When two different databases have identical names, an alias must be used for one of the databases if both are to be open simultaneously.

In fact, databases, tables, and columns are always referenced by alias names. If an alias is not explicitly declared, Butler SQL uses the name of the database, column, or table as its alias. In the SQL statement syntax, *dbalias*, *tblalias*, and *colalias* refer to a declared alias name or to the actual name of the database, table, or column if no alias has been explicitly declared.

**Note**

SQL aliases are different from Macintosh file aliases. Butler SQL also supports file aliases to Butler database files. See the "Butler SQL User's Guide" for more information.

## Rowsets

Butler SQL supports set-level access to data in a host database through the concept of a rowset. A rowset is a collection of rows and columns from a database that has the same characteristics as a table. All SELECT and DESCRIBE statements output data in rowsets.

Conceptually, a rowset is formed by

- selecting rows from one or more tables from a database (with a FROM clause of a SELECT statement or with the table reference of a DELETE, INSERT, or UPDATE statement);
- forming the Cartesian product of the chosen tables (a table containing all combinations of one row from each of the chosen tables) and possibly adding calculated columns whose values are calculated from the contents of other columns;
- selecting those rows that meet a specified search criterion (with a WHERE clause of a SELECT statement);
- optionally grouping the resulting rows into groups whose member rows share the same value for one or more columns (with a GROUP BY clause of a SELECT statement);
- optionally selecting only those row groups that meet a specified search criterion, discarding the other row groups (with a HAVING clause of a SELECT statement);
- if grouping is requested, replacing each row group with a single summary row containing summary data for the group (with aggregate functions);
- eliminating all columns except those explicitly requested;
- optionally eliminating identical duplicate rows (with a DISTINCT clause of a SELECT statement).

Typically the underlying DBMS performs the work of forming a rowset. In some cases, a rowset is formed for the duration of a single SQL statement and is then destroyed. For example, the SQL statement

```
update staff set salary = salary * 1.1 where dept = "Sales" ;
```

conceptually forms a rowset consisting of rows from the "staff" table that represent employees in the "Sales" department, updates all rows in the rowset, and then destroys the rowset.

In other cases, rowsets persist and are manipulated by a SQL program for row-by-row processing. For example, in the following SQL fragment, the SELECT statement forms a rowset, and each row of the rowset is processed individually by the FOR EACH statement:

```
select * from staff where dept = "Sales" ;
for each {
  print name, salary ;
  if (jobcode="Mgr")
    mgr_total = mgr_total + salary ;
  else
    nonmgr_total = nonmgr_total + salary ;
}
print mgr_total, nonmgr_total ;
```

## Cursors

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.

The SELECT statement stores automatically the cursor that it creates in the SQL system variable **$cursor.** This cursor is used implicitly by subsequent data-manipulation statements. As a result, SQL programs that process a single rowset at a time do not need explicit cursor references. For SQL programs that process several rowsets concurrently, a cursor can be stored in a SQL variable that has the CURSOR data type. Subsequent data-manipulation statements reference the CURSOR variable to indicate the rowset to be processed.

When a cursor is first created by the SELECT statement, there is no current row; the cursor points just before the first row of the rowset. The FETCH statement moves the current row forward and backward through the rowset or to a specific row within the rowset. If the FETCH statement positions the cursor past the last row of the rowset, there is once again no current row. Finally, the DESELECT statement destroys a cursor and deactivates the associated rowset.

## Cursor-based references

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 holding the cursor. A cursor-based row reference identifies the current row as a unit and has the form

CURRENT OF *cursor*

A cursor-based column reference identifies a particular column of the current row and has one of the following three forms:

[ [ *cursor* ] —> ] *colref*
[*cursor*] —> *colnr*
[*cursor*] —> [:]*varname*

The first form identifies the column by name. If the cursor part of the reference is omitted, the program assumes the default cursor (stored in the system CURSOR variable **$cursor**). This shorthand can be used in SQL assignments and program control statements without ambiguity. Nevertheless, the cursor qualification must be included in a query specification, because SQL interprets unqualified references in query specifications as non-cursor-qualified column references.

The second and third form of cursor-qualified column references identify the column by its ordinal number. The column number is specified literally as *colnr* or is taken from the value of the INTEGER variable *varname.* The *varname* can be a VARCHAR or OBJNAME. The PRINT statement in the following example illustrates the three types of references, all identifying the same column:

```
declare integer i = 2 ;
select last_name, sales_ytd, quota_ytd from staff into x ;
fetch next x ;
print x—>sales_ytd, x—>2, x—>:i, sales_ytd ;
```

# Chapter 2 • Data types

This chapter describes the data types available for Butler SQL columns and variables.

## SQL Data types

Butler SQL provides a set of standard data types that represent data commonly found in data processing files and databases. SQL automatically maps the native data types supported by a host system into these standard SQL types when it manipulates host data sources. The standard SQL types are also used to declare variables within a SQL program. The standard SQL data types are shown in Table 8.

**Table 8: SQL data types**

| Data type | Type code | Description |
|---|---|---|
| BOOLEAN | 1 | A logical value that can assume the values TRUE, FALSE, or NULL (unknown) |
| SMINT | 2 | A signed 16-bit integer |
| INTEGER | 3 | A signed 32-bit integer |
| SMFLOAT | 4 | A signed 32-bit floating-point number |
| FLOAT | 5 | A signed 64-bit floating-point number |
| DATE | 6 | A date consisting of a year, month, and day |
| TIME | 7 | A time consisting of an hour (0–23), minute, second, and hundredth of a second |

**Table 8: SQL data types**

| Data type | Type code | Description |
|---|---|---|
| TIMESTAMP | 8 | A date-and-time stamp, with the combined components of the DATE and TIME types |
| CHAR | 9 | A fixed-length character string, with a constant length (number of characters) |
| DECIMAL | 10 | A signed decimal number that has an associated precision (total number of decimal digits) and scale (number of digits to the right of the decimal point) |
| MONEY | 11 | A data type with the same characteristics as DECIMAL, interpreted as a currency amount |
| VARCHAR | 12 | A variable-length character string, with an associated length (current number of characters) |
| VARBIN | 13 | A variable-length byte string, with an associated length (current number of bytes) |
| LONGCHAR | 14 | A variable-length character string, with an associated length (current number of characters) up to 2.3G |
| ❖ See “Butler’s extended data types” on page 37 for information on data types 13–28. | | |
| PICTURE | 24 | Identical to VARBIN, except data is expected to be in the Macintosh ‘PICT’ (picture) resource format. |
| SOUND | 25 | Identical to VARBIN, except data is expected to be in the Macintosh ‘snd ’ (sound) resource format |
| ICON | 26 | Identical to VARBIN, except data is expected to be in the Macintosh ‘cicn’ (color icon) resource format |
| MOVIE | 27 | Identical to VARBIN, except data is expected to be in the Macintosh ‘moov’ (QuickTime movie information) resource format |
| DOCUMENT | 28 | Identical to VARBIN, except that the data stored is a Macintosh file or alias in binary format. |

**Table 8: SQL data types**

| Data type | Type code | Description |
|---|---|---|
| CURSOR | none | A value identifying an active SQL rowset |
| OBJNAME | none | A data item whose value identifies a SQL identifier |
| GENERIC | none | An item used to declare a SQL variable that can assume any of the other data types when data is assigned to it |

The first 12 data types (BOOLEAN through VARCHAR) are the atomic data types that SQL uses to represent data from a host database. They can also be used to declare SQL variables for use in SQL programs.

The CURSOR data type is used in SQL variable declarations only. A variable of CURSOR type holds a value that uniquely identifies a SQL rowset—a tabular collection of row and column data created by the SQL SELECT statement or one of the SQL DESCRIBE statements. The CURSOR variable can be used to identify the rowset as a whole (for example, in the PRINTALL statement, which outputs the contents of a rowset). It can also be used in cursor-qualified references to identify specific rows or columns of the rowset. (See “Cursors” on page 31.)

The OBJNAME data type is used to declare a SQL variable whose value is to be interpreted as an identifier. The value of an OBJNAME variable is a variable-length character string and is similar to the value of a VARCHAR variable. The OBJNAME variable is also assigned a value in the same manner as a VARCHAR variable: by receiving the value of a SQL string literal, a CHAR or VARCHAR variable, or a string expression involving some combination of these. When an OBJNAME variable is referenced in a SQL expression or statement, however, its value (a string) is used as an identifier. Thus, you can reference an OBJNAME variable in places that normally require a SQL identifier (such as a table name or database name) to provide more runtime flexibility in the language.

**Note**

OBJNAME variables used in database, table, and column references must

constitute the *entire* reference. OBJNAME variables used as parts of a reference will generate an error. For example, the example shown below would work correctly with Butler SQL, because the entire reference is an OBJNAME variable.

---

In the following example, an OBJNAME variable is used to construct a table reference. Note that the table referenced by the SELECT statement varies, depending on the value of the OBJNAME variable:

```
/* Declare and initialize variables */
declare objname mytblref ;
declare varchar mydb = "accounting" ;
declare varchar mytbl = "customers" ;

. . . other processing . . .

/* Assemble fully qualified table reference (db!table) */
mytblref = mydb + "!" + mytbl ;

. . . other processing . . .

/* Retrieve all data from designated table and output it */
select * from mytblref ;
printall ;
```

The GENERIC data type is used to declare a SQL variable whose data type is malleable. The data type of a GENERIC variable dynamically adjusts to the type of data that it holds. If you assign an integer value to a GENERIC variable, its data type becomes INTEGER. If you later assign a character string value to the same variable, its data type becomes VARCHAR. The data type of a GENERIC variable is automatically changed when the variable appears on the left side of a SQL assignment statement, as shown in the following example:

```
declare generic myvbl ;
declare integer myint = 17 ;
declare varchar mystr = "Hello" ;

myvbl = myint ;
print $typeof(myvbl), myvbl ; /* reports an integer of value
  17 */

myvbl = mystr ;
```

```
print $typeof(myvbl), myvbl ; /* reports a string of value
  "Hello" */
```

The GENERIC data type is especially useful for receiving query results from a host database when the data type of those results is not known in advance, as shown in the following example:

```
declare generic myvbl1, myvbl2 ;

/* Select data from database & fetch it */
select city, sales/quota from offices
  where office_nr = 101
fetch ;

/* Assign columns to generic variables */
myvbl1 = —> 1;
myvbl2 = —> 2;
```

At the end of this example, "myvbl1" has either a VARCHAR or CHAR data type, and "myvbl2" has some numeric data type, probably DECIMAL or a floating-point data type, depending upon how the particular DBMS brand handles the division operator.

On a given client system presenting a SQL API, each standard SQL data type has a defined physical representation. The physical representation conforms to the corresponding native data type of the client system wherever possible.

## Butler's extended data types

Data types 13 through 28 can be used both for storing data in columns of a table and for variables in SQL programs. (See "SQL data types" on page 33.)

The PICTURE, SOUND, ICON, and MOVIE types are table stored internally by Butler in the same way as the VARBIN type. No data format checking is done. You could, for example, store picture data in a SOUND column and Butler SQL would not complain.

The purpose of the different types is to provide information to generic client applications about the kind of binary information stored in a column or variable. With this information, the data can be presented to the user in an appropriate way (displaying a picture, playing a sound, etc.). If you are writing an application whose database will be accessed only by custom front-ends, you may find it preferable to use the VARBIN data type for such data instead. This would, for example, make it possible to store in one column a different type of data in each row of a table.

The DOCUMENT type is a special one, used for storing binary representations of Macintosh files and file aliases. See "Document Data Types" on page 45 for more information on this data type.

## Using the new data types in applications

Butler SQL improves upon DAL/DAM-compatible servers in its support of data items larger than 32K and its special multimedia types. This means that most currently available DAL/DAM client applications are unable to deal with Butler SQL databases including such types. When encountering an unknown data type, some applications will politely inform you that an unknown data type was encountered, while others will simply crash.

Client application developers are encouraged to allow queries of databases including our new types, even if they don't support retrieval of the items. We hope that some will go further by taking full advantage of Butler's capabilities—allowing retrieval of text items greater than 32K, viewing of pictures, playing of sounds, etc. Appendix A • Butler SQL DAM specifics describes how to use these types in your C or Pascal based application.

## DAM data type codes

The four-character Data Access Manager codes for Butler's new types are shown in Table 16 "Appendix A • Butler SQL DAM specifics" on page 173. These are needed only by those using low-level DAM functions.

## Date, time, and timestamp formats

By modifying **$datefmt**, **$month**, and **$timefmt** system variables, you can change how date, time, and timestamp literals are input and displayed.

The date literal format is set with **$datefmt**. Entries are not case sensitive (YY is the same as yy). The default date literal format is:

```
$datefmt = "MM/DD/YYYY";
```

where

| | | |
|---|---|---|
| YY | 2-digit year | 87, 01 |
| MMM | $month string | Jan, Feb |
| YYYY | 4-digit year | 1987, 2001 |
| MM | 2-digit month | 05, 12 |
| M | 1- or 2-digit month | 5, 12 |
| DD | 2-digit day | 04, 23 |
| D | 1- or 2-digit day | 4, 23 |
| DDD | 3-digit Julian day | 001, 020, 345 |

**Note**

Some DAL/DAM client applications do not handle DATE data types. In order to see the results of changes made to the $datefmt system variable, you may change all date results to text by issuing the PRINTCTL 0 statement.

For example:

```
$datefmt = "DD/MM/YY"
```

would output a date in this format:

```
13/06/55
```

The **$month** variable allows redefinition of the text used for a month when a string, MMM, is specified in **$datefmt.** The default at startup is

```
$month = "Jan+Feb+Mar+Apr+May+Jun+Jul+Aug+Sep+Oct+Nov+Dec";
```

The + is the delimiter between the 12 text values.

The time literal is set with **$timefmt.** Entries are not case sensitive (HH is the same as hh, except for the case of the AM/PM indicator). If the AM/PM indicator appears anywhere in the string, the normal 24-hour "hour" field is converted to 12-hour representation. The default date format literal is:

```
$timefmt = "HH:MM:SS.hu";
```

where

| | | |
|---|---|---|
| HH | 2-digit hour | 03, 12, 23 |
| H | 1- or 2-digit hour | 7, 10, 20 |
| MM | 2-digit minutes | 05, 10, 55 |
| M | 1- or 2-digit minute | 5, 10, 55 |
| SS | 2-digit seconds | 06, 11, 56 |
| S | 1- or 2-digit seconds | 6, 11, 56 |
| hu | 2-digit hundredths | 03, 32, 78 |
| T | 1-digit tenth | 3, 7, 9 |
| A.M.\|P.M.\|X.M. | AM/PM mark | A.M., P.M. |
| AM\|PM\|XM | AM/PM mark | AM, PM |
| a.m.\|p.m.\|x.m. | AM/PM mark | a.m., p.m. |
| am\|pm\|xm | AM/PM mark | am/pm |

For example:

```
$timefmt = "HH:MM:SS"
```

would output the time in this format:

```
23:30:59
```

## Date Formats

Many customers have commented on the formatting differences between Butler SQL and ButlerClient or have had difficulties with date formatting for international formats. Setting the **$datefmt** in Butler SQL and casting a data value to a VARCHAR on the server may yield a significantly different date output than retrieving the raw binary value and having your client application convert the value to its textual representation.

In general, you should use your database to store data in a client-independent format (e.g. all dates stored in mm/dd/yyyy format). Your client applications are responsible for the changing dates into a specific format.

### DAL dates

Inside Macintosh VI, Chapter 8, “The Data Access Manager”, describes the representation of dates in DAL: two bytes for year followed by one byte representing the month and one byte representing the day.

When binary dates are sent or retrieved, the data type will be 'date'. As discussed below, Butler SQL is capable of type casting and returning dates in a VARCHAR format.

Confusion can arise if developers are unclear about where the conversion of a binary date to a textual representation occurs: on the Butler server before the value is returned to the client, or at the client by the client application which receives the raw binary value.

## ODBC date literals

The biggest problem when dealing with dates, times and timestamps is maintaining a consistent format throughout the clients and servers. Date literals in ODBC are maintained through the vendor expression syntax. Passing date literals as text with a consistent date format of yyyy-mm-dd allows different servers to deal with the same SQL in a uniform manner. Even if the standard date format is mm/dd/yy for your server, if you use the ODBC vendor expression date literal format, it will always use a format of yyyy-mm-dd.

Example:

```
Update Employee
Set Birthday =
--(* vendor(Microsoft), product(ODBC) d '1967-01-15' *)--
where name = 'Smith, John'
Update Employee
Set Birthday =
{ d '1967-01-15'}
where name = 'Smith, John'
```

## Butler SQL date formats

Butler sets a date format of MM/DD/YYYY by default, regardless of the date format set in the Macintosh operating system (Date and Time control Panel). The server administrator may choose to change the default date format (so that each client doesn't have to change it independently). To change the default, add $datefmt = "DD/MM/YYYY" (or whatever you want the default to be) to the msad$procedures file and place it in a folder named "Procedures" inside the Butler Preferences Folder on the server. (See "Stored procedures for performance" on page 170 for more information on using stored procedure files.)

Date literals in Butler must be enclosed in quotes, otherwise the month, day, and year delimiters may be interpreted as division operators. Client applications sending dates in binary form do not need to enclose the item in quotes because the binary representation is interpreted correctly by Butler.

When a DATE variable “myDate” is assigned a value, it is assumed that the day and month values are in the same ordering as the current setting of $datefmt. That is, myDate = '10/2/1992' corresponds to October 2, 1992 if $datefmt = 'MM/DD/YYYY' or February 10, 1992 if the $datefmt = 'DD/MM/YYYY'.

Since Butler SQL is capable of handling any date in the range ±32000 BC/AD the number of digits in the year is very important. If a user inputs a two-digit value, assuming the year is in the current century, and the current $datefmt specifies a four-digit year, the date will be interpreted as 0092 AD.

Finally, output of date values can take many forms. A $format function or PRINTF statement converts the binary representation of dates on the server and returns to the client application a VARCHAR representing the date based on the current setting of $datefmt. The $datefmt is also used to convert the date on the server before returning the character representation if you have, for example, used "PRINTCTL 0;" or "PRINTCTL $date = $char;" before printing the date or when you explicitly cast the date to a character type (e.g.: "PRINT VARCHAR myDate;").

## Float formats

Some differences exist between different SQL servers which are not properly addressed by the SQL syntax. Specifically, there is no method to specify the floating point format, default scale or rounding behavior of servers.

Butler SQL implements SMFLOAT, FLOAT and DECIMAL types as Extended values and uses the Macintosh built-in software library to convert precision floating point values to the Apple SANE representation when arithmetic expressions are evaluated. When floating point values are stored into the Butler SQL database file, they are represented by single, double and extended format floating point values, as documented in Apple's SANE format, which is compliant with the IEEE standards on floating point representation.

Butler SQL will convert automatically factors in a numeric evaluation to the appropriate data type. Evaluations including floating point expressions are always evaluated in 10-byte (80-bit) extended format. Butler does not require that an FPU be installed; however, a math co-processor may be used if it is present.

Normally, floating point evaluations have results that round up from zero. However, when a floating point value is cast to an integer, Butler SQL truncates the resulting value rather than rounding it up to the precision of the stored variable or column.

## Floating point vs. ASCII formats

By default, Butler's floating point format is a scientific notation representation of the floating point values with eight decimal place precision. This differs from other SQL servers which may use four or five decimal places by default and always format output in fixed notation.

Butler will allow floating point constants to be assigned or used within expressions. That is, both 123.456 and 1.23456E2 are allowed as values in the assignment to variables or as factors in an arithmetic expression.

Butler will convert and print a floating point value in fixed notation if the user calls either the $format or PRINTF statements with a fixed notation format character (%d,%i,%p). Otherwise, printing and typecasting of floating point values will use scientific notation format. The default behavior for a client can modified by changing the state of the “floatFmt” runtime setting.

### Floating point canonical representations

Changing the appearance of the fixed and scientific representations is accomplished by adjusting the client’s canonical representation (picture format) of floating point values.

Fixed numeric format:

```
$SetRuntime(floatFmt, "##,###,##0.000")
```

Scientific notation format:

```
$SetRuntime(floatFmt, “±#.000e±000”)
```

# - nonzero leading significant digit
0 - a significant digit, including zero
“.” - decimal place
“,” - thousands separator

**Note**

The decimal and thousands separators should be indicated per proper separator based on the server’s active system script.

### Default form of floating point

Normally, clients use scientific notation for floating point data. You can change the default to either scientific or fixed decimal notation. Use the $SetRuntime(printfForm, <boolean>); statement to change the default from scientific notation ($true) to fixed decimal notation ($false);

### Default scale of floating point values

For display, the conversion utilities in Butler allow the user to define the default number of decimal places.

“$SetRuntime(floatScale, 4);” sets the floating point precision for display to 4, for example.

## Document Data Types

**Note**

Some of the information in this section—particularly in the “Document retrieval” and “Document storage” sections—is applicable mainly to those writing client applications in low-level programming languages

such as C or Pascal. If you are not one of these people, you can safely ignore the information referring to the SQL and DAM API functions.

This section describes Butler's DOCUMENT data type. Unlike most data types, a DOCUMENT column or variable can hold two different types of data: a representation of the document in binary format, or an alias to the document in the format of an alias handle. Either type of data can be stored in a database, and the type can vary from row to row in a table.

## Document functions and variables

Butler SQL includes several SQL system functions and one system variable have been added for manipulation of DOCUMENT values. Examples of how to use each of these are included at the end of this section.

$LoadFileNamed(stringexp)

Takes as a parameter a string expression that evaluates to the path of a file in the server's file system. It returns a DOCUMENT value which is the specified file in binary format.

$LoadFileAliased()

Takes as a parameter a DOCUMENT variable or column. If the parameter is an alias, a DOCUMENT value which is the original file, in binary format, is returned. If the input parameter is a document, it is returned unmodified.

$MakeAlias()

Takes as a parameter a string expression that evaluates to the path of a file in the server's file system. It returns a DOCUMENT value which is an alias of the specified file.

$ResolveAlias()

Takes as a parameter a DOCUMENT variable or column containing an alias. It returns a string value which is the path of the original file.

$SaveFile()

Takes a parameters a DOCUMENT variable or column and, optionally, a string expression that evaluates to a path in the server's file system.This function saves the document data to the server and returns a string which is the path of the saved file. If no file path is specified in the second parameter, the file is saved in the directory containing the Butler application.

$AutoResolveAlias

A BOOLEAN system variable which controls the resolution of aliases. Initially, **$AutoResolveAlias** is TRUE, meaning that when DOCUMENT aliases are printed, or saved to the server using **$SaveFile()**, the alias is resolved and the original document is transmitted to the client or saved. If **$AutoResolveAlias** is set to FALSE, DOCUMENT alias values are printed and saved as Finder alias files.

## Retrieving documents

Getting document data to a client from the server is achieved simply by printing a DOCUMENT column or variable. When a DOCUMENT item is retrieved, a data item of type 'docu' is queued on the server, which can then retrieve.

What is somewhat irregular about the retrieval of a DOCUMENT item is the data returned to the client application is not the document data, but a Pascal string with the document's file name. The document itself is saved automatically on the client machine by the Butler database extension.

As mentioned, the **$AutoResolveAlias** system variable can be used to control alias resolution. If the variable is set to FALSE and a DOCUMENT item containing an alias is printed, the disk file saved to the client machine upon retieval will be a Finder alias file instead of the original document.

The internal format used for storing document data includes the document's file name. Therefore, when a document is saved on the client machine, no file name needs to be specified. The file is saved with the original file name in the default directory set by the client application.

If a file already exists with the original file name in the specified location, it will be overwritten. If there is a danger of duplicate file names, then, after the retieval, use the file name returned (and the knowledge that it is in the default directory) to rename or move it.

## Document storage

If the file to be stored resides on the server machine, the system functions described previously are used. If the document is to come from the client, the regular API function, DBSendItem, is used. The buffer passed to DBSendItem points to a Pascal string that specifies the full path to the document to be stored.

## Examples

The following SQL fragments (in this case, DAM examples) illustrate the use of the DOCUMENT data type and the system functions and variables described above.

```
/* Create an alias for a file on the server*/
DOCUMENT myDoc, myAlias;
myAlias = $MakeAlias(“IIsi:Desktop Folder:Pencil Test”);

/* Prove that we got it right; it should return the file path above */
PRINT $ResolveAlias(myAlias);

/* Load another file into a local variable */
myDoc = $LoadFileNamed(“IIsi:Desktop Folder:Pencil Test”);

/*Save that variable into another file on the server.
We are specifying a complete destination path here
The saved file name will be returned */
PRINT $SaveFile(myDoc, “IIsi:Desktop Folder:Pencil Copy”);

/* Try saving it again, this time in the default
directory. The saved file name will be returned */
PRINT $SaveFile(myDoc);

/* Try returning a file to client */
myAlias = $MakeAlias("IIsi:Desktop Folder:Pencil Test");
myDoc = $LoadFileAliased(myAlias);
```

```
/* …create the document on the client machine. The
file name will be returned from DBGetItem */
PRINT myDoc;

/* …create an alias file on client machine. The alias
name will be returned from DBGetItem */
$autoResolveAliases = 0; print myAlias;

/* …create the document on the client machine.
The file name will be returned by DBGetItem */ $autoResolveAliases = 1;
PRINT myAlias;
```

## Memory considerations

To use document data types with Butler SQL, the complete document must be able to fit into available RAM on the server. However, on the client side, when document data is being received from or sent to the server, the file transmission is sent in blocks to fit limited available RAM.

## Document examples

The following SQL fragment (in this case, DAM examples) illustrate the use of the Document data type and the system functions and variables described above.

```
Document myDoc, myAlias;
/* Create an alias for a file on the server*/
myAlias = $MakeAlias("myHD:Desktop Folder:Pencil Test");

/* Prove that we got it right; it should return the file path above */
print $ResolveAlias(myAlias);

/* Suck up another file into a local variable */
myDoc = $LoadFileNamed("myHD:Desktop Folder:Pencil Test");

/* Save that variable into another file on the server */
/* We are specifying a complete destination path here */
/* The saved file name will be returned */
print $SaveFile(myDoc, "myHD:Desktop Folder:Pencil Copy");

/* Try saving it again, this time in the default directory */
/* The saved file name will be returned */
```

```
print $SaveFile(myDoc);

/* Try returning a file to client */
myAlias = $MakeAlias("myHD:Desktop Folder:Pencil Test");
myDoc = $LoadFileAliased(myAlias);

/* …create the document on the client machine */
/* the file name will be returned from DBGetItem */
print myDoc;

/* …create an alias file on client machine */
/* the alias name will be returned from DBGetItem */
$autoresolvealiases = 0;

print myAlias;

/* create the document on the client machine */
/* the file name will be returned from DBGetItem */
$autoresolvealiases = 1;
print myAlias;
```

# Text Collation

This section describes text collation sequencing and the use of the Macintosh operating system's International Sorting capabilities to provide localization of a Butler database/application for foreign languages. This affects CHAR, VARCHAR and LONGCHAR data types.

There are several statements in Butler which involve the comparison of text variables, columns or string literals. Further, the evaluation of comparison operators inside a query specification, by default, operate in a different manner to other conditional statements.

## Comparison, sorting and WHERE conditions

There are three types of text comparisons performed in Butler: during condition comparisons for SQL statements other than a query specification, when ordering a rowset and during the execution of the WHERE or HAVING conditions of a query specification.

The first is comparisons within SQL condition statements like the IF, WHILE, CASE statements. For example:

```
if (“A” <> “a”)
  print "not equal";
else
  print "equal";
```

The second type of text comparison is performed while ordering a rowset that has been generated as the result of a SELECT statement. Butler defaults to using the international sorting sequence and case sensitivity. It is important to note that the sort collation and secOrdering flags apply to ORDER BY, GROUP BY and DISTINCT processing throughout our cursor/rowset generation code.

The third type of text comparison is performed during selection of rows for the rowset while a SELECT statement is being composed. That is, WHERE and HAVING condition comparisons of text columns, variables and/or literals. Conditions in this group default to case insensitive such that “ROSE” and “rose” compare as equivalent values. Use the “whereSecOrdering” to indicate whether these comparisons are case sensitive or not. Use the “cmpCollation” to change the default collation sequencing from ASCII to International for this type of text comparison.

## Collation sequences

Butler can use the built-in set of international collation facilities to determine ordering of character values using the script system selected. This allows Butler to collate properly when using non-ascii based script systems such as Kanji or Chinese. It also allows proper sorting and comparison of Roman script systems that contain diacritical marks, such as French and German.

By default, Butler uses the standard ASCII collation methods. This offers the best performance. For compatibility with international script systems, you can choose to use the collation facilities provided by the Macintosh operating systems.

| | |
|---|---|
| $SetRuntime(cmpCollation, 0); | - sets collation sequence to ASCII [the default] |
| $SetRuntime(cmpCollation, 1); | - sets collation sequence to use internationalized system routines |

If the data being sorted or compared is larger than 63 characters, Butler will always use the international comparison routines.

**Note**

Normally, North American users do not have to worry about this switch since the International Sorting collation is basically the same as the ASCII collation sequence.

By default, Butler will use a case and diacritical mark sensitive comparison for condition statements, query specifications, and ordering selection rowsets (for ORDER BY, GROUP BY and DISTINCT processing). To change these settings, use the following routine:

| | |
|---|---|
| $SetRuntime(cmpSecOrdering, $true); | - for case sensitive comparisons |
| $SetRuntime(cmpSecOrdering, $false); | - for case insensitive comparisons |
| $SetRuntime(whereSecOrdering, $false); | - for case insensitive query specifications |
| $SetRuntime(whereSecOrdering, $true); | - for case sensitive query specifications |
| $SetRuntime(sortSecOrdering, $true); | - for case sensitive rowset ordering |
| $SetRuntime(sortSecOrdering, $false); | - for case insensitive rowset ordering |

## Building indexes

When the user issues CREATE INDEX statements the current settings of the cmpCollation and whereSecOrdering determine the collation and ordering settings for the indices created. This information is stored with the index and is used the selecting and ordering of data is based on the index.

If the current settings are different from the settings used when the index was created, the selection and ordering of rows based on the index may not be accurate.

When installing your server, choose a configuration for your server based on your country, location and what languages may be passed as variables, literals, or columns in the database. Once chosen, do not modify the settings. If settings are modified, drop and recreate the indexes based on the collation option settings.

To ensure the correct collation settings are always used, you may want to place the SetRuntime commands into the msad$procedures file. This procedure file is executed automatically each time Butler is launched.

## Using ButlerTools

By using the collation option runtime flags all work done by Butler will be standardized. However, ButlerTools doesn't read or utilize the msad$procedures file and there is currently no mechanism for runtime flags in ButlerTools. ButlerTools will therefore default to using ASCII collation without regard to the collation options. Use the appropriate checkboxes in ButlerTools' index definition dialog to override the defaults when necessary or submit CREATE INDEX statements from Butler instead.

# Chapter 3 • Query specifications

This chapter describes how you phrase query specifications used in SELECT statements as well as in searched INSERT, UPDATE, and DELETE statements.

## 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.

The format of a query specification is

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

where the clauses of the specification are as follows:

| | |
|---|---|
| ALL \| DISTINCT | Specifies whether duplicate rows are to be eliminated. |
| *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 contained 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, if neither keyword is specified, 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, or
- 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 SQL 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 SQL identifier. The *expr* may include references to columns of source tables (see the section "Expressions" earlier in this chapter). The specified alias becomes the name of the column in the resulting rowset. If *colalias* is not specified, SQL 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. SQL 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. SQL 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 SQL client applications because the meaning of a select expression may change if the structure of the referenced tables changes.

# Clauses

## 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 this form:

*tblref* [ <AS> *tblalias* ]

where *tblref* specifies a table or view in a currently open database and *tblalias* is a SQL 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 "Search conditions" on page 62.

## 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. For more information on Aggregate functions, see "Aggregate functions" on page 112.

When the GROUP BY clause is not specified, then either:

- no aggregate functions can appear in *select_list* or, or
- 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.

**Note**

Butler SQL 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 "Search conditions" on page 62.

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.

## ODBC Outer-joins

ODBC supports the ANSI SQL-92 left outer join syntax. The escape clause ODBC uses for outer-joins is:

{**OJ** *outer-join*}

where outer-join is:

*table-reference* `LEFT OUTER JOIN` {*table-reference* | *outer-join*}

`ON` *search-condition*

Table-reference specifies a table name, and search-condition specifies the join condition between the table references.

An outer join request must appear after the FROM keyword and before the WHERE clause (if one exists). For example, the following statement uses the short hand syntax to create the result set of the names and departments of employees working on project 455.

```
SELECT EMPLOYEE.NAME, DEPT.DEPTNAME

FROM {oj EMPLOYEE LEFT OUTER JOIN DEPT

ON EMPLOYEE.DEPTID=DEPT.DEPTID}

WHERE EMPLOYEE.PROJID=455
```

To determine the level of outer-joins a data source supports, an application calls SQLGetInfo with the SQL_OUTER_JOINS flag. Drag sources can support two-table outer-joins, partially support multi-table outer-joins, fully support multi-table outer-joins.

## Search conditions

Butler SQL supports standard ANSI SQL search conditions for restricting the rows that appear in a rowset. Search conditions appear in the WHERE and HAVING clauses of various data-manipulation statements. A search condition is a BOOLEAN expression. Its operands may include standard SQL BOOLEAN expressions and the following special search conditions:

- Range check
- Pattern match
- Set membership test
- Null test
- Subquery test

### Range check

The BETWEEN operator tests its three operands to determine whether the value of the first operand lies between the values of the second and third operands. Its syntax is as follows:

*expr1* BETWEEN *expr2* AND *expr3*

The data types of the three operands must be comparable. If any of the operands is NULL, the operator returns NULL. For example, the following search condition finds all instances of “ord_amount” between 2000 and 3000:

```
...where ord_amount between 2000 and 3000
```

## Pattern match

The LIKE operator supports pattern matching for columns with string data types. Its syntax is

*colref* LIKE *pattern*

where *colref* is a column of a source table and *pattern* is a string constant.

The column reference must specify a column with a CHAR or VARCHAR data type. The expression is TRUE if the value of *colref* matches *pattern* in one of the following ways:

- a percent sign (%) in the pattern matches any sequence of zero or more characters in the first operand,
- an underscore ( _ ) in the pattern matches any single character in the first operand, or
- any other character in the pattern must literally match the corresponding character in the first operand.

For example, this search condition finds all cities that begin with *N*:

```
...where city like 'N%'
```

## Set membership test

The IN operator tests whether its first operand, a column reference, matches one of the items in its second operator, a list of values. All the items in the list must be of the same data type, which must be comparable to the type of column referenced. The IN operator's syntax is

*colref* IN (*val_list*)

If *colref* is NULL, the operator returns NULL.

For example, the following search condition finds all instances where "state" is equal to *CA, WA* or *OR*:

```
...where state in ('CA','WA','OR')
```

## The NULL test

SQL contains a special expression, the NULL test, which can be used to determine if a SQL variable or other identifier (such as a column reference) has a NULL value. The NULL test is a unary postfix BOOLEAN operator that returns TRUE if its operand is NULL, and its syntax follows this form:

*value* IS NULL;

The operator always returns TRUE or FALSE; it cannot return NULL. For example, this search condition finds all instances where "qty_ship" has a NULL value:

```
...where qty_ship is null
```

## Subquery tests

The subquery test operators test the rowset produced by a subquery. A subquery is a query specification within a query specification. Four kinds of tests are provided:

| | |
|---|---|
| EXISTS | Tests whether the rowset has at least one row. |
| ALL | Tests whether a specified condition is TRUE for all rows of the rowset. |
| SOME | Tests whether a specified condition is TRUE for at least one row of the rowset. |
| IN | Tests whether the value of a column in the "main rowset" matches a value in one of the rows of the subquery's rowset. |

### EXISTS

The EXISTS operator is a unary prefix operator that returns TRUE if and only if the subquery has one or more rows. Its syntax is as follows:

EXISTS (*qryspec*)

The *select_list* of the query specification must specify a single column or must be an asterisk (*).

## ALL

The ALL operator is a ternary operator. Its first operand is a scalar expression, its second operand is a comparison operator, and its third operator is a subquery that specifies a single-column rowset. Its syntax is as follows:

*expr cmp_op* ALL (*qryspec*)

The ALL operator returns TRUE if and only if the expression

*expr cmp_op value*

is TRUE for each value generated by the subquery. If the expression is NULL or if value is NULL for any row generated by the subquery, then the operator returns NULL.

## SOME

The SOME operator is a ternary operator that determines whether some condition is TRUE for at least one row of the subquery. Its first operand is a scalar expression, its second operand is a comparison operator, and its third operator is a subquery that specifies a single column rowset. Its syntax is as follows:

*expr cmp_op* SOME (*qryspec*)

The SOME operator returns TRUE if and only if the expression

*expr* cmp_op value

is TRUE for at least one value generated by the subquery. If *expr* is NULL, then the operator returns NULL. For compatibility with the ANSI standard, the keyword ANY may replace the keyword SOME.

### IN

The IN operator is a shorthand expression for a specific form of the SOME operator. Its syntax is

*colref* IN (*qryspec*)

where *colref* is a reference to a column in the rowset of the SQL statement containing the search condition, and *qryspec* is a subquery that specifies a single-column rowset. The column reference must not reference a column of a table named in the query specification. The previous expression is equivalent to

*colref* = SOME (*qryspec*)

## Calculated columns

By specifying an expression in a SELECT statement, you can produce a calculated column of query results, as in the following queries:

```
select office_nr + 1 from offices; /* query one */
select sum(office_nr) from offices; /* query two */
```

The data types of calculated columns vary from one DBMS brand to another. In the examples, although "office_nr" has an integer data type, the results produced by the queries may have a data type of SMINT, INTEGER, DECIMAL, or FLOAT, depending on the DBMS brand. SQL does not alter the data type used by the DBMS for calculated columns; it passes the data type through to the client program.

To determine the data types of the calculated columns produced by a query, you must use the PRINTINFO statement. You cannot use the DESCRIBE COLUMNS statement and make assumptions about the data-type conversions performed by the DBMS. In the previous examples, the first query could be rewritten as

```
select office_nr from offices; /* query one */
for each print —> office_nr + 1;
```

and the second query could be rewritten as

```
select office_nr from offices; /* query two */
int i = 0;
for each set i = i + —> office_nr;
print i;
```

In these examples, Butler produces an integer result and SQL handles the arithmetic, returning integer results back to the client application.

# Chapter 4 • Butler SQL runtime environment

## Runtime parameters

Many aspects of the runtime environment on the Butler SQL server can be adjusted to provide efficient and compatible functions with existing applications. Also, several server parameters, including all the settings accessible through the preferences and the tuning dialogs, can be adjusted. Currently, security on these adjustments is limited and much of this section may be less useful to your client/server environment.

### Security issues

For security reasons, several runtime parameters are not currently documented or are restricted to the Database Administrator only.

| **Client settings** | | |
|---|---|---|
| resolveAliases | boolean<br>$true | $true = resolve alias documents on server before printing to the client, $false = return the alias document to the client Macintosh to be resolved by the client application. |
| commitFlushes | boolean<br>$false | $true = on executing a commit statement, clear the client's output queue; $false = don't perform special processing on COMMIT statements. |
| truncateNums | boolean<br>$false | $true = perform truncation; $false = perform rounding (half add and truncate) of smfloat, float, double, money and decimal data types during expression evaluation and data type conversions. See section 18 for additional information on floating point arithmetic and various data type representations. |

| | | |
|---|---|---|
| showRowIDs | boolean<br>$true | $true = include “rowid” as a column of every table; $false = never include the “rowid” column. The rowid column is a pseudo column added to each table. It represents the arrival sequence ordering of the table and will be a unique number never reused during the life of the table (even after deleting rows and compacting the database). |
| defaultScale | integer<br>8 | Defines the default number of fixed decimal places defined for variables declared of type DECIMAL. This setting is necessary because the SQL syntax doesn't provide a mechanism for specifying the number of decimal places to be kept during division, multiplication and type conversions. |
| defaultCharLen | integer<br>255 | Defines the default length assigned to CHAR variables declared. That is, “DECLARE CHAR matcher;” will be declared with a character length as defined by this setting. |
| defaultVarLen | integer<br>32,767 | Defines the default maximum length assigned to VAR-CHAR variables. |
| autoCommitRoll-bac | boolean<br>$false | $true = automatically commit open transactions; $false = automatically rollback open transactions. This setting is used if the table, database or session is closed. |
| tableInColName | boolean<br>$false | $true = include table name; $false = don't include table name. This setting is used for the $colname and PRINT-INFO statements. |
| printfEndsRow | boolean<br>$false | On some implementations of SQL servers, the printf instruction only ends a row if the /n format character exists in the format string literal. In Butler, you can control the behavior of the formatting based on this setting. |
| execClearsErrors | boolean<br>$true | The execution of another fragment for the same client will clear the existing error status to zero. On some Apple SQL servers, this is the only way to automatically clear the error queue. |

| | | |
|---|---|---|
| maxErrors | integer<br>unlimited | Indicates how deep the error queue may be during processing. That is, many other servers will abort processing on the first error so their maxErrors would be set to one. In Butler, our parser can complete an entire fragment's processing before relinquishing control back to the user. This is very useful for checking syntax and having to repeat the send, error, correction cycle. |
| userName | string login name | The current user name for this session. Note: this user name will appear on each line of the activity log pertinent to this client. |
| breakReturnsError | boolean<br>$false | The default action on many Apple SQL servers is to return ceBREAK when a break is detected as being sent by the client. |
| columnTrimSet | string<br><br><br><br>space, null | When storing values into variables and columns, this set of ASCII characters defines characters to be stripped from the end of the value. Trim sets allow for minor input validation and more importantly, less errors in query optimization and retrieval; especially when performing equa-conditions on the columns domain. |
| floatFmt | string<br><br>±0.0000E±0 | Canonical representation of the display conversion of floating point values. |
| printfForm | boolean<br>$false | $false = scientific notation; $true = fixed decimal notation. |
| printfScale | integer<br>8 | This defines the default number of decimal places to show if the printfForm is fixed decimal notation. |
| inactiveTimeout | integer | The maximum amount of time this client may remain connected to the server without performing SQL requests before timing out (whereby the server automatically terminates the session). |
| maxSessionTime | integer | The maximum amount of time this client may remain connected to the server before timing out (whereby the server automatically terminates the session). |

| | | |
|---|---|---|
| userMessage | string<br>SetRuntime<br><br>only | The string passed in the second parameter will be appended into the server's activity log. This is extremely useful for audit purposes and to mark processes while debugging. |
| cmpCollation | integer<br>0 | 0 = ASCII; 1 = localized comparisons. See the separate section on text handling and localization. |
| cmpSecOrdering | boolean<br>$false | $true means that comparisons will be case sensitive; $false means that all case and diacritic differences are ignored. See the separate section on text handling and localization. |
| sortCollation | integer<br>0 | 0 = ASCII; 1 = localized comparisons. See the separate section on text handling and localization to different character sets. |
| sortSecOrdering | boolean<br>$false | $true means that comparisons will be case sensitive; $false means that all case and diacritic differences are ignored. See the separate section on text handling and localization. |
| whereSecOrdering | boolean<br>$false | $true means that comparisons will be case sensitive; $false means that all case and diacritic differences are ignored. See the separate section on text handling and localization. |
| commitFlushDB | boolean<br>$true | This command tells Butler that when committing a transaction, flush all data to disk (from the file's system cache). |
| flushTableUpdates | boolean<br>$false | This command tells Butler that after performing every modification to a database table (during a transaction) ensure that the data is flushed to disk (from the system cache). Warning: this makes processing insert, update and delete very slow. |
| flushDatabase | string n/a | This command takes a database name as the second parameter and forces that database's volume to be flushed to disk. |
| allowReopen | boolean | This command allows a second "open table" command to be executed on a table. The second open table must be "exclusive update" mode and will force Butler to apply a full-table lock until the COMMIT or ROLLBACK of the current transaction is performed. |

| | | |
|---|---|---|
| nullString | string “$null” | Defines what characters are returned when a $null char or varchar data item are sent back to the client (using Print) when printctl 0; is in effect. This is especially useful for compatibility with DataPrism and Excel products. |
| printfEndsBuffer | boolean | Determines specific properties of the print buffering code which may cause compatibility issues. See the separate notes regarding Butler’s print buffers (available in versions greater than 1.2.4). |
| pseudonym | string “” | Once a user has logged onto the server with an authenticated user name, he may choose to be addressed by a psuedonym for more accurate descriptions in the server’s activity log and the Users window. |
| serialNum | integer | This is the current server’s serial number and may be useful for adding software licensing protection to your client front-end. |
| tableUpdateCount | integer | This is a unique (always incremental) counter that indicates the number of insert/updated, or delete statements which have just completed. Use this counter as a way of determining if any updates have occurred to this table since you last referenced it (by saving the update count at that time and comparing it to the current value). |
| printBufferSize | integer | This is a value between 0 and 64K which is used to define the current print buffer size used for this client. Note that the buffer cannot always be enlarged on a client machine and therefore may cause processing problems if dynamically changing the size. A better way would be to change the configuration of the database extension. See the separate section on print buffers for more details. |
| **Advanced** | | |
| outerJoins | boolean $false | Obsolete way to enable left outer join support. To perform a left or right outer join use *= or =* respectively. Note that outer join support is not an official feature of Butler and therefore complicated outer join support is not recommended. |

| | | |
|---|---|---|
| tableRowCount | integer<br><br>(output | SELECT count(*) FROM table; is the long form of $GetRuntime(tableRowCount, table); which Butler can parse and execute much faster. Warning: using this mechanism is not portable to any other server and you cannot introduce WHERE criteria. |
| preferRowsetType | integer<br><br>not yet implemented | This is the default type of rowsets to be utilized for the SELECT statements with a FOR EXTRACT, FOR SCROLLING and FOR UPDATE clause when Butler is running under low memory conditions (when the memory reserve is needed). |
| diskRowsetsEnabled | boolean<br><br>preference | 0 = in application memory only;<br>1 = disk-based when swapping;<br>2 = temporary finder memory when swapping;<br>3 = try temporary then disk when necessary. |
| mfRowsetsEnabled | boolean<br><br>not yet implemented | $true = yes, use disk-based rowsets when necessary;<br>$false = never use disk-based rowsets |
| diskRowsetsSwapAt | integer<br><br>preference | $true = yes, use multifinder temporary memory based rowsets when necessary;<br>$false = never use temporary memory rowsets. |
| mfRowsetsSwapAt | integer not yet implemented | Percentage of Butler's application memory which becomes the threshold under which disk-based rowsets are utilized. |
| trimAllColumns | booleans<br><br>not yet implemented | $true = use the current trim set to strip trailing characters from column data before saving into the database;<br>$false = do not trim columns before saving into the database.<br>Warning: currently RMR must strip null characters from CHAR and VARCHAR columns since Butler stores all text in a null terminated format (regardless of fixed/variable length sizing). |

| | | |
|---|---|---|
| defaultTrimSet | string<br><br><br>$null | Defines the set of ASCII codes which are used as the trim set. This string literal can also be a hex string with every two hexadecimal digits representing the ordinal value of the ASCII representative characters (which allows you to easily handle control characters and other non-printing text). |
| :The following settings are available only with $GetRuntime and will generate an error if used with $SetRuntime | | |
| partitionSixe | integer | This is the current size of the Butler application's memory partition, in bytes. |
| lowMemReserve | integer | This is the current size of the low memory reserve presently assigned to Butler SQL. |
| totalFreeMem | integer | This is the total number of free bytes available in the Butler memory partition. |
| totalContigMem | integer | This is the total number of free bytes available in the Butler memory partition. |
| numUsers | integer | This is the current number of users logged into the Butler server. |
| freeDiskSpace | integer | This is the current size (in bytes) of available free disk space on the boot volume. |
| verifyDatabase | database name | This process could take awhile! See the separate release notes on verify/recover features within the Butler database or call technical support for details. This routine returns 0 if the database is OK and an RMR DBMS error if it is not verifiable. You can also accomplish a database verify in the latest ButlerTools DBA utility and additional information/repair options are available there. |
| timeSlice | integer | The number of ticks (1/60ths of a second) that this client will receive processing time before relinquishing control to another client simultaneously executing. |
| **Programming** | | |

| | | |
|---|---|---|
| activityLogLevel | boolean | This runtime setting changes the default activity log level setting available in the preferences dialog. Using the $Set-Runtime to assign the activity log level allows clients' logging to be differentiated while specific clients are debugged. Further, during the course of a single script, you could choose to change the log level to avoid extraneous messages from being added to the activity log. Changes to begin recording client scripts to the activity log will cause subsequent fragments sent to the server by this client to be logged.<br>0 = no activity logging<br>1 = system errors only<br>2 = authority violations<br>3 = client errors<br>4 = client scripts<br>5 = ODBC Commands<br>6 = ODBC Parameters |
| logQueryOpt | integer | This runtime setting allows a programmer to ensure that the query optimizer is indeed providing optimal access to the database. By setting this property, Butler can direct optimizer messages to either the activity log or directly into the client's output queue so that, using ButlerClient, a complete log of optimization requests can be reviewed.<br>0 = no query optimizer logging<br>1 = send messages to the activity log<br>2 = send messages to the client's output queue<br>3 = send messages to both<br>See the separate section on the query optimizer and generating optimal queries. |
| heapCheckLevel | integer | This forces Butler to periodically perform an internal memory test to ensure that the application heap is still valid and that stack hit heap conditions have not occurred (Butler is performing a lot of operations with the heap checker disabled and therefore stack-hit-heap conditions may not be detected). |
| queryOptFlags | integer | This 32 bit value is a bit mask defining specific actions which could be performed by the query optimizer to force optimizations in a nonstandard manner. See the additional technical note on query optimizations. |

| | | |
|---|---|---|
| debugging | boolean | Butler SQL will operate much slower and perform additional verifications during execution of fragments. Do not use this setting unless directed to do so by a technical support staff member. |
| debugTraps | boolean | $true = debugstr traps in Butler will be executed (you must have a debugger installed). This setting is of absolutely no use to end-users but could be useful when directly working on unexplained system hangs and crashes. |
| onlyParse | boolean | $true = only parse the SQL fragments sent to the server; $false = normal behavior which parses the SQL and then executes the fragment. |
| **Server settings** | | |
| trxnProc | boolean (output | Returns a boolean $true if transaction processing is currently enabled on the server; $false otherwise. |
| inBuffStats | integer | Returns information about the current buffer usage on the server. Note that the information currently available is limited but this process will be improved over time. |
| lowMemTimeout | integer | When the low memory threshold is passed, this time-out value is used to determine how long to wait before purging a client's current cursor. This time-out exists to give clients who have generated a lot of rows of output time to retrieve the rows and therefore reduce the memory consumed back below the threshold. |
| allocDefault | integer | Default for all allocations unless overridden explicitly. |
| allocDefns | integer | Internal RMR structures. |
| allocRecBuffers | integer | RMR record buffers. |
| allocKeyBuffer | integers | RMR key buffers. |
| allocLocks | integer | Row locks. |
| allocSets | integer | RMR rowid sets. |

## Action commands

The following commands can also be executed using the following syntax:

```
$SetRuntime(performAction, <action selector>)
```

This form of the $SetRuntime() function allows specific actions which expand the SQL/SQL syntax to perform specific Butler SQL or RMR actions on the server.

| | |
|---|---|
| cmdInBufferDump | Creates a report of the buffer statistics. |
| cmdLockRelease | Releases all record locks held for this client — WARNING: this could release records involved in a transaction. Subsequent actions by other users may void the ability to rollback changes. |
| cmdForceToDisk | The additional parameter to this call is a cursor reference and this cursor's rowset is automatically swapped to a disk-based rowset. |
| cmdFlushOutput | Clears all outstanding output row buffers from the current client's output queue. |
| cmdFlushInput | Clears all outstanding input row buffers from the current client's input queue. |
| cmdFlushErrors | Removes all outstanding errors from the current client's error queue. |
| cmdFlushAll | Removes all outstanding input, output and errors from the current client's queues. |
| cmdBreak | This is functionally equivalent to making the API call DBBreak(); except that rows currently cached at the client's database extension will not be flushed. |

# Memory Allocation & Buffers

The following guidelines will assist you in configuring your Butler SQL Server. When setting up Butler SQL, it is best to tune your settings based on the memory allocation and the number of users that will be accessing the server.

The preferred setting for Butler SQL memory allocation assumes few users and no msad$procedure files.

For each port established, add 100K, or you can calculate it using the following formula:

> add 4K + (#primary buffers * size) + (#secondary buffers * size)

> if default tuning is used, add approximately 4 + (64 * 0.25) + (16 * 4096) [4+16+64] = 74K

For each user that will connect, add 100K. This assumes use of an msad$procedures and allows working room for database tables and cursors but mileage may vary.

## RAM Allocation & Primary Buffer Settings

Assuming two ports and a MSAD, the following settings are recommended:

| # Users | RAM 68K | RAM PPC | # Primary Buffers |
|---|---|---|---|
| 1 | 3500K | 4600K | 64 |
| 5 | 4000K | 5100K | 64 |
| 10 | 4500K | 6000K | 64 |
| 20 | 5500K | 7000K | 64 |

| # Users | RAM 68K | RAM PPC | # Primary Buffers |
|---|---|---|---|
| 50 | 9000K | 11000K | 64 |
| 100 | 14000K | 16000K | 105 |
| 200 | 23000k | 25000K | 205 |

If the amount of RAM that can be assigned to Butler SQL is limited, you should set the number of maximum users accordingly. For example, if you could only assign 6 MB to Butler SQL, you should set the max users to 20. Also, you should leave some free system memory as Butler SQL can make use of it when required.

## Other settings

We recommend the following settings:

### Tuning Window:

- PRIMARY BUFFER SIZE should be set to 256 bytes and rarely needs to be changed.
- Number of SECONDARY BUFFERS should be 16.
- SECONDARY BUFFER SIZE at 4096 bytes. The secondary buffers are only used for blob data types.
- PROCESS TIME SLICE at 30 ticks.
- MIN. MEMORY RESERVE should be set to 6%.
- PROCESS TIME SLICE at 30 ticks.

### Rowsets Window:

DISK BASED ROWSETS should be turned off when exceeding the memory/user chart above, with the result being a significant impact on performance.

# Access Control

Since the SQL specification allows server control of access at various levels, the access control requirements for Butler SQL are relatively complex. This section attempts to outline the access control requirements in order that they may be implemented in Butler SQL and administered in the ButlerTools application.

## Levels of access control

The SQL specification allows for access control when:

- connecting to a server,
- opening a DBMS, and
- opening a database.

Additionally, table access control is provided for by limiting the tables a user can see using the DESCRIBE TABLES command.

The user must always supply a name and password at the server connection level, and optionally provide the same information at the DBMS and database level. If the identification information is not provided at a lower level, the information provided at the higher level is used.

**Note**

The ability to specify a user name and password at the connection, DBMS open, and database open levels, with each one being potentially different, has likely evolved from the various security requirements for mainframe databases. While Butler SQL currently supports all of these security levels, the need for them on a native Macintosh database server is questionable.

Access control is not only limited to those who can see, but also limited to how they can see. Specifically, a database and table can be opened in a variety of modes and access types. The modes are:

- shared (other users can access the database/table at the same time),
- protected (others can access the database/table at the same time provided they don't directly modify what you are accessing),
- exclusive (no other users can access the database/table).

The allowable access types are:

- read-only (user cannot modify the database/table),
- update (user can modify the database/table).

Thus, a user may have shared/update access to a database, but only shared/read-only access to a particular table in the database. Butler SQL and ButlerTools allow users to be explicitly granted shared, protected and exclusive access to a database or table, despite the fact these modes may be considered hierarchical. For example, a user may be allowed to open a table with shared or exclusive access, but not protected access.

In addition to access rights that are tested on opening databases and tables, users can also be assigned rights on the specific operations they can perform. These “action rights”, which can be individually assigned, include:

- allowing a user to execute SELECT table rows,
- allowing a user to insert table rows,
- allowing a user to update table rows,
- allowing a user to delete table rows,
- allowing a user to index a table, and
- allowing a user to create tables.

As a final point on the access levels, the table access rights may be inherited from database access rights. This allows the server administrator to grant specific access rights to all tables in a database by assigning those rights to the database. Of course, the administrator is always free to then change any or all of those rights on a table-by-table basis. This feature allows for rapid manipulation of access rights.

## Users and groups

As a convenience to the server administrator, group administration is supported to enable identical access privileges to be assigned to a number of users.

Group access control is only meaningful at the database and table levels, and assigning access privileges to a particular group implies that all members of that group would inherit those privileges. A user's access privileges should always limit those inherited from the group; a user's total privileges will then be that which has been inherited from the group minus any additional restrictions that may have been imposed.

For Butler SQL, nested groups are not allowed; i.e. groups cannot be members of other groups. Also, users cannot be members of more than one group.

## Current implementation limitations

The current implementations of Butler SQL and ButlerTools deviate from the features noted above in the following ways.

The DESCRIBE TABLES command does not check table access for a given user, so all tables in the database are shown, even though a user may not be able to open all of them.

Table access rights cannot be assigned in ButlerTools unless some database access rights have been assigned first.

Databases and tables created by a user during a Butler session will not automatically inherit the user's access rights. These rights must be assigned by the server administrator using ButlerTools.

## Transaction Processing

The following section describes the transaction processing model used in Butler SQL. The transaction model includes a description of what a transaction is and how database integrity is maintained in a multi-client environment through the use of row locking.

### Cursor stability

At the root of it all, Butler's transaction model is designed to provide "cursor stability", as defined in the DESCRIBE DBMS section of SQLPR: "the client application [is] sure only that data read through a single cursor is consistent". [This is distinct from "repeatable-read stability", a stricter requirement under which previously read rows are identical if they are reread, until a transaction is completed.]

In the above definition, data consistency is taken to mean that as a cursor is used to traverse a rowset, the data for a particular row will not change in that rowset; a given row may be FETCHed repeatedly and the data will always be the same.

While the multi-user implications of this will be detailed later, put simply, if two clients have built rowsets spanning the same set of table rows, then any updates to those rows made by one client will not affect the other client's data.

### Transactions

A transaction is a series of database commands that modify data in the database, and that have 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 state it

was in before the transaction started. The key is that either all commands of a transaction are successfully performed on the database, or none of them are.

In Butler SQL, all database manipulation commands are assumed to be a part of a transaction until a COMMIT or ROLLBACK command is issued, at which time the transaction is completed. If a COMMIT statement is issued, the changes to the database are made permanent, and if a ROLLBACK statement is issued, then the changes are all reversed.

Butler records all database in a journal file, so that if a ROLLBACK command is issued, all the changes can be reversed. Note that Butler is optimized assuming that the number of ROLLBACK commands will be much less than the number of COMMIT commands, resulting in very low overhead for the execution of the COMMIT, while maintaining full database integrity.

**Note**

Database Data Definition statements (e.g. ALTER DATABASE, CREATE DATABASE) do not allow transactions (e.g. ROLLBACK) in Butler SQL.

## Table locking

In a multi-user environment, certain client transactions must be guaranteed in order that other clients will not interfere with the processing of the transaction. This interference can take the form of actual modification of row data, or simply preventing a row from being accessed or modified due to row locking.

One way of dealing with data contention between clients is through table sharing modes. When a table is opened, it can have one of three sharing modes: SHARED, PROTECTED and EXCLUSIVE. (Note that some clients may not be allowed, by the system administrator, to all of these modes.)

If a client opens a table in EXCLUSIVE mode, then no other client may access that table while it is open. If a client opens a table in PROTECTED mode, then other clients may access the table as long as they do not interfere with the protected client. If a client opens a table in SHARED mode, then other clients have full access to the table; any data contention is handled through row locking (described below).

There is no difference between the shared and protected mode. Also, no read backs are implemented; you can always read regardless of update status.

In addition to the sharing mode, a table is opened with one of two update modes: READONLY and UPDATE. These modes define what operations that client will be allowed to perform on the table. Note that the selected update mode may or may not be possible based on other clients' sharing modes.

While choosing an appropriate sharing mode can protect a client against future clients, the sharing mode may conflict with clients that already have the table open. For example, opening a table in PROTECTED or EXCLUSIVE mode is not possible if another client already has it open for SHARED UPDATE. Similarly, multiple clients cannot open a table in PROTECTED mode unless all of them open it READONLY as well.

While the possible permutations and combinations of sharing and update modes may seem complex, the results are clearly defined, and all contention issues can be (and are) resolved at the time the table is opened. These modes are therefore a very effective way to protect against data contention.

## Row locking

In many cases, the protection offered by table sharing and update modes is too coarse, and would limit other clients too severely. For these cases, data contention is controlled by row locking.

SQL does not explicitly provide for a way to lock rows from other users. It does, however, imply certain locking behaviors under different SELECT conditions, but these can be subtle and provide a lot of latitude for implementation. This section will clearly outline the behavior of Butler SQL under all conditions, within the boundaries required by cursor stability, in order that no ambiguity exists as to what rows are locked, when they are locked, and when they are unlocked.

The first case to consider is when a client is modifying one or more rows. The rows may have been selected explicitly with a SELECT or UPDATE command, or implicitly in the searched UPDATE or DELETE commands. When the rows are explicitly selected, then the row is locked to prevent other users from accessing the data as it is FETCHed, as the positioned UPDATE or DELETE may not occur for some time after the row is FETCHed (or it may not occur at all). When the rows are implicitly selected, the rows are locked as they are actually modified. In both of these update situations, all modified rows are unlocked automatically when a COMMIT or ROLLBACK command is issued.

## READONLY, SCROLLING and EXTRACT

The second case to consider is when a client is simply reading the database and not modifying any rows. This can be achieved with a variety of SELECT modes: READONLY, SCROLLING and EXTRACT. These modes allow different cursor positioning methods to access the rowset created by the SELECT, and have varying overheads associated with building the rowsets to support the cursor positioning. What is relevant to this discussion is that for Butler to be cursor-stable, the data in the rowset must not change while the cursor for the rowset is defined; this must remain true for each of these SELECT modes.

## READONLY

When data is selected in READONLY mode, the rows can only be accessed through FETCH NEXT commands, and a previously FETCHed row cannot be re-accessed unless a new SELECT is done. Because of the buffering mechanisms built into Butler, it is not required that you create an explicit read lock on the current row to ensure that the data in the

buffer is not modified. Another client reading the same row, will have a separate copy of the data; any modifications to that data will not affect the data of the first client.

### SCROLLING

When data is selected in SCROLLING mode, rows previously FETCHed may be re-accessed. However, since a shadow rowset is created and filled in as each row is FETCHed, the Butler buffering again ensures that any previously read row will not change in the rowset.

### EXTRACT

Selecting data in EXTRACT mode ensures that rowset data will not change simply by copying the rowset data as the SELECT is executed. For locking purposes, it behaves in exactly the same way as SCROLLING mode; the only difference is when the shadow rowset is created.

Finally, the important point in the three read scenarios is that cursor stability requires only that a client's copy of row data not be modifiable by other clients; this is achieved through buffering and the rowset shadowing of some SELECT modes. No additional locking is required. This is distinct from the "repeatable-read stability" described earlier, which would require that any row read from the database be locked against update for the entire transaction. This type of read locking is currently unsupported by Butler. Note that repeatable-read stability can still be achieved using a cursor-stable DBMS like Butler through the use of EXCLUSIVE sharing modes when a table is opened.

# Chapter 5 • Function Reference

This section describes the functions available in Butler SQL. There are several different categories of functions available

- standard ODBC functions supported by Butler SQL
- Butler SQL's built-in functions

## ODBC Scalar functions

The ODBC API supports five categories of scalar functions: numeric, string, time and date, system, and conversion. Butler SQL supports those functions from both ODBC and DAL/DAM connections The following sections describe each scalar function. The functions are listed at the left with an explanation for each function following.

### String functions

String functions are used to exact substrings, perform case conversion, and determine string length.

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 string_exp2 is inserted into string_exp1 at position start, replacing characters from position start to position start + length. If length is zero, then string_exp2 is inserted without replacing any characters in string_exp1.

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 argument, start, is specified. If start is specified, the search begins with the character position indicated by the value of start. The first character position 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 supported by Butler SQL.

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.

DEGREES( numeric_exp)

Returns the number of degrees converted from numeric_exp 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.

LOG10( float_exp)

Returns the base 10 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.

RADIANS( numeric_exp)

Returns the number of radians converted from numeric_exp degrees.

RAND([ integer_exp])

Returns a random floating point value using integer_exp as the optional seed value.

ROUND( numeric_exp, integer_exp)

Returns numeric_exp rounded to integer_exp places right of the decimal point. If integer_exp is negative, numeric_exp is rounded to | integer_exp | places to the left of the decimal point.

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.

TRUNCATE( numeric_exp, integer_exp)

Returns numeric_exp truncated to integer_exp places right of the decimal point. If integer_exp is negative, numeric_exp is truncated to | integer_exp | places to the left of the decimal point.

## 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.

DAYNAME( date_exp)

Returns a character string containing the data source–specific name of the day (for example, Sunday, through Saturday or Sun. through Sat. for a data source that uses English, or Sonntag through Samstag for a data source that uses German) for the day portion of date_exp.

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.

MONTHNAME( date_exp)

Returns a character string containing the data source–specific name of the month (for example, January through December or Jan. through Dec. for a data source that uses English, or Januar through December for a data source that uses German) for the month portion of date_exp.

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.

TIMESTAMPADD( interval, integer_exp, timestamp_exp)

Returns the timestamp calculated by adding integer_exp intervals of type interval to timestamp_exp. Valid values of interval are the following keywords:

SQL_TSI_FRAC_SECOND

SQL_TSI_SECOND

SQL_TSI_MINUTE

SQL_TSI_HOUR

SQL_TSI_DAY

SQL_TSI_WEEK

SQL_TSI_MONTH

SQL_TSI_QUARTER

SQL_TSI_YEAR

Fractional seconds are expressed in billionths of a second. For example, the following SQL statement returns the name of each employee and their one-year anniversary dates:

SELECT NAME, TIMESTAMPADD (SQL_TSI_YEAR, 1, HIRE_DATE) FROM EMPLOYEES

If timestamp_exp is a time value and interval specifies days, weeks, months, quarters, or years, the date portion of timestamp_exp is set to the current date before calculating the resulting timestamp.

If timestamp_exp is a date value and interval specifies fractional seconds, seconds, minutes, or hours, the time portion of timestamp_exp is set to 0 before calculating the resulting timestamp.

An application can determine which intervals a data source supports by calling the SQLGetInfo function with the SQL_TIMEDATE_ADD_INTERVALS option.

## System functions

The following list describes Butler SQL's system functions.

DATABASE( )

Returns the name of the database corresponding to the current connection.

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 name used to log on to Butler SQL).

## Explicit Conversion function

The data type conversion function converts a value to a different data type.

*Syntax* CONVERT(*value_exp, data_type*)

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

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

The ODBC syntax for the explicit data type conversion function does not support specification of conversion format. Specification of explicit formats is supported by Butler SQL, and therefore an ODBC driver must be used to specify a default value or implement format specification to support the data type formats of the specified Butler SQL database.

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

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

converts the result of the CURDATE scalar function to a character string.

## Butler SQL Built-in functions

Butler offers a number of built-in functions that can be used in SQL expressions. These functions include general-purpose functions, string-manipulation functions, and functions that can be used to examine rowsets. Like system variables, the names of the built-in functions begin with a dollar sign ($), so they cannot be confused with user-defined SQL functions.

Although you can use these functions from ODBC, they are not part of the standard ODBC syntax. If you use them, your application may not be compatible with other ODBC-compliant DBMS's.

### General-purpose functions

The general-purpose built-in functions are as follows:

| | |
|---|---|
| $format( ) | Formats data into a character string |
| $len( ) | Returns the length of a data item |
| $typeof( ) | Returns the SQL data type of an item |
| $freemem( ) | Returns the amount of free server application memory, in bytes |
| $ticks( ) | Returns the number of ticks (one tick is approximately 1/60 of a second) that have elapsed since the server Macintosh was started |

The $format( ) function

The **$format( )** function is the most complex of the built-in functions. It takes a variable number of arguments and combines them into a single VARCHAR string according to a user-specified format. The

function performed by **$format( )** is parallel to that performed by the C language function sprintf( ), but it has a slightly different form. The form of the SQL **$format( )** function is

```
outstr = $format ( fmtstr, data1, data2, . . . )
```

The return value of *outstr* is a VARCHAR string containing a formatted version of the data items (*data1,* and so on). The first argument of *fmtstr* is a string expression that specifies the format to be used. The remaining arguments are the data items to be combined according to *fmtstr*. The specific contents of *fmtstr* determine the number of data items.

The format string can contain three types of formatting specifications: ordinary characters, special characters, and conversion specifications. These formatting specifications can be intermixed within the format string and are interpreted in left-to-right order.

Ordinary characters (such as *a*, *b*, and *c*) appearing in *fmtstr* stand for themselves. When the **$format( )** function encounters an ordinary character, the character is simply appended to the end of *outstr*.

Special non-printing characters (such as a tab) can be specified in *fmtstr* by a two-character sequence consisting of the backslash (\) character and a second character. A backslash prefix can also be used to force literal interpretation of characters that otherwise have special meanings in *fmtstr,* such as the percent sign and the backslash itself. These special two-character sequences include:

| | |
|---|---|
| \b | generates a backspace character |
| \n | generates a newline character |
| \r | generates a return character |
| \t | generates a tab character |
| \\ | generates a backslash character |
| \% | generates a percent character |

**Note**

The backslash character is read and interpreted by the **$format( )** function itself, not by the Butler SQL parser (unlike the backslash in C, where the compiler replaces the two-character sequence with the special character itself). Thus, the backslash cannot be used to escape characters in SQL string literals or to escape the quote characters. Its use is limited to format strings.

The following example shows the use of ordinary and special characters:

```
declare varchar myvar ;
myvar = $format("My Name Is:\tJoe\n") ;
print myvar ;
```

The output from the example contains a tab character and ends in a newline character.

A conversion specification within *fmtstr* directs the **$format( )** function to take the next data argument and convert it into a sequence of characters to be appended to the end of *outstr*. A conversion specification is introduced by the percent character (%) and concluded by a conversion character (*fmtchar*) in the following form:

% [ - ] [ *width* ] [ *.precision* ] *fmtchar*

**Note**

The spaces in the above specification are for readability only; the conversion specification must contain no embedded spaces between the % character and *fmtchar*.

The *fmtchar* conversion character determines the type of conversion taking place. The following *fmtchar* values are supported:

| | |
|---|---|
| d or u | Expects an INTEGER data argument and converts it to a string, representing its value, in the SQL integer literal format. |
| p | Expects a DECIMAL or MONEY data argument and converts it to a string, representing its value, in the SQL decimal literal format. |
| c | Expects the ASCII code of a single-character as a data argument and copies the character. |
| f | Expects a floating-point data argument and converts it to a string, representing its value, in the SQL floating-point literal format. |
| s | Expects a string data argument and copies the string value according to other conversion parameters. |
| x or X | Converts its data argument to a sequence of characters that are the unsigned hexadecimal representation of the data item. A capital *X* results in uppercase hex digits A–F; a lowercase *x* results in lowercase hex digits a–f. |

A minus sign (–) in the conversion specification indicates that the data is to be left justified upon conversion. Otherwise the data is right justified. Justification takes place relative to *width* and *precision,* which specify the width of the converted data string.

For the exact numeric formats (d, u, and p), *width* specifies the minimum width of the output field. It is an unsigned integer with an optional leading zero. The converted number will be printed in a field at least *width* positions wide. If necessary, the number will be padded (on the left or right, depending on the minus sign) with a padding character to fill the field. The padding character is a blank, unless a leading zero is specified for *width,* in which case the padding character is a zero (0). The *precision* parameter specifies the maximum size of the output field. If *precision* is specified, the output string for the data argument will be truncated so that it does not exceed *precision* positions.

The following example illustrates the exact numeric formats:

```
declare integer myint= 12345 ;
print $format(":%10d:",myint) ;
print $format(":%-10d:",myint) ;
print $format(":%010d:",myint) ;
print $format(":%-010d:",myint) ;
print $format(":%3d:",myint) ;
```

The example generates the following output:

```
:     12345:
:12345     :
:1234500000:
:0000012345:
:12345:
```

For the floating-point format (f), *width* again specifies the minimum size of the output field, with the same zero-padding or space-padding as described for the exact numeric formats. The *precision* parameter, if present, specifies the number of digits to the right of the decimal point that are to be printed in the output string.

The following example illustrates the floating-point format:

```
declare float myflt = 1.2345E3 ;
print $format(":%10f:",myflt) ;
print $format(":%-10f:",myflt) ;
print $format(":%10.2f:",myflt) ;
print $format(":%10.5f:",myflt) ;
```

The example generates the following output:

```
:  1.2345E3:
:1.2345E3  :
:    1.23E3:
: 1.23450E3:
```

For the string format (s), *width* and *precision* and the minus sign provide precise control over string justification and padding; *width* specifies the minimum number of characters in the output field, and *precision* specifies the maximum number of characters of the data argument to be copied to the output field. The string data argument is truncated or blank-filled to fit the field specification, as shown in the following example:

```
declare varchar mystr = "SQL strings" ;
print $format(":%10s:",mystr) ;
print $format(":%-10s:",mystr) ;
print $format(":%20s:",mystr) ;
print $format(":%-20s:",mystr) ;
print $format(":%20.10s:",mystr) ;
print $format(":%-20.10s:",mystr) ;
print $format(":%.10s:",mystr) ;
```

The example generates the following output:

```
:SQL strings:
:SQL strings:
:       SQL strings:

:SQL strings       :

:        SQL string:
:SQL string        :
:SQL string:
```

For the string format (s) specification only, *fmtchar* can be preceded by a caret (^) or an exclamation point (!) character. A caret causes the output field contents to be converted to all uppercase characters. The exclamation point causes the field contents to be converted to all lowercase characters.

Finally, *width* and *precision* in a conversion specification can be specified as the special-character asterisk (*). If an asterisk appears for one of these parameters, the numeric value for the parameter is taken from the next data argument to the **$format( )** call. For example, the function call

```
integer i = 20 ;
integer j = 10 ;
$format(":%*.*^s:",i,j,"SQL Strings") ;
```

is interpreted as

```
$format(":%20.10^s:");
```

and produces the output

```
:SQL STRING         :
```

For every call to the **$format( )** function, the number of conversion specifications in *fmtstr* and the number of data arguments supplied should match. If the number of data arguments is less than the number of conversion specifications, errors may result.

### The $len( ) function

The **$len( )** function returns the length of its argument, which must have a string data type (CHAR or VARCHAR).

**Note**

The **$len ( )** function can also be used with Butler SQL's LONGCHAR and binary data types.

The function returns the number of characters (or bytes) in the string. The following example shows the use of the **$len( )** function:

```
declare varchar mystr1 = "Hello" ;
declare char mystr2 = "World" ;
declare varchar

mystr3; mystr3 = mystr1 + " " + mystr2 ;
print $len(mystr3) ;
```

The example outputs the single integer 11, which is the number of characters in the concatenated string.

### The $typeof( ) function

The **$typeof( )** function returns the data type of its argument, which may be an expression of any valid SQL data type. The function returns an INTEGER value that is the SQL code for the data type (see Table 1-8, "SQL data types," on page 33, for the data type represented by the code). The following example shows the use of the **$typeof( )** function:

```
declare integer myint = 17 ;
declare varchar mystr= "Hello" ;
declare generic mygen ;
mygen = mystr ;
print $typeof(myint), $typeof(mystr), $typeof(mygen) ;
```

The example outputs the three integers: 3 (for INTEGER), 12 (for VARCHAR), and 12 (for VARCHAR).

### The $freemem( ) function

The **$freemem( )** function takes no argument and returns the amount of free server application memory, in bytes. This is useful especially when dealing with the large data items possible using Butler's extended data types. Clients can use the result of this function to help them determine, before sending a particular data item, whether the server has enough memory to receive and process it. See "Other Memory Considerations" on page 65 for more information on the memory requirements of the Butler server when working with large data items.

Example:

```
IF $freemem() > 1000000 {

DOCUMENT bigDoc = $loadFileNamed("Server HD:Big File");

PRINT "File loaded OK.";

}

ELSE

PRINT "Not enough server memory to load file";
```

The $ticks( ) function

The **$ticks( )** function takes no argument and returns the number of ticks (one tick is approximately 1/60 of a second) that have elapsed since the server Macintosh was started. This function is useful for doing precise timing of server operations.

Example:

```
PRINT $ticks();
```

## ODBC built-in functions

When you are writing SQL using scalar functions and other ODBC built-ins, if you wish to make your code as generic as possible, you should use what is known as a vendor expression. The vendor expression allows you to use a predefined structure which does not assume anything about the server to which you are connecting.

The vendor expression comes in two forms:

- (Long form) --(*vendor(vendor-name), product(product-name0 extension *)--
- (Short form) {extension}

The long form of the vendor expression is used hardly ever and we will only deal with the short form.

Butler's SQL dialect supports a number of ODBC vendor expressions such as date, time, timestamp conversions, and scalar functions calls.

Examples of their use are as follows:

```
UPDATE employee SET birthday = {d '1967-01-15'}
    where name = "Smith, John";
SELECT {fn UCASE(NAME)) FROM employee;
```

Butler's SQL dialect supports use of scalar functions in any expression without using a vendor expression clause unless it needs it to avoid ambiguity. If you are writing code that will generally be hitting Butler SQL, you can make your code more readable and reduce the complexity by not using the vendor expression.

Examples of the use of scalar functions are as follows:

```
SELECT {fn SUBSTRING(NAME, 1,3)} FROM employee;

SELECT SUBSTRING(Name, 1,3) FROM employee;

SELECT UCASE(NAME) FROM employee;
```

## String-manipulation functions

These built-in functions are used for string manipulation:

| | |
|---|---|
| $left( ) | Returns a leading substring 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 |
| $rtrim( ) | Trims trailing blanks from a string |
| $substr( ) | Returns a substring of its argument |
| $trim( ) | Trims leading and trailing blanks from a string |

### The $left( ) and $right( ) functions

The **$left( )** and **$right( )** functions provide the same string searching capability as the **$locate( )** function, but they extract portions of the input string rather than returning a position within it. These functions have the form:

outstr = $left( *instr*, *pattern* [ , *count* ] )
outstr = $right(*instr*, *pattern* [ , *count* ] )

where *instr* is a string to be searched and *pattern* is the substring to be located, as in the **$locate( )** function. The **$left( )** function returns a leading substring of *instr*, up to but not including the first character of *pattern* found within *instr*. The **$right( )** function returns a trailing substring of *instr*, beginning with the first character following *pattern*. The *count* argument is optional for both functions. If it is omitted, the functions search for the first occurrence of *pattern* within *instr*, for example:

```
$left("database.table.column", ".")yields"database"
$right("database.table.column", ".")yields"table.column"
```

As in the **$locate( )** function, a positive *count* value causes the search of *instr* to continue until *count* occurrences have been found. A negative *count* value causes the search to begin at the end of *instr* and proceed from right to left:

```
$left("database.table.column", ".", 2)yields"database.table"
$left("database.table.column", ".", -1)yields"database.table"
$left("database.table.column", ".", -2)yields"database"
$right("database.table.column", ".", 2)yields"column"
$right("database.table.column", ".", -1)yields"column"
$right("database.table.column", ".", -2)yields"table.column"
```

### The $locate( ) function

The **$locate( )** function locates a substring within a string and returns its position as an integer. Its form is:

```
position = $locate(instr, pattern [ , count ] )
```

where *instr* contains the string to be searched, and *pattern* is the substring to be located within *instr*. The *count* argument is optional. If it is omitted, the first occurrence of *pattern* within *instr* is located and its position is returned by the function:

```
$locate("database.table.column", ".")yields9
```

If *count* is specified, it instructs the function to continue past the first occurrence of *pattern* until *count* occurrences have been found:

```
$locate("database.table.column", ".", 2)yields15
$locate("database.table.column", ".", 1)yields9
```

A negative *count* value instructs the function to begin its search at the end of *instr* and search backward through the string:

```
$locate("database.table.column", ".", -1)yields15
$locate("database.table.column", ".", -2)yields9
```

Note that, with a negative *count* value, even though the search proceeds from right to left, the returned *position* is always the position relative to the beginning of *instr,* with the first character as position 1.

The $substr( ) function

The **$substr( )** function is used to extract a substring from a string variable or constant. The form of the function is:

```
outstr = $substr( instr, position [ , length ] )
```

where *instr* is the input string from which the output string (*outstr*) is to be extracted, and *position* is an integer specifying the starting position within *instr.* The first character of *instr* is specified as position 1. The *length* argument is optional and specifies the number of characters to extract. If *length* is omitted, the function extracts the entire remainder of *instr* up to its final character, for example:

```
$substr("ABCDE", 3)yields"CDE"
```

If *length* is specified, then up to *length* characters are extracted:

```
$substr("ABCDE", 3, 2)yields"CD"
$substr("ABCDE", 3, 7)yields"CDE"
```

If *length* is negative, then *position* specifies the last character of the extracted string, and extraction proceeds to the left instead of to the right:

```
$substr("ABCDE", 3, -2)yields"BC"
```

Finally, if *position* is negative, then it specifies a position relative to the end of *instr.* The final character of *instr* is designated by the value –1:

```
$substr("ABCDE", -2, 2)yields"DE"
$substr("ABCDE", -2, 1)yields"D"
$substr("ABCDE", -2, -2)yields"CD"
```

The $trim( ), $ltrim( ), and $rtrim( ) functions

Three other built-in string functions are used to strip leading and trailing blanks from SQL character strings. All three functions take a single string argument and return a string with the blanks removed. The **$trim( )** function strips leading and trailing blanks. The **$ltrim( )** function strips leading blanks from its argument, but leaves trailing blanks. The **$rtrim( )** function strips trailing blanks, but leaves leading ones. Thus:

```
$trim("  this is a string   ")yields"this is a string"
$ltrim("   this is a string   ")yields"this is a string   "
$rtrim("   this is a string   ")yields"   this is a string"
```

The **$rtrim( )** function is especially useful for stripping trailing blanks from CHAR data that is retrieved from a database, effectively converting it to a VARCHAR representation of the same string.

The following procedure, *printtrim,* deletes leading and trailing spaces from rowsets returned by a SQL query:

```
procedure printtrim ()
{
  integer i;
  for each  /* for each row in the default cursor, $cursor */
  {

  for (i=1;i<=$cols(); i++)  /* for each column in a row */
   print $trim(varchar —> :i);  /* type-cast col i to varchar
   and trim it */
  }
}
end procedure printtrim;
```

## Functions for examining rowsets

Several built-in system functions are useful for examining rowsets produced by the SELECT and DESCRIBE statements. These functions all take a cursor (optional) and a column number as arguments and return various pieces of information about the specified column in the current row of the specified rowset, as shown in the following example:

```
declare cursor c ;
declare varchar name ;
```

```
declare int type, len ;
select * from offices into c ;
fetch ;
for (i = 1; i <= $colcnt ; i++) {

   name = $colname(c,i) ;
   type = $coltype(c,i) ;
   print "Column", i, "name:", name, "type:", type ;
}
```

The built-in functions that describe individual columns of a rowset, and the information returned by each, are shown in Table 9.

**Table 9: Functions that describe columns of a rowset**

| Function | Data type | Information |
|---|---|---|
| $colname( ) | VARCHAR | Column name |
| $coltype( ) | INTEGER | Column's data type code |
| $collen( ) | INTEGER | Column length |
| $colplaces( ) | INTEGER | Number of decimal places |
| $colwidth( ) | INTEGER | Display width for column |

The built-in functions that describe a rowset as a whole are shown in Table 10.

**Table 10: Functions that describe a rowset**

| Function | Data type | Information |
|---|---|---|
| $cols( ) | INTEGER | Number of columns in a rowset |
| $rows( ) | INTEGER | Number of rows in a rowset |

## Aggregate functions

Aggregate functions are used to compute a statistical function of the values in a particular column over all of the rows in a row group. The row group is the grouping specified by a GROUP BY clause or the entire collection of rows if no GROUP BY clause is present. There are five statistical functions that take an expression as an argument:

| | |
|---|---|
| MAX( ) | Computes the maximum value of a column in all rows of the group. |
| MIN( ) | Computes the minimum value of a column in all rows of the group. |
| AVG( ) | Computes the average value of a column over all rows of the group. |
| SUM( ) | Computes the sum of the values of a column over all rows of the group. |
| COUNT( ) | Computes the number of rows in the group with non-null values of a column. |

A sixth statistical function takes no arguments:

| | |
|---|---|
| COUNT(*) | Computes a count of all rows in the group |

Aggregate functions may appear in expressions in the following situations:

- In the *select_list* of a query specification when a GROUP BY clause is present. In this case, the aggregate function specifies how the value of the corresponding column is calculated for each summary row in the rowset.
- In the *have_condition* of the HAVING clause of a query specification. In this case, the aggregate function specifies how a value is to be calculated for a group in order to evaluate the search condition and determine whether the group will be included in the rowset.

Aggregate functions may not be "nested", or appear in expressions within the WHERE clause, except when the WHERE clause is in a subquery and the argument of the aggregate function is an outer reference to a column of a containing the subquery.

The MAX( ) and MIN( )

The MAX( ) and MIN( ) aggregate functions take an expression as their argument and return the largest and smallest values, respectively, when the expression is evaluated for each row in the row group. The expression can be a numeric expression (involving references to columns with numeric types) or a string column reference. Examples of the MAX( ) and MIN( ) functions follow:

| | |
|---|---|
| MIN(quota) | Minimum quota |
| MAX(sales - quota) | Biggest amount over quota |
| MIN(last_name) | Earliest name, in alphabetical order |

AVG( ) and SUM( )

The AVG( ) and SUM( ) aggregate functions take a numeric expression as their argument and return the average and sum of the values, respectively, when the expression is evaluated for each row in the row group. Examples of the AVG( ) and SUM( ) functions follow:

| | |
|---|---|
| AVG(quota) | Average quota |
| SUM(sales - quota) | Total amount over/under quota |
| AVG(sales/quota) | Average quota performance |

Just as the DISTINCT keyword in a query specification can eliminate duplicate values in a rowset, the DISTINCT keyword can be specified to eliminate duplicate values of the argument before the AVG( ) and SUM( ) functions are applied. When DISTINCT is specified as part of an aggregate function argument, all duplicate values of the argument are eliminated before the aggregate function is applied. With the DISTINCT keyword, the argument to the aggregate function must be a simple column reference with an optional unary minus sign; more complex expressions are not allowed. Two examples of using DISTINCT with aggregate functions follow:

| | |
|---|---|
| AVG(DISTINCT quota) | Average of all the distinct quotas |
| SUM(DISTINCT quota) | Sum of all the distinct quotas |

Used in this context, the DISTINCT keyword causes duplicate row elimination in much the same way it does when it is used at the beginning of *select_list*. For this reason, the DISTINCT keyword can appear at most once in a query specification, except that it may appear in subqueries contained within that query specification.

COUNT( )

The COUNT( ) aggregate function must be used with the DISTINCT keyword. It takes a column reference as its argument and returns the number of non-NULL occurrences of the column in the row group. The final aggregate function, COUNT(*), returns a count of the total number of rows in the row group, regardless of the contents of the rows. Examples of these functions follow:

| | |
|---|---|
| COUNT(DISTINCT quota) | Number of different quotas |
| COUNT(DISTINCT office_nr) | Number of different offices |
| COUNT(*) | Number of sales reps |

**Note**

Butler SQL does allow COUNT (DISTINCT *).

## Identifiers in query specifications

An identifier appearing in a query specification can refer to many different types of entities, including a database, table, column, cursor, or SQL variable. References to databases and tables are always clear from the context. However, column references, cursor-based column references, and SQL variable references can potentially be confused, particularly if qualifiers are omitted.

## Cursor-based column references

Cursor-based column references within a query specification must always be explicitly qualified by a cursor name; the cursor qualification may not be dropped. Note that when a cursor-qualified reference appears in a query specification, its value is effectively a constant for purposes of evaluating the query, because it identifies a column in a row of some other rowset, not the rowset being created by the query specification.

## Table-qualified references

A table-qualified reference is always interpreted as a reference to a column of a source table. The presence of the table qualifier prevents its accidental interpretation as a variable name. The value of such a reference when used in an expression is the value of the referenced column in the source row.

If the table-qualified reference occurs in a subquery, it may refer to a source table of a containing query specification rather than the one in which it appears. This kind of reference is called an outer reference. The value of an outer reference is effectively a constant for purposes of evaluating the subquery because its value does not change over the rows of the subquery's rowset.

When a table-qualified reference encounters an unqualified reference, SQL first tries to interpret it as a reference to a SQL variable currently in scope. If it cannot be so interpreted, SQL tries to interpret the reference as a column reference for a source table. Interpretation of an identifier as a SQL variable can be forced by prefixing a colon (:) to the variable name. Thus, the token

*:myvar*

appearing in a SQL statement is always interpreted as a reference to the SQL variable *myvar*. This notation is the same as that used for host variables in the ANSI SQL standard.

Despite the complexity of the previous discussion, note that a properly qualified identifier is always unambiguously interpreted:

- a cursor name qualifier forces a cursor-based column reference,
- a table qualifier forces a column reference to a source table, and
- a leading colon forces interpretation as a SQL variable name.

# Chapter 6 • Designing Butler SQL databases

## Normalized Databases

In database theory, there is the concept of an imaginary universal table that represents all of the data in the database. However, we would never really wish to store the data in a universal table because of the various anomalies and redundancies that are present in a group of data.

Normal form is a theoretical concept developed as a set of rules to decompose a universal table into multiple tables, and the decomposition process along the normal forms is called normalization. The most important of these normal forms are:

- First Normal Form, a relation is in 1NF if and only if all underlying domains contain scalar values only. In the other words, there must be a scalar value in each intersection of row and column.
- Second Normal Form, a relation is in 2NF if and only if it is in 1NF and every nonkey attribute is irreducibly dependent on the primary key. In the other words, the primary key is the column or columns whose values uniquely identify a row in a table, and all the columns rather than the primary key must be dependent on the entire primary key.
- Third Normal Form, arelation is in 3NF if and only if it is in 2NF and every nonkey attribute is nontransitively dependent on the primary key. In the other words, no nonkey column depends on another nonkey column or columns.

Butler SQL databases function most effectively along the structure of a normalized database.

In general, the advantages of normalization include:

- clear logical design,

- reduced data redundancy,
- performance gains,
- protection against update and delete anomalies, and
- ease of restructuring of databases.

As normalized databases, Butler SQL databases decentralize a mass of data into uniquely defined tables. When using ButlerTools to define the schema of a Butler SQL database, you should attempt to exploit its method of associating data. In other words, you should attempt to design a database which takes advantage of the unique attributes of the data it is storing.

A table can have a key column, which is a column or a group of columns whose values uniquely identify each row. An example of a key column would be a row_id column. A row_id column can be used to store a different number with every row in a table and thereby uniquely identify each row.

The relationship between two tables is defined by one or more columns which both tables contain. For example, in an order entry system, the customer name and address information is stored in one table, such as a "Customers" table, and the ordering information is stored in a separate table, such as an "Orders" table. Both tables are related by a common customer number which is recorded in a customer number column ("cus_num") in each table.

To find information about a particular order, you specify the common customer number from the "Orders" table, which, in turn, enables you to find the customer name and address from the "Customers" table.

By using ButlerTools to design a normalized database you will be able to manage your data in the most profitable manner. Your Butler SQL databases will allow changes and manipulation of data quickly and safely, while maintaining unique definitions for each of the entities being described.

# Butler SQL Create Formats

Butler can parse create database, create index, and a create table commands. The following is the syntax for these create commands.

## Create Database

```
CREATE [dbbrand] DATABASE "dbName"
[<IN> LOCATION "dbLoc" ]
[ <AS> USER "user" [<WITH> PASSWORD "passwd"]] ;
```

## Create Table

```
CREATE TABLE "tableName"
 (
FIELDNAME FIELDTYPE [(LENGTH [,NUMDECS])] , …
 ) ;
```

FieldType can take on the following values:

- BOOLEAN, DATE,
- SMINT, TIME,
- INTEGER,TIMESTAMP,
- SMFLOAT, CHAR,
- FLOAT, VARCHAR,
- MONEY, LONGCHAR,
- DECIMAL, and
- NUMBER.

Length is valid for all field types.

The number of decimals will be ignored if the field type is not a numeric value.

### Create Index

The Create Index statement has been changed so that a length can be specified for a keyed column.

```
CREATE [UNIQUE] INDEX "indexName" ON "tableName"
( FIELDNAME[(Length)] [<ASC>|<DESC>] , … ) ;
```

The field name must be a valid field in the table.

The length of the keyed column can be specified after the field name.

The keyed column's length should be specified in round brackets and should be an INTEGER expression.

If a sorting order is not specified then ascending (ASC) is assumed.

## Indices

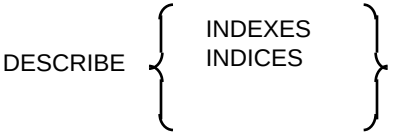
### Describing Indices: "The key to SQL"

An index consists of a number of keys with a single key definition, and a key contains numerous keyed columns or fields.

SQL provides the describe statements as a means for the script writer to find out information about the characteristics of the dbms, databases, tables and columns of the system being accessed. Since the RMR dbms utilizes indexes to a large extent, we have extended SQL to provide the script writer with the Describe Indexes and Describe Keys statements. This will enable a user to query the database about what indexes are associated with a table and what the keyed columns are for that index.

## DESCRIBE INDEXES (Indicies)

*Syntax*




*Parameters*

*tblRef* — A table reference indentifying the table whose indexes are to be described.

*cursor* — A CURSOR variable to receive the rowset.

*Results* — The resulting rowset has one row for each index that is associated with the table. Each row contains the 5 columns of information shown in the following table.

**Table 11: Describe Index rowset**

| Column # | Data type | Name | Information |
|---|---|---|---|
| 1 | VARCHAR [31] | indname | name of the index |
| 2 | BOOLEAN | unique | is the index unique |
| 3 | INTEGER | numkeyed | number of columns keyed |
| 4 | INTEGER | numkeys | number of keys |
| 5 | VARCHAR [31] | primekey | name of the primary keyed column |

## Describe Keys

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

*Parameters*

*indexName* — The name of the index whose keyed columns are to be described.

*cursor* — A CURSOR variable to receive the rowset.

*Results* — The resulting rowset has one row for each keyed column that is associated with the index. Each row contains the 7 columns of information shown in the following table.

**Table 12: Describe Keys rowset**

| Column # | Data type | Name | Information |
|---|---|---|---|
| 1 | VARCHAR [31] | keyname | name of the keyed column |
| 2 | VARCHAR [31] | keytitle | title of the keyed column |
| 3 | INTEGER | size | size of the keyed column |
| 4 | SMINT | keytype | data type of the keyed column |
| 5 | CHAR [4] | sort_order | ASC, Descending |
| 6 | BOOLEAN | case_sens | second ordering |
| 7 | CHAR [1] | comparetype | (A)scii, (L)ocalized |

## Composite Indexing in Butler SQL

Index selection and usage has been improved. In earlier Butler SQL, Butler's choice of which index to use was very limited. In subsequent versions, we have added significant improvements in the choice of index and also optimized its execution.

The following "Question and Answer" approach should help you understand how to take advantage of this new feature.

## Q: What is the composite index?

A: The composite index is an index consisting of more than one column.

The first column is called the primary column of the composite index, and the order of the columns specified in the composition of the index is significant.

## Q: Why do I need a composite index?

A: Building and using composite indices properly could significantly increase query speed.

In general, if several columns in a table are most likely used together, and the search conditions are equalities with at most one exception, you should consider building a composite index on the several columns. As we said earlier, the order of the columns in the index is important, and the general rule here is that the column in the non-equalities must be the last column, and the others are ordered by their frequency in the whole set of queries. Also, we should mention that the leading portion of the composite index could be utilized in query optimization, so avoid building redundant indices.

Using a composite index is superior to several single-column indices in both performance and space. For example, it is common to build a composite index on LastName, FirstName and MiddleName in a employee table, and all the following queries can fully or partially use the index to speed things up:

/* fully use composite index */

```
SELECT * FROM employee WHERE LastName = 'Feng' and FirstName =
'Bryan' and MiddleName = 'Yongbing';

SELECT * FROM employee WHERE LastName = 'Feng' and FirstName =
'Bryan' and MiddleName between  'Yongbing' and 'Yongping';
```

/* use the leading two columns of composite index */

```
SELECT * FROM employee WHERE LastName = 'Feng' and FirstName =
'Bryan' and MiddleName like 'Y%';
```

/* In the next formal release the above example will fully use the composite index */

```
SELECT * FROM employee WHERE LastName = 'Feng' and FirstName
in ('Bryan', 'Brian');
```

/* partially use the first column of composite index */

```
SELECT * FROM employee WHERE LastName = 'Feng' and Middle-
Name like 'Yong%';

SELECT * FROM employee WHERE LastName like 'Feng%';

SELECT * FROM employee WHERE LastName = 'Feng';
```

However, the following query cannot utilize the index:

```
SELECT * FROM employee WHERE FirstName = 'Bryan';
```

It should be pointed out that a contains will not use the index.

```
SELECT * FROM employee WHERE LastName like '%Feng%';
```

## Q: How do composite indices in Butler SQL differ?

A: Before the current Butler SQL, there were several limitations of building and using a composite index:

- You could not build more than one index with the same primary column. If you did, only the first created index is valid, and the other indices are never utilized in query optimization.
- Query optimization did not fully take advantage of the composite index. The second example shown above only used the index as a single-column index, and did not use the index as a composite index on LastName and FirstName, so the speed was not fast enough.

Since earlier Butler SQL versions, these two limitations have been removed, and now Butler SQL can fully take advantage of the composite index. Since the limitation of only using the first index created with the same primary key is now removed, we can actually use multiple indices and take advantage of both.

For example, we can now build two composite indices, one on LastName and FirstName, and another on LastName and MiddleName, the first query shown above now can use both composite indices fully or as shown below.

```
/* fully use both composite indices */

    SELECT * FROM employee WHERE LastName = 'Feng' and FirstName =
'Bryan' and MiddleName = 'Yongbing';

/* fully use second composite index */

    SELECT * FROM employee WHERE LastName = 'Feng' and Middle-
Name = 'Yongbing';
```

Using two composite indices in this manner is actually more efficient and will deal with more variations in the where clause.

# Backup and Restore

This section deals with two statements supported by Butler SQL. The Backup statement is new to Butler SQL in the ODBC version. The Restore statement exists in earlier Butler SQL versions as well as the ODBC version. For information on other statements, see the SQL Reference Guide included in your Butler SQL package.

## BACKUP DATABASE Statement

This statement belongs to the Database-access statements.

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.

*Syntax*

`BACKUP DATABASE` <*database name*>

`[IN LOCATION` <*folder path*>]

`TO` <*folder path*>

`[WITH {JOURNAL, ACTIVITYLOG}]`

`[TIMEOUT` <*timeout seconds*>]

*Parameters*

*folder path* — is any valid Macintosh folder path string expression specified. The folder path should end with a colon. If the specified destination folder doesn't exist, it will be created by the server.

*timeout seconds* — the default timeout period is 0 which represents an infinite timeout period. Other timeout periods are specified as an integral expression in seconds.

*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 "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 has been specified and open transactions exist at the completion of the timeout period, the server will return an error. No client is permitted to begin a new transaction once this command has been issued (until the backup has completed or a timeout occurs).

Clients are held only when they have no outstanding transactions for this database, and 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 as well. This clause works by performing a detach of the current journal/log and then performs a move of the file from the current folder to the same destination folder as the database backup.

A timeout can be specified for this process so that connected clients are not held for too long waiting for others to complete existing transactions prior to being able to perform the backup. When a timeout occurs the client issuing the backup will receive an error and all held clients will be released to continue processing.

## RESTORE DATABASE Statement

There are two different forms of the Restore Database statement. The Restore All Databases statement will restore all transactions for every database in the specified journal. The Restore Database statement will restore all transactions that pertain to the database name specified in the restore statement.

This statement belongs to the Database-access statements.

There are two forms of the RESTORE DATABASE statement:

- 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 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 (“:”).<br><br>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 must be in the Public Databases folder in the Butler Preferences Folder and 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 DATABAS 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 applys to the database "SQL_Demo" located in the Public Databases folder.

## Maintaining your Database

Butler SQL gives you a number of statement that allow you to maintain your database in an online system. The commands BACKUP, ALTER, and COMPACT are all very powerful commands that allow you to maintain your database. Each of the statement prevent users from accessing certain resources and databases, as well as preventing new users from logging on.

### BACKUP

In maintaining your Butler SQL database, it is important to backup your data at regular intervals. The BACKUP statement can be an important part of your backup strategy. Your data is the most important part of your business. The BACKUP statement allows you to make copies of your database at regular intervals while Butler SQL is still online; and to backup your database before doing statements, such as an ALTER statement, in order to maintain copies before any schema changes.

### ALTER

The ALTER statement allows you to make changes to the schema of your database while your system is still online. This allows you to maintain business rules, add, modify, or drop columns, and change defaults without taking down the server.

### COMPACT

After executing an ALTER, major changes to the structure of your database have occurred and you should schedule time to do a COMPACT as soon as possible. The COMPACT statement cleans up the database and

deals with reclaiming any spaces that transactions have created. If you compact your database at certain intervals, depending on the frequency of transactions, you will get better performance.

**Note**

These commands should not be done at peak times.

In conclusion, you should be able to provide a stable and efficient environment for users by developing a strategy that combines BACKUP, ALTER and COMPACT.

## Using Char vs. Varchar columns

In general, most data processing data should be contained in CHAR columns within a Butler SQL table for maximum performance. However, there are several guidelines which can be applied to determine when to declare a column VARCHAR.

**Note**

Since Butler SQL treats VARCHAR and LONGCHAR internally as the same data type, it is important to note that this section applies to LONGCHAR also.

Make your column VARCHAR whenever one of the following applies:

- when the column is not to be indexed and is used as an informational field exceeding 128 characters in length,
- when the number of characters in the field varies wildly, or
- when the number of characters on average is less than half the size of the maximum column size needed.

There is one other reason to make columns VARCHAR. This has to do with the current behavior of the file system. If, by making your columns VARCHAR, you dramatically reduce the overall size of the rows, then

you will automatically gain speed in accessing multiple rows from a database file as more rows will be stored in one disk block. This optimization can only be determined by experimenting with the existing rows (performing SELECT statements on large rowsets) and will normally not be a major factor unless there are several large CHAR fields which could be switched to VARCHAR.

While defining a database schema in ButlerTools, you may notice that the program tells you the maximum row size. This is important because VARCHAR columns are packed into the rows using additional storage. Here is a simple chart you can use:

| | |
|---|---|
| rowsize < 32K | 4 bytes overhead per VARCHAR column |
| rowsize ≥ 32K | 8 bytes overhead per VARCHAR column |

This overhead is, of course, in addition to the actual length of the character string being stored.

For indexed columns, keep your CHAR or VARCHAR columns less than 160 characters to maintain maximum performance and reduce disk size. Index keys are always of a fixed size regardless of defining a column as CHAR or VARCHAR. This is necessary to guarantee the nature of the index's balanced tree mechanism. Butler SQL's index nodes are normally 512 bytes with a maximum key size of 168 bytes, or a maximum of 160 characters each (the minimum keys per node must be above 2). This means that if the character column you are indexing is larger than 160 (i.e. CHAR(160) or VARCHAR(152) to include the overhead bytes), Butler SQL is forced to increase it's index node size and perform multiple file block reads per node — thus leading to reduced performance.

If you must have large text columns, then when building the indexes, make the size of the keyed component less than the total size of the column. If you do so, Butler SQL will use the index to suggest possible matches but still apply the condition after rows are read and are being qualified for the SELECT statement. Utilizing this mechanism to keep keyed character columns below the sizes listed above will improve performance.

Finally, you should be aware that comparisons of VARCHAR columns (for conditions in a WHERE clause, or conditional statements in the procedural SQL) take considerably longer.

## Data-dictionary information

SQL provides information about the structure and organization of entities that it manipulates through a special data-manipulation statement. Using the DESCRIBE statement, the client application can determine the names and data types of the entities stored in the database and any implicit relationships among them. The client application can have SQL describe

- the supported DBMS brands on a particular host system,
- available databases for a particular brand,
- currently open DBMS brands and databases,
- the tables in a database,
- the columns in a table,

Butler SQL includes two additional data dictionary information statements. They describe

- the indexes associated with a table, and
- the keys of an index.

The DESCRIBE statement generates a rowset containing the requested definition in the same way as the SELECT statement creates rowsets.

## Unique Record Identifiers

In the earlier days of FoxPro and dBase databases, a user could have an ID table. Whenever the user needed a unique ID, he would LOCK the table, grab the next id, increase the value, then UNLOCK the table when he was

done. In a multiple-user system, this insured each record got a unique ID assigned to it. In SQL, however, you cannot LOCK a table. To combat this, SQL provides a rowid.

### Rowid

A new pseudo-field is now returned by Butler SQL queries called "rowid". Rowid is only guaranteed to be unique and valid within a session. Although you can lock a table in SQL, it is better to just lock the row of the table that tracks the unique ID for a given table.

To lock a table, you would use

```
OPEN TABLE "tableName" for EXCLUSIVE UPDATE;
```

To free it, you would close it

```
CLOSE TABLE "tableName";
```

### Important

Do not use Butler SQL's rowid columns as the foreign key of another table. These columns are "psuedo" columns and **are not** stored within the row itself; therefore, normal join processing, etc. should not be performed on them.

### Unique Indentifier

The "rowid" column returned by select statements is a unique identifier for rows. It is available to ensure that you have a unique ID to use when performing a subsequent update or delete. Rowids are sequential numbers which are assigned in arrival sequence of rows. Unlike some other DBMS's, the rowids are based on physical table rows, not cursor specific ordering or transaction specific values; and rowids are never modified once assigned to a row. Butler SQL's rowids are guaranteed to be unique and accurate within one client's session.

Rowids are used internally in Butler as the “address” of the row within the physical table. Rowids are always increasing sequence numbers but cannot be used in the same manner as Oracle's SEQUENCE. Since rowids are table addresses, SELECT statements cannot be guaranteed to return rows through a cursor in rowid sequence nor will a specific rowid exist if the associated row is deleted.

**Note**

Butler SQL, at the moment, does not reuse rowids, but there is no guarantee against this remaining a property of rowids in future versions.

Butler SQL's rowids are not maintained across actions like database rebuild (table export, table drop, table create, table import) or database recovery processing. Opening Butler SQL following a server crash where uncommitted updates were being done causes Butler SQL to auto-roll-back these transactions for DBMS integrity.

You can tell Butler to hide rowid columns from DESCRIBE COLUMN and SELECT * processing if you want. This can be done using the Butler SQL specific SQL statement "$SetRuntime(showRowIDs, $false);"

In the future, a feature very similar to Oracle's SEQUENCE will be implemented to allow for unique ID generation; however, in the meantime, it is preferable that you create a special purpose table (called “catalog” below) and declare a procedure as follows:

```
CREATE TABLE “catalog” (seq_name char(10), seq_value INTEGER);

/* you need a record to exist for this to work */

INSERT into catalog (seq_name,seq_value) VALUES ("mySequence-
Name",1001);COMMIT;
```

which will contain rows for each "CATALOG" you require referenced by seq_name

```
PROCEDURE NextSequenceValue(aName) RETURNS INTEGER;

ARGUMENT VARCHAR aName;
```

```
        {
            INTEGER i,j retValue = -1;
/*return -1 to indicate we can't get the catalog reference*/
            errorctl 1;                    /*we'll take care of errors*/
/* retry 10 times to get the row… */
            for ( i= 0; i<10; i++) {
                select seq_value from catalog where seq_name = aName for update;
                fetch next;
                err = $sqlcode;
                if (err == 0)  break;
                for (j = 0; j< 1000; j++) ;
/* wait awhile */
                }
            errorctl 0;
/* returns error control to Butler SQL*/
            if (err == 0) {
                retValue = ->updateCount;
/* this is the number we want, next increase and store */
                update catalog set updateCount = updateCount + 1 where current of
    $cursor; commit;
                }
            return retValue;
        } END PROCEDURE NextSequenceValue;
```

now by executing

```
PRINT NextSequenceValue("mySequenceName");
```

the next unique value will be returned.

**Note**

There is an iterative loop braced by ERRORCTL processing to allow for deadlock conflict resolution. Without this loop, if the sequence table were being referenced and updated by two people at the same time, you would have a record lock issued (ceLOCK) and the procedure would fail.

## Other Uses for Unique Record Identifiers

```
/* Get a unique number for a table */

        integer num;
/* declare a variable */

/* use procedure NextSequenceValue to get a number and put into the
variable */

        call NextSequenceValue("mySequenceName") returning num;
```

OR

```
        num = NextSequenceValue("mySequenceName");
/* now use the value to update a column in the database */

        if num != -1 {
            update myTable set ID = num where rowid = 123;
        }
        else call MyErrorHandler();  /* optional */
/* use the procedure directly in the update/insert statement */
```

```
update myTable set ID = NextSequenceValue("mySequenceName") where
rowid = 123;
```

OR

```
insert into invoice (invoiceNum, Morecolumns...)

  VALUES(NextSequenceValue("mySequenceName"), othervalues...);
```

There are pro's and con's for each of the calling methods and circumstances dictate the best approach at the time.

# Chapter 7 • SQL Procedures

This section describes SQL procedures in Butler SQL. A SQL procedure can consist of any SQL command, operator, or variable, and can be used in various ways.

There are four different types of procedures, the first three are stored at the server while the fourth is sent from the client. The differences between these four types are when they are loaded and disposed of, and whether or not they are shared.

The four procedure types are:

- msad$procedures —a procedure file that is automatically loaded and parsed by Butler SQL at launch time, is shared by all users, and exists until Butler is quit;
- userName$procedures — automatically loaded when the first user with that name connects, shared by all users with userName, exists until all users with that name log off, and Butler SQL needs to recover the memory;
- explicitly loaded for client using execute file command — loaded on demand, shared, exists until client disconnects; and
- sent as SQL from client — only needs to be sent once, not shared, exists until client disconnects.

## General Rules to Follow

There are several basic principles you need to keep in mind in order to get the maximum benefit out of the procedures you write. Procedures can be used to handle complex SQL operations, minimizing network traffic, and to separate SQL routines from the source code of the client application. This also allows the same SQL procedures to be used by different front ends, thus eliminating the need to duplicate and maintain multiple copies

of the SQL procedures. Also, since the procedure is already parsed, it will execute faster than if you were to send the same statement block to the server repeatedly.

These are a number of rules to follow when writing stored procedures.

1. Open or close a database within a procedure with caution. If you have already opened the database and then call a procedure that attempts to open it a second time, it will fail. Continually opening and closing the database will affect performance. Explicitly close tables if no longer required.
2. Routines can be general purpose by both returning results to another procedure and printing the data to the output queue for the client. (Make sure you do the print statements prior to the return statement in these cases). THIS IS NOT APPLICABLE TO ODBC.
3. Procedures are self contained. Procedures in a file cannot reference procedures or variables contained in other procedure files.
4. An execute immediate or an execute from cannot declare a procedure within another procedure.
5. An execute file will be considered to have a scope outside of the existing execution and can be used anywhere in the code.
6. A variable declared in an execute immediate or execute from is only valid during the life of that execute unless it is in the user frame.
7. A procedure declared within an execute from or execute immediate will be allowed in the user frame only.
8. During the execute of the msad, an execute immediate or execute from will be allowed, but it is assumed that it declares no variables or procedures which are referenced after the contents of the execute statement has been run.
9. An execute from statement in the outer block or user frame can

declare procedures or variables that are used right after the statement.

## When to use a procedure

You should consider using a procedure if the statement blocks are large or complex and need to be sent more than once in a session. Also, as some client applications only allow single table querying, a procedure can be used to do multiple table joins returning the data to the client as if the data came from a single table.

## Procedure types

Each of the different types of procedures can be used by the same client; although procedures of one kind may or may not be able to interact with procedures of another kind.

### msad$procedures

The msad$procedures are the place to store procedures which are necessary for all clients and databases. Runtime settings such as date or formatting options can be globally set here. Procedures stored here can be called from procedures used in the other three methods. The msads should be specific to the sever and not contain procedures for specific applications.

CONS: difficult to maintain - if more than one application uses msads, these procedures need to be merged.

### userName$procedures

UserName$procedures are the place to store procedures that are needed for a particular application. In this case, each client should connect with the same name, possibly using a psuedonym to indicate the different users in the log on and in the user window. The first user in, takes a performance hit while the procedures are loaded and parsed, thereafter, each new client has the same benefits as the msad procedures. Procedures declared here with the same name and arguments as an msad will take precedence over the msad procedure.

PROS: easy to maintain at the server.

CONS: first user has performance penalty.

### ExecutefileName

ExecutefileName will execute the contents of a file stored in the procedures folder. It can consist of raw SQL as well as procedures. These are best used to execute maintenance procedures at the server or other special tasks on a periodic basis.

PROS: easily maintained, can contain what amounts to batch processing commands.

### SQL from clients

A procedure can also be sent directly from the client and used throughout the current session. Procedures should be written and debugged in client application, such as ButlerClient, and then moved to a file for use by the other three types. These procedures are not shared.

PROS: procedures can be maintained at the client end without disturbing the server,

CONS: increases network traffic, more memory usage at server if more than one user uses the same procedure

## Procedure Syntax

Regardless of type, the syntax for all procedures is the same. Also, as there is no current way to prototype functions, the order in which they are declared is important: procedures referenced by other procedures must be declared first.

```
declare procedure myProc ([parms1],[parms2],[ ...])
[returns dataTypeList];
[argument dataTypeList];
{
```

For DAL statements;

```
    [return dataTypeList];
} end procedure myProc;
```

## "Redeclaration" of Procedures

Procedures can be declared for a second time, but only if they possess the same number and kind of arguments and return values. A procedure "redeclaration" is effective only within its scope.

**Note**

Recursive procedures are not supported.

## How to use a Procedure

How one uses a procedure depends on how the procedure is going to return data. Procedures can return more than one value and send data to the client. Procedures can be called in several ways. The following example retreives the next unique number to be used.

### Example 1:

```
/* get a unique NUMBER for a table */

    integer num;

/*use procedure NEXTNUMBER to get a number */

        call nextNumber('theNextID') returning num;
```

OR

```
       num = NEXTNUMBER('theNextID');

/* now use the value to update a column in the database */

       update myTable set ID = num where rowid = 123;
```

OR

```
/* use the procedure directly in the update statement */

    update myTable set ID = NEXTNUMBER('theNextID') where rowid =
    123;

    declare procedure NEXTNUMBER(ColName)

    returns integer;

    argument char ColName;

    {

      integer id;

/* get the correct record from the database */

      select theNEXTNUMBER from control where name = ColName into
    c1 for update;

       fetch;

       if $sqlcode <> $sqlnotfound {

             id = theNEXTNUMBER;

/* store the value in a variable */

             update control set theNEXTNUMBER = id + 1 where cur-
    rent of c1;

/* increment the variable*/

           return id;

/* send it back to the calling procedure */

       }

       else

           return 0;

/* you want to make sure this is handled at the calling procedure
*/

    } end procedure NEXTNUMBER;
```

### Example 2:

```
/* Do a join and return the data to the client */

    Print GetCustomerBal('1123');

/* this method allows the proc to be used by SQL and ODBC */

/* the print statement is not required when called from ODBC */

    declare procedure GetCustomerBal(num)

    returns money;

    argument integer num;

    {

/* calculates the balance from the invoice file */

      select sum(invoiceTot- AmountPaid) tot from invoices where
    billed = $true and cust_num = :num;

        fetch;

       if $sqlcode <> $sqlnotfound {

             return tot;

       }

       else

            return 0;

    }end procedure GetCustomerBal;
```

### Example 3:

```
/* same as example 2 but for SQL only */

    GetCustomerBal2('1123');

    declare procedure GetCustomerBal2(num)

    returns money;

    argument integer num;
```

```
    {
/* calculates the balance from the invoice file */

      select sum(invoiceTot- AmountPaid) tot from invoices where
    billed = $true and cust_num = :num;

        fetch;

       if $sqlcode <> $sqlnotfound {

             print tot;

/* print statements are not valid ODBC commands*/

       }

       else

            print 0;

    }end procedure GetCustomerBal2;
```

## Common Mistakes

Some common mistakes which occur when using procedures:

- Not testing the procedures using ButlerClient before implementing them in your application,
- "Redeclaring" a procedure with different parameters,
- Recursion (i.e. a procedure calling itself),
- Printing values in a procedure instead of returning them (especially when using ODBC),
- Lots of SQL in a few procedures instead of a little SQL in many procedures, and
- Incorrectly using implicit type casts (i.e. don't quote values for numeric column types in selects).

# Chapter 8 • Performance and optimization

This chapter describes some techniques you can use to increase the performance of your Butler SQL server. The information here goes beyond the basics of creating indices. It describes in detail how Butler's query optimizer works and details some of other mechanisms you can use to improve performance.

## Query Optimizer

In general, Butler SQL will provide an optimal solution to the query without your intervention. You need only to understand the query optimizer to improve performance of a specific SELECT or query specification (those defined within searched UPDATE, INSERT or DELETE statements).

By using a runtime flag and any SQL query tool, a database administrator can examine and modify the access plan for any query specification. Output from the query optimizer can be directed either to the current server activity log or to the client's output queue (for return to the client front-end as part of the data retrieved). By then adding/editing indexes or modifying the query specification, the administrator can affect fast query execution.

### Terminology for this section

The following terminology is used throughout this chapter.

**Table 13: Terminology**

| | |
|---|---|
| QO | Query optimizer |
| Query specification | A query specification is used by the SELECT, UPDATE, INSERT and DELETE statements to define the database tables, rows and columns to be referenced. |
| Projection | The process of extracting the data from relational tables into a result rowset as a part of the query specification. |
| 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. |
| Ranger | The specific access plan for a table in the query specification. There is one ranger for each table in the query specification. |

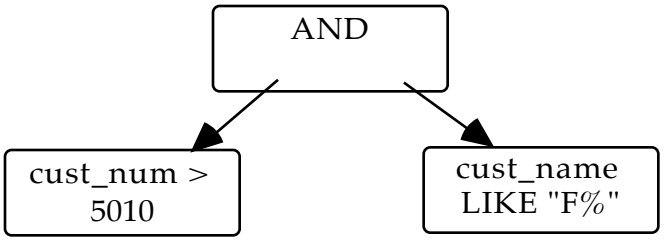
## Query Optimizer Details

When Butler SQL receives a SQL statement, its SQL parser must first deduce the details of what the request is by converting the text sent (the SQL) into a usable form which can be executed. As Butler SQL parses a SELECT statement, it generates several lists of information, including the conditions within the WHERE clause.

### WHERE clause decision tree

As the where clause is parsed, a decision tree which defines the grouped conditions and the AND/OR manner in which the conditions relate is generated.

For example, the statement 'SELECT * FROM customer WHERE cust_num > 5010 AND cust_name LIKE "F%';' will be interpreted by Butler SQL's parser generates a decision tree as follows:




The decision tree contains a terminal leaf for each condition in the WHERE clause, and a branch node for each AND or OR condition. The decision tree is naturally weighted to the left because conditions in WHERE clauses are executed in declared order unless the query optimizer modifies the conditions to improve performance.

**Important**

By adding parentheses, a user can modify the construction of the branches within the tree. By adding parentheses to change the order of evaluation, a query may be interpreted differently by the optimizer.

### Resolving column references

The next step in the optimization process is to evaluate and resolve column references within the WHERE conditions. The most efficient form of a column reference is an unqualified column reference, where the column is uniquely identified within the tables defined for this query specification.

By defining a column alias in the select list, the user can eliminate the need for table qualified table references in the WHERE clause and improve optimization speed (and execution speed, in some cases). Using a table alias with the actual column name is the least efficient manner of referencing columns.

### Join conditions

After column references have been resolved, the optimizer scans the decision tree for join conditions which are recognized as containing a column reference on either side of a comparison operator. Join definitions describe the access paths used to propagate from one table to another within the execution of the database access plan. The QO uses the join definitions to generate table rangers, which are used to implement each table's access plan for the query specification.

### Unrecognized conditions

Some types of conditions cannot be handled by Butler's query optimizer and are therefore eliminated as "unusable" leaves within the decision tree. The optimizer looks for conditions which meet the following criteria:

(Col cmp K) or (K cmp Col), where

- Col is a direct column reference (objname references are not always optimized),
- cmp is one of >, <, >=, <=, =, LIKE, BETWEEN, IS NULL, IN (value list), and
- K is a query constant.

Only direct column references (Col) are fully supported by the optimizer; however, some conditions are available. See the "Bias table" on page 152 for details about the specific optimizations on conditions involving objname column references.

A query constant (K) is any value which will not change during the execution of the query. Any variable, parameter, and literal or numeric value can be used and optimized, but are treated as a constant, the value of which is known when the query specification is presented. For example, if the SQL "SELECT * FROM X WHERE colref = :Y;" is submitted with Y declared as a variable or parameter, then the value of Y will be used when the SQL is executed, and considered constant for that SELECT statement. Changes to :Y during subsequent FETCH operations will have no effect on the projection from the database.

### Combined conditions

During the course of scanning the decision tree, some conditions are combined into more efficient forms. This feature allows for query tools to generate generic SQL, which is more fully optimized by Butler SQL. These combined conditions are:

- **value lists** are combined from ORed conditions,
- **between conditions** are composed from two ">=" and "<=" conditions, and
- **composite keys** of an index are identified.

### Condition bias values

After the previous steps have been taken, each condition (leaf of the decision tree) is assigned a bias based on the kind of comparison and type of column.

### Conditional operators

In the following condition bias table, the following conditional operators are used.

- non-equal comparison — theta-conditions (<, ≤, >, ≥, <>)
- equality comparison — equa-condition (=)
- pattern-matching like conditions
  - begins with (LIKE "X%")
  - contains (LIKE "%X%")
  - ends with (LIKE "%X")

Condition bias values are displayed in the following table.

**Table 14: Bias table**

| bias | column | definition |
|---|---|---|
| 0 | N | no bias |
| 10 | N | non-indexed, non-equal comparison (<, ≤, >, ≥, <>) |
| *20 | N | contains pattern match (column like “%X%”) |
| *25 | N | non-indexed, equality comparison |
| all biases above 30 are optimized, all conditions with 30 or less are ignored by the query optimizer. | | |
| 34 | I | ORed index column ranges |
| 40 | R | rowid, non-equal comparison (<, ≤, >, ≥, <>) |
| 50 | I | indexed column, non-equal comparison (<, ≤, >, ≥, <>) |
| *54 | U | non-equal comparison (<, ≤, >, ≥, <>) |
| 60 | I | begins with pattern (LIKE “X%”) |
| *64 | U | begins with pattern (LIKE “X%”) |
| 70 | I | range (BETWEEN a AND b) |
| 74 | CI | non-equal comparison (<, ≤, >, ≥, <>) |
| *76 | U | range (BETWEEN a AND b) |
| *78 | CU | non-equal comparison (<, ≤, >, ≥, <>) |
| 80 | I | value list (IN (a, b, c…)) |
| 84 | I | equality comparison |
| *90 | CU | value list (IN (a, b, c…)) |
| *94 | CU | equality conditions |
| *98 | CU | equality comparison |

**Table 14: Bias table**

| bias | column | definition |
|---|---|---|
| 100 | R | rowid equality |
| * = bias is new or changed from earlier Butler SQL versions | | |
| N = non-indexed column | | |
| I = index column | | |
| R = rowid pseudo column | | |
| U = uniquely keyed index column | | |
| CI = composite index key | | |
| CU = uniquely keyed composite index key | | |

Composite index keys are utilized when the WHERE clause conditions are ANDed conditions on different columns which are part of the same index key. For example, if an index exists on {LastName, FirstName}, it is considered a composite index, and WHERE clauses which AND a lastName equality condition with a FirstName condition (other than inequality) will yield a condition bias type of CI. Further, if the index is keyed uniquely, then the bias type will be CU. The very high biasing of these combined conditions will normally yield a very fast access plan.

The higher the bias value, the more weight placed on that condition. Butler SQL uses the condition bias to determine which condition will yield the least number of projected rows for each table. All conditions with a bias of more than 30 are considered important to the access plan; whereas, other conditions are applied to the candidate rows after the result row is projected and calculated columns have been evaluated.

## Optimal primary table and condition(s)

The next step in the optimization process is only performed when multiple tables are included in the FROM clause, indicating a joined projection from the database. In such cases, the objective is to determine the optimal table from which to begin the join process. All other tables involved in this query specification are read after the primary table's row(s) are read.

Butler SQL utilizes a project and join strategy while performing the rowset generation. We read a primary table row and, using that row's column values, we read from all adjacent tables (those who are defined as directly joined to the primary table based on the join conditions within the WHERE clause). This process is repeated until all tables are read. By default, if one table cannot be read using the dominant table's current column data, then no selection is performed using this information, and the dominant table's next matching row is read.

In the case where no matching rows exist in a joined table, the join is considered null and the next row from the primary table is read (i.e. unless outer-joins are requested, Butler SQL will only include fully qualified rows in the resulting rowset).

## Outer-joins

Outer-joins are supported in Butler SQL in the form of a left outer-join or right outer-join. Full outer-joins are not supported. To specify a left outer join, use*= as the conditional operator. To specify a right outer join, specify =* as the conditional operator.

A left outer–join will yield a resulting rowset containing all qualified rows of table A joined to associated rows of table B where possible or a $null in all columns of the result from table B where a null join is detected. A right outer-join is completely symmetric with a left outer-join.

Determining the optimal primary table begins by defining an adjacency matrix of all tables contained within the query specification. This collects information about the number of comparisons needed to fully join all

table rows together, and contains a condition bias about the type of comparison. Further, if additional select/omit criteria are applied to a table, it is included also in the joining bias calculated within the adjacency matrix.

Finally, the adjacency matrix assigns a "distance" bias and "directional" bias on each table, indicating whether an index can be used during the join process. Non-indexed columns yield significantly poorer performance; therefore, this bias is weighted heavily towards index utilization.

Outer–join support in Butler SQL adds another qualification which will have significance to optimal performance. If a left outer join is performed, all directional biasing within the adjacency matrix is biased such that the left table will ALWAYS join towards the right table. For more information on ODBC outer-joins, see "ODBC Outer-joins" on page 61.

## Considering order by/group by sequencing

After the primary table is chosen and the optimal condition is determined from the condition biasing, the QO tries to determine whether an index sequential access could be used to avoid having to sort the projected results. If an index can be utilized, the order by condition is dropped, other index biased conditions are combined using selection sets, and an ordered ranger object is allocated to handle the actual data fetching.

## Final stage

If no ranger object is allocated from the previous step, then an appropriate ranger is allocated in this stage of the QO. There are several types of rangers which may be allocated based on the condition chosen as optimal. The following table displays the rangers which may be allocated.

**Table 15: Rangers**

| Ranger | Description |
|---|---|
| kDefaultRanger | Reads all rows from a table. |
| kRowIDRanger | Reads a range of rows based on arrival (insert) sequence. |
| kIndexRanger | Reads an index-sequential range of rows. |
| kRowsetRanger | Ranges a pre-extracted set of rows (used for describe statements). |
| kNoIndexRanger | Reads a range of rows and confirms that a specific field is within range. |
| kValueListRanger | Reads a series of rows based on a list of values supplied by an IN |
| kAlikeRanger | Performs a partial keyed read of a range of values matching a pattern. |
| kOrderedRanger | Reads a range of records based on a predefined selection set. |

If the queryOptFlags runtime is used, Butler SQL can be forced to generate a special type of ranger which will use the resulting selection set when defining the ranger object. The queryOptFlags is a special purpose runtime setting which can be varied for very specific circumstances. See “Selection sets” on page 157 for more details.

### Selection sets

Butler SQL has the ability to perform selection set projections of rowids from each table. Each set generated for the same table can be UNIONed for ORed conditions and INTERSECTED for ANDed conditions. Selection sets of rowids from multiple tables are used to generate projections representing the join definitions, but this facility is not controlled or reported to the user since it is not adjustable.

You can force Butler SQL to always generate and perform UNION/INTERSECTION operations on the selections by setting a special bit flags field (see "queryOptFlags" on page 159). This process interrupts the normal process of choosing the most optimal table condition, assigning a ranger and then adding a selection set to that ranger. Basically, the queryOptFlag allows you to always generate selection sets for every indexed condition, UNION or INTERSECT these selection sets to yield the absolute selection set which represents all indexed conditions and then have a selection set ranger assigned in place of the indexed ranger.

Why would you want to override the default optimizer's behavior? You would want to redirect the query optimizer in cases where it's rule based decisions yield an inefficient access plan. By reviewing the access plan of a specific SQL statement, the user can have ultimate control on the efficiency of the queries being made and thus improve overall performance.

Selection sets are rowid sets which can be allocated in application memory or multifinder temporary memory. By default sets are built in temporary memory because they typically have short life spans and other SQL clients will be less affected by out-of-memory conditions this way.

**Note**

Butler SQL is smart enough to switch automatically from one memory zone to another when the default one is exhausted and sets may actually exist in two separate zones. Naturally, the more RAM available to the server, the less work needed within this context is needed.

### Query Optimer Runtime settings

The follow are details on using the query optimizer logging to examine an access plan.

### logQueryOpt

A runtime used to generate an explanation of the access plan for queries submitted by a client.

$SetRuntime(logQueryOpt, X) where X is a selector code:

- bit 0 - set = send output to activity log; clear = don't generate an access plan explanation,
- bit 1 - set = send output to client output queue; clear = don't generate an access plan explanation.

So,

```
$SetRuntime(logQueryOpt, 0);
```

- disables access plan explanations

```
$SetRuntime(logQueryOpt, 1);
```

- sends explanations to the server's activity log

```
$SetRuntime(logQueryOpt, 2);
```

- sends explanations to the client's output queue

```
$SetRuntime(logQueryOpt, 3);
```

- sends explanations to client & activity log

**Note**

The explanation is represented by a rowset of VARCHAR(255) columns that are sent automatically to the appropriate location without disruption to the query's execution. If you direct to the client's output queue, the QO

explanation will precede all data returned as a result of the query's execution. Use ButlerClient or a similar client application when directing explanations to the client's output queue. Direct output to the activity log, when reviewing access plans for existing client applications whose control of the output queue, is limited to expected data results.

**queryOptFlags**

A runtime used to modify the default rules used by the query optimizer;

$SetRuntime(queryOptFlags, X) where X is a bit mask with the following bits used:

- bit 0 - set = use selection sets when possible [set = default]; clear = never use selection sets,
- bit 1 - set = always use selection sets for all indexed conditions; clear = choose optimal ranger and then apply other selection set criteria [clear = default]. bit 0 must be set for this action to occur.

## Special Conditions

This section describes how the query optimer handles, or sometimes doesn't handle, specific types of queries in your SQL.

### Self-joins

Self-joins can be defined as table selections, where the table on either side of the join equation are referred to the same physical table. For example, 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.

In Butler SQL, self-joins are supported and handled in a proper manner in the query optimizer. That is, self-joins will not impose any more performance impact than any other join between two different tables.

The use of multiple table aliases to the same physical table is also a good optimization for sub-queries. At times, multiple sub-queries on a secondary table can be revised into joins to the secondary table with additional conditions added which replace the sub-query. Where multiple sub-queries on the same secondary table are necessary, the optimization will require the use of a second reference to the same secondary table in the FROM clause. Butler SQL will properly handle optimization of this type of query specification.

## Multiple indices

Multiple indices on the same table beginning with the same first column will confuse Butler SQL and will probably yield less desirable effects. Butler SQL will always choose to reference a column's index based on the first index created with this column as it's primary ordering. In general, create only indices with the primary keyed columns uniquely named. That is, if a table 1 has columns a, b, c, d then do not create more than one index with column a as a primary key element. Creating indexes {a, b}, {a, c} and {a, d} will be inefficient as only the first index created with column a as its primary element will be used by the query optimizer.

## Cost-based conditional biasing

In a truly cost based optimizer, a histogram of data content is maintained and used when making decisions about access plans. This form of optimizer will normally yield better results as it can make decisions about the smallest projects from which to draw the access plan.

Butler SQL does not allow for the gathering or analysis of table/index content prior to making access plans. This would allow Butler SQL to make decisions based on the uniqueness (and the size of a selection) for each condition it is considering.

Butler SQL does, however, use table sizes during the process of evaluating the optimal conditions, and can use this information when making decisions about the combination of selection sets and index rangers. Moreover, table sizes are also used when making decisions about the optimal table.

### Subqueries

Subqueries are truly non-optimal database projections at the moment. During the execution of a query which utilizes a subquery, the subquery will be regenerated for each row retrieved from the outer database projection. This means that if 100 rows are read as part of the outer query, 100 separate query specs are executed to properly qualify the rows despite the fact that most subqueries do not qualify rows by columns from the outer query.

### Contains and ends

Contains or ends-with type conditions are not optimized. Regardless of whether the columns are indexed or not, the server will read the entire record, extract the appropriate column, and qualify the row for selection based on the LIKE condition's results.

### Value lists in composite keys

If a table is indexed on columns {a, b, c} and the WHERE clause exists as "WHERE a = xxx AND b in [x, y, z]", Butler SQL will not optimize and take advantage of the composite keys (namely, {xxx, x}, {xxx, y}, {xxx, z}); but rather, Butler SQL will use the single a = xxx condition to select a range of records and generate a selection set of all elements matching b = x, y or z. The selection set is then assigned into the index ranger on the a=xxx condition, and all records within the range which exist in the selection set will be read. This may yield much higher RAM requirements (to fit the selection set of all rows where b = x, y or z).

One "work-around" is to force a selection set ranger to be used so that the selection set a = xxx and the selection set b = x, y or z are intersected prior to any row projections (thus using more ram for a much shorter period of time). See "Query Optimer Runtime settings" on page 158 for more information on rangers.

### Conditions with expressions

Butler SQL presently ignores any condition which includes the evaluation of an expression. This is important because one method of avoiding a condition from optimization is to simply add zero (or concatenate an empty

string) to the column condition, and it will be eliminated from the QO stage. “Denormalized” table columns which require the use of Substring processing in conditions are therefore not optimized by Butler SQL. The solution is to normalize your database.

### Objname references

Objname references in a where clause will not always be resolved during the optimization phase because the reference may be another database object or a parameter to a procedure.

### Negation (NOT)

Currently, the NOT operator will cause the negated condition clause to be biased to zero and will eliminate it from the process of further optimizations.

### IS NULL

Using the IS NULL condition on an indexed field is optimized and biased as an equality index condition. Using a condition with IS NOT NULL is currently not optimized.

### Using the query optimizer logging capability

Using the Butler SQL system function, you can enable or disable the optimization logging capability. Use the selector code “logQueryOpt” to indicate that it is this feature you would like to enable/disable. Use the following values as a second parameter:

| Values | Description |
|---|---|
| 0 | disable logging |
| 1 | enable logging to the server's activity log |
| 2 | enable logging to the client's output |
| 3 | enable logging to activity log and client's output queue |

For example, `"$SetRuntime(logQueryOpt, 2);"` will enable query optimizer logging and return the log information to the client's window in the form of a single VARCHAR column rowset. Calling `"$SetRuntime(logQueryOpt, 1);"` inside the msad$procedures file will allow the DBA to review the activity log from the server and see all QO logging from all active clients. Sending `"$SetRuntime(logQueryOpt, 0);"` will disable all QO output.

Enable logging to the server's activity log in circumstances where you need to review optimizations in a program not capable of receiving the output directly, or when multiple clients need to be reviewed simultaneously under load/stress situations.

Enable logging to the client's output when using applications like ButlerClient, SQL Terminal, and ClearAccess. Normally, the DBA will use ButlerClient to submit the SQL query under review and can then immediately receive the QO output.

**Note**

By using ButlerClient and setting QO logging to clients' output you can isolate other possible speed impediments which may affect the results of the SQL Execute. Further, if you send only the SELECT statement (not the PRINTALL) you need not receive the actual data being selected, only the QO output.

### Interpreting the log output

Interpreting the QO log is a matter of reviewing the biasing assigned to each query condition and analyzing the results of the QO by reviewing what type of ranger has been assigned. Further, specific join operations will yield results which are not optimal. By reviewing the choice of how to perform joins (choosing the primary table and the order of chained joins) the DBA will understand how to optimize the information for better performance.

### Influencing the query optimizer

Changing the order and other conditions within a where clause will allow the user to greatly influence the outcome of a query optimization. This process is normally called 'hinting'. Specifically, by influencing the bias of conditions, the query optimizer will choose other optimal access paths, different primary tables, thus yielding the result desired.

For example, by making an equality condition like "cust_num = :theCustomer" into "cust_num = :theCustomer + 0" (assuming that cust_num is a uniquely keyed indexed column) the bias will be changed from 94 to 0 thus eliminating this condition from influencing the outcome of the QO process.

Under what circumstances would you need to influence the QO? Hinting is mainly used in Butler SQL to change the primary table chosen during multiple table select statements. Because Butler SQL chooses the primary table based on the assumption that the table with the most conditions will yield the smallest database projection (table reads), the server may choose a table with many rows from which to start the projection.

## Default Tables

The "Default Tables" file is a text file in the'Butler Preferences Folder' with each line representing a different table. The format of every line of the default table file is "database!tablename" [no quotes] and may have other parameters added after the tablename with comma delimiters. These will include a keyword "NOCACHE" or "CACHE". If the keyword doesn't exist, it defaults to "NOCACHE" (see the follow section on how caching works).

This feature is an application level setting and will therefore affect all users.

| | |
|---|---|
| *Syntax*: | databaseName!tableName,[cache \| noCache] |
| e.g.: | dal_demo!customer, cache |

```
dal_demo!orders, noCache
dal_demo!staff
```

There are two main benefits to be gained by this feature:

- reduce the number of table open journal entries to reduce journal size, and
- increase OPEN TABLE speeds.

Tables opened in this manner will NOT be defined as OPEN, but prepared for open. While Butler SQL has opened the tables and logged the open into the Butler SQL journal for transaction processing, they are not usable nor displayed by Butler SQL as "open" until implicitly opened using some DML, or explicitly opened with an OPEN TABLE statement. Therefore, DDL statements like ALTER, DROP, and CREATE should be unaffected by this feature. BACKUP is also be unaffected by these prepared tables. COMPACT represents a problem because Butler SQL expects all Butler SQL table references to be closed prior to the compact.

Two runtime commands have been implemented to allow manipulation of the default tables function: one to unload all default tables, and one to reload default tables.

The runtimes to load or unload the default tables are:

- `$setruntime(loaddefaulttables, true);  /* loads the default tables */`
- `$setruntime(loaddefaulttables, false); /* unloads the default tables */`

## Index Caching

Index caching improves performance of inserts, updates and deletes when processing many records in a table with indices. Single record handling will not benefit from caching.

In addition to using the default tables file to pre-open tables and enable/disable caching for a given database and table, the following runtimes can be used either in conjunction with the default tables file or independently. These settings are application wide, so all users of Butler SQL are affected.

The following runtimes turn index caching on or off for a given table. The current status of the table caching can be examined by using the $getruntime function. All indices associated with the table are affected by these settings.

- `$setruntime(enableTableIndexCaching, “tableName”); /* turn caching on for this table */`
- `$setruntime(disableTableIndexCaching, “tableName”); /* turn caching off for this table */`
- `print $getruntime(tableIndexCacheStatus, “tableName”); /* gets the status of caching for this table */`

The following runtimes turn caching on or off for any given index. The current status of the table caching can be examined by using the $getruntime variable. Only the specified index is affected.

- `$setruntime(enableIndexCaching, “indexName”); /* turn on caching for this index */`
- `$setruntime(disableIndexCaching, “indexName”); /* turn off caching for this index */`
- `print $getruntime(indexCacheStatus, “indexName”); /* gets the status of caching for this index */`

The following runtimes force the index caching on or off for all tables as they are opened. There is no runtime to determine the status of 'forceTableIndexCaching'.

- `$setruntime(forceTableIndexCaching, true); /* force index caching on tables as they are opened */`
- `$setruntime(forceTableIndexCaching, false); /* if table is opened don't force index caching */`

Each index cache requires approximately 5K of Butler SQL's RAM when turned on. This memory is set free when caching is turned off; therefore, a table with 5 indexes will require 25K of RAM. Caches are in effect until the termination of Butler SQL or when explicitly turned off by any user.

## Alternatives to Subqueries

Subqueries contain queries in their where clause, providing values that determine part of the search criteria. In general, the outer select is called outer query while the inner select is called inner query. In Butler SQL, the inner query is evaluated for each candidate row in the outer query. When the inner query is not affected by the result of the outer query, it is often better to change the makeup of the subquery. The following examples illustrate subqueries that are not optimized by Butler. The next section describes how to change these examples to yield better performance.

Query 1: [Find the identifier number and name for all the customers who made orders before Jan. 1st, 1995.]

```
select cust_num, cust_name from customer
where cust_num in (select cust_nr from orders where ord_date < '01/01/
1995');
```

Query 2: [Find the identifier number and name for all the customers who made orders through the salesperson whose first name is Charles, and the order amount in one of the orders made through Charles is greater than $10000.00.]

```
select cust_num, cust_name from customer
where cust_num in
 (select cust_nr from orders
 where salesrep in (select rep_nr from staff where first_name = “Charles”)
 and ord_amount > 10000.00);
```

Query 3: [Find the identifier number and name for all the customers who made orders through both the salesperson whose first name is Charles and the salesperson whose first name is Diana.]

```
select cust_num, cust_name from customer
where cust_num in
 (select cust_nr from orders
   where salesrep in (select rep_nr from staff where first_name = “Charles”))
and
 cust_num in
 (select cust_nr from orders
   where salesrep in (select rep_nr from staff where first_name = “Diana”));
```

## Correlated subqueries

Correlated subqueries are those whose inner query search criteria depend on values from the tables identified in the outer queries. Based on the Butler SQL access plan for a subquery, executing a correlated subquery is better for performance than executing a non-correlated subquery for the same task, as additional search criteria can limit the inner query to a smaller range; therefore, you should write a correlated subquery whenever possible.

As a matter of fact, you can easily convert most non-correlated subqueries into correlated subqueries by replacing IN subqueries by EXISTS subqueries. However, the conversion must be done with care, because EXISTS subselect uses two-value logic while IN subselect uses three-value logic. The following are the correlated subqueries converted from the example subqueries given earlier.

Query 4: [equivelant to Query 1]

```
select cust_num, cust_name from customer
   where exists (select cust_nr from orders
    where ord_date < '01/01/1995' and cust_nr = customer.cust_num);
```

Query 5: [equivelant to Query 2]

```
select cust_num, cust_name from customer c
where exists
 (select cust_nr from orders o
   where cust_nr = c.cust_num
    and exists (select rep_nr from staff
     where rep_nr = o.salesrep and first_name = “Charles”)
```

```
 and ord_amount > 10000.00);
```

Query 6: [equivelant to Query 3]

```
select cust_num, cust_name from customer c
where exists
 (select cust_nr from orders o1
   where cust_nr = c.cust_num
    and exists (select rep_nr from staff
     where rep_nr = o1.salesrep and first_name = “Charles”))
and
 exists
  (select cust_nr from orders o2
   where cust_nr = c.cust_num
    and exists (select rep_nr from staff
     where rep_nr = o2.salesrep and first_name = “Diana”));
```

Query 7: [equivelant to Query 3, further rewrite Query 6]

```
select cust_num, cust_name from customer c
where exists
  (select cust_nr from orders o1
   where cust_nr = c.cust_num
    and exists (select rep_nr from staff
     where rep_nr = o1.salesrep and first_name = “Charles”)
    and exists (select rep_nr from staff
     where rep_nr = o2.salesrep and first_name = “Diana”));
```

## Joins

Even if one can achieve better performance by writing a correlated subquery, one is still stuck on the access order presented in the subquery: from outer query to inner query, and back to outer query; let alone parse the inner query every time the inner query is evaluated.

In most cases, join is another alternative to subquery by parsing the query only once, narrowing the search, and choosing the better access order. However, join may return duplicate rows if the join relation is one-to-many, where the DISTINCT keyword has to be added to the select list of the query in order to eliminate the duplication.

The following queries are converted from the subqueries given before.

Query 8: [equivelant to Query 1]

```
select distinct cust_num, cust_name from customer, orders
where cust_num = cust_nr and ord_date < '01/01/1995';
```

Query 9: [partially equivelant to Query 2, because of duplication]

```
select cust_num, cust_name from customer c, orders o, staff s
where c.cust_num = o.cust_nr and o.ord_amount > 10000.00 and
  o.salesrep = s.rep_nr and s.first_name = “Charles”;
```

Query 10: [equivelant to Query 2]

```
select distinct cust_num, cust_name from customer c, orders o, staff s
where c.cust_num = o.cust_nr and o.ord_amount > 10000.00 and
  o.salesrep = s.rep_nr and s.first_name = “Charles”;
```

Query 11: [equivelant to Query 3]

```
select distinct cust_num, cust_name
from customer c, orders o1, orders o2, staff s1, staff s2
where c.cust_num = o1.cust_nr and c.cust_num = o2.cust_nr and
  o1.salesrep = s1.rep_nr and s1.first_name = “Charles” and
  o2.salesrep = s2.rep_nr and s2.first_name = “Diana”;
```

It is also possible to join a table with itself; this is called a self-join. A self-join compares values within a column of a single table. For example, you can use a self-join to find out which authors in Oakland, California have exactly the same zip code.

For more information, see “Self-joins” on page 159.

## Stored procedures for performance

The use of stored procedures has several advantages affecting the efficient use of your server and its performance. Naturally, stored procedures are only applicable to applications which repeat the same SQL requests many

times. For query and analysis tools which execute single queries and move on to other requests, stored procedures have little impact on performance. Also, stored procedures which could negatively affect performance will be discussed in this section.

Procedures allow for the collection of business and database specific actions which are to be used in a common and uniform manner. Moreover, stored procedure files are shared between all users; therefore, lower are the memory requirements placed on the server.

Lower RAM "footprint" help the server to perform better as memory is available for other purposes including caching and query optimizer sets. Client applications which use stored procedures will have less SQL that is sent across the LAN to fulfill the request. Reduced network traffic will lead to better network performance. Using stored procedures forces the server to parse the procedure file in advance and avoids subsequent parse phases for each execution of the desired action.

Stored procedures are most effective when used to increase concurrent processing between the client and the server. While the server is processing a stored procedure, the client will be able to process returned rows.

Some stored procedures will actually inhibit server performance, but may still be necessary. For example, procedure calls executed as part of an expression on each column will be very expensive. Alternately, see if there is an aggregation function which can be used instead to perform calculations on each row, and perform a final expression on the aggregation when necessary. Further, avoid use of PRINTCTL procedures for maximum performance; rather, use ODBC's client driver data type conversions.

# Appendix A • Butler SQL DAM specifics

This appendix describes additional information about Butler SQL which relates only to DAL/DAM. The majority of the information here is relevant to low level programming (C, C++, Pascal) only. You do not need the information presented here if you are using a client application or are writing applications using a higher level language like HyperCard or AppleScript.

## DAM Data Types

**Table 16: SQL and DAM data types**

| Data type | SQL type | DAM type | Description |
|---|---|---|---|
| BOOLEAN | 1 | ‘bool’ | A logical value that can assume the values TRUE, FALSE, or NULL (unknown) |
| SMINT | 2 | ‘shor’ | A signed 16-bit integer |
| INTEGER | 3 | ‘long’ | A signed 32-bit integer |
| SMFLOAT | 4 | ‘sing’ | A signed 32-bit floating-point number |
| FLOAT | 5 | ‘doub’ | A signed 64-bit floating-point number |
| DATE | 6 | ‘date’ | A date consisting of a year, month, and day |
| TIME | 7 | ‘time’ | A time consisting of an hour (0–23), minute, second, and hundredth of a second |
| TIMESTAMP | 8 | ‘tims’ | A date-and-time stamp, with the combined components of the DATE and TIME types |
| CHAR | 9 | ‘TEXT’ | A fixed-length character string, with a constant length (number of characters) |
| DECIMAL | 10 | ‘deci’ | A signed decimal number that has an associated precision (total number of decimal digits) and scale (number of digits to the right of the decimal point) |
| MONEY | 11 | ‘mone’ | A data type with the same characteristics as DECIMAL, interpreted as a currency amount |
| VARCHAR | 12 | ‘vcha’ | A variable-length character string, with an associated length (current number of characters) |
| VARBIN | 13 | ‘vbin’ | A variable-length byte string, with an associated length (current number of bytes) |
| LONGCHAR | 14 | ‘lcha’ | A variable-length character string, with an associated length (current number of characters) up to 2.3G |

**Table 16: SQL and DAM data types**

| Data type | SQL type | DAM type | Description |
|---|---|---|---|
| ❖ See “Butler’s extended data types” on page 37 for information on data types 13–28. | | | |
| PICTURE | 24 | ‘pict’ | Identical to VARBIN, except data is expected to be in the Macintosh ‘PICT’ (picture) resource format. |
| SOUND | 25 | ‘snd’ | Identical to VARBIN, except data is expected to be in the Macintosh ‘snd ’ (sound) resource format |
| ICON | 26 | ‘cicn’ | Identical to VARBIN, except data is expected to be in the Macintosh ‘cicn’ (color icon) resource format |
| MOVIE | 27 | ‘moov’ | Identical to VARBIN, except data is expected to be in the Macintosh ‘moov’ (QuickTime movie information) resource format |
| DOCUMENT | 28 | ‘docu’ | Identical to VARBIN, except that the data stored is a Macintosh file or alias in binary format. |
| CURSOR | | | A value identifying an active SQL rowset |
| OBJNAME | | | A data item whose value identifies a SQL identifier |
| GENERIC | | | An item used to declare a SQL variable that can assume any of the other data types when data is assigned to it |

# Data Access Manager

## Long Data Items

The current interfaces by which client applications can communicate with Butler SQL, namely the Data Access Manager (DAM) and the older (but still commonly used) DAL (previously called CL/1) driver, both suffer a serious limitation in that data items sent to, or received from, the server are limited to being 32K bytes long. This is due to the fact that both interfaces allow for a data length parameter that is only a two byte value.

Needless to say, this is a problem if you want to transfer two minutes of dogcow barking sounds, or a QuickTime movie of barking dogcows grazing in a field. Aside from the minor detail of needing SQL data types to handle sounds and movies, the interfaces must be adjusted so that these long items can be handled.

To this end, the DAM and CL/1 interfaces have been tweaked to allow for long integer data lengths to be passed.

## How it's done

In the case of DAM, all calls to DAM go through the _Pack13 trap, with a routine selector code in D0. The glue for the routines in DatabaseAccess.p expand the routine call in order to put the appropriate selector into D0 and invoke _Pack13. The Butler Access database extension patches the _Pack13 trap when it receives the DBOpen message (a private message from DAM to database extensions). The patch is removed when DAM sends the DBClose message.

The patch code intercepts _Pack13 calls before DAM gets a hold of them and interprets two new selector codes: –1, for DBGetLongItem, and –2 for DBSendLongItem. (These values were chosen because they stand the best chance of not already being defined and used publicly or privately by DAM, as all the other DAM selectors appear to be positive numbers.) Then the patch code saves off the long lengths in a private storage area, sets a private flag to signal that a "long" call was made, adjusts the stack and selector code to appear as though a "short" call was being made, and passes on to DAM.

Once DAM calls Butler Access to process the call, the flag is interrogated to determine if a long call was actually made, and uses the saved off long length instead of the now invalid value passed by DAM. When the call is finished, the patch code again gets control, fixes up the stack and return value, and returns to the calling program.

### Important

This code could break if the selector codes –1 and –2 were at some point

used by DAM. Also, this is a tail patch—a programming technique not recommended by Apple.

The DAM routines are simply copies of the "short" routines with long length parameters. They also pass the length in a different field in the cl1CB control block structure since the length field is only a word; the exinf1 field is used instead. The routine selectors have been changed to tell the driver that the length is not where it should be.

As with the DAM patches, there is a possibility that Apple will eventually use these selector codes for other purposes, in which case the code will no longer work.

## The long item interface

To allow for larger data items, the DAM and DAL interfaces have been tweaked. This section describes these new interfaces, how to use them, and how they were changed; as future incarnations of the official interfaces may break them.

## Are long items supported?

Before making any long item calls, you should make sure that the database extension in use can handle them. In the Butler Access and Butler SQL database extensions, there is a CNFG resource with an ID of 0 which controls various aspects of their operation. The Pascal record structure of the resource is:

```
TConfigFlagsHdl = ^TConfigFlagsPtr;
TConfigFlagsPtr = ^TConfigFlagsRec;
TConfigFlagsRec = RECORD
quietCTB: Boolean;
filler1: Boolean;
quietPPC: Boolean;
filler2: Boolean;
allowWNE: Boolean;
filler3: Boolean;
longItems: Boolean;
filler4: Boolean;
```

```
END;
```

If longItems is TRUE (1), you are safe in making long item calls. Of course, if the resource does not exist (as in pre-1.2 and all current non-Butler database extensions), long items are not supported.

## DAM interface

The DAM interface sends and receives data items with the DBSendItem and DBGetItem calls, whose interfaces are:

```
FUNCTION DBGetItem(sessID: Longint; timeout: Longint;
    VAR dataType: DBType;
    VAR len, places, flags: Integer; buffer: Ptr;
    asyncPb: DBAsyncParmBlkPtr): OSErr;
FUNCTION DBSendItem(sessID: Longint; dataType: DBType;
    len, places, flags: Integer;
    buffer: Ptr; asyncPb: DBAsyncParmBlkPtr): OSErr;
```

The new interfaces with their in-line glue code are:

```
FUNCTION DBGetLongItem(sessID: Longint;
    timeout: Longint; VAR dataType: DBType;
    VAR len: Longint; VAR places, flags: Integer;
    buffer: Ptr; asyncPb: DBAsyncParmBlkPtr): OSErr;
INLINE $303C,$FFFF,$A82F;
FUNCTION DBSendLongItem(sessID: Longint;
    dataType: DBType; len: Longint;
    places, flags: Integer; buffer: Ptr;
    asyncPb: DBAsyncParmBlkPtr): OSErr;
INLINE $303C,$FFFE,$A82F;
```

These interfaces can be added directly into the DatabaseAccess.p interface file or declared somewhere in your own code. It is hoped that a future version of DAM will have long integer length parameters, making the in-line glue unnecessary.

## CL/1

Many applications written before DAM was a reality communicate with SQL servers, including Butler SQL, using the CL/1 API. This API works through a driver (.CL1E5) instead of a database extension. The Butler SQL database extension (or Butler Six, on System 6 machines) replaces the driver installed by the SQL database extension with one that knows how to talk to Butler SQL.

Again, the CL/1 interface that client applications call all have word length parameters:

```
FUNCTION CLGetItem (sessid : longint; timeout : integer;
 var typ, len, places, flags : integer;
buffer : ptr) : integer;

FUNCTION CLSendItem (sessid : longint; typ, len, places, flags : integer;
buffer : ptr) : integer;
```

The new API calls to handle long data items are:

```
FUNCTION CLGetLongItem (sessid : longint; timeout : integer;
 var typ : integer; var len : longint;
var places, flags : integer; buffer : ptr) : integer;
FUNCTION CLSendLongItem (sessid : longint; typ : integer;
len : longint; places, flags : integer;
buffer : ptr) : integer;
```

or, for C:

```
CLGetLongItem (longsessid, int*timeout, short*typep, int*lenp, short*placesp,
short*flagsp, char*buffer)

CLSendLongItem (longsessid, inttyp, longlen, intplaces, intflags, char*buffer)
```

The "glue" for these additional routines is in the form of a object module that must be compiled and linked with your code in the same way that the other CL/1 API routines are glued in. Both Pascal and C versions of these routines are available, and are reproduced at the end of this note.

As with the DAM patches, these additional routines are available only if the appropriate code is linked in, and even then it is only currently understood by the Butler SQL version of the.CL1E5 driver. Other drivers will likely return a paramErr, or some reasonable facsimile.

### "The Fine Print"

The CL/1 routines are also not particularly magical, and are simply copies of the "short" routines with long length parameters. They pass the length in a different field in the cl1CB control block structure as the length field is only a word; the exinf1 field is used instead. Also, the routine selectors have been changed to tell the driver that the length is not where it should be. As with the DAM patches, due to the fact that new selector codes have been defined, the possibility of these codes being officially used at some point in the future exists; in which case, the code will no longer work.

## Hosts

This section details the process by which SQL client applications can access Butler SQL and, in doing so, provide a standard that can be used to interact with any database access extension.

The motivation behind doing this is that currently all SQL client applications use the SQL database extension (under System 7.0) or the Cl/1 driver (under System 6) provided by Apple, and connect to data servers that run on other machines. In doing so, they have not had to deal with interfacing to other database extensions, and so have hard-coded some

things that should be made general and used approaches specific to the Apple database extension that, again, should be more general. While we hope that our approach helps in providing a standard, we are, of course, more interested in getting any standard adopted; so long as we all agree on what it is and can get on to the business of using it.

The three aspects of data server communication that are of immediate interest, and that will be talked about in this section, are database extension names, hosts supported by a database extension, and specific capabilities/requirements of the data server.

## Finding database extensions

Under System 7.0, the method of communicating with a data server is, of course, through the database extension. Apple's stated intent is that all data server developers supply a database extension that will be used for communication; rather than supply details of the communication protocols to the Apple SQL extension. Those protocols are constantly evolving and would be difficult for developers to keep up with.

Database extensions are stored in the Extensions folder within the System Folder, and are of type 'ddev'. Therefore, any application that supports communication using a variety of database extensions must enumerate through this folder to find all appropriate files. The names of the files should be displayed to the user for selecting an extension.

All database extensions must have a 'STR ' resource, number 128, that defines the name of the ddev. This name is to be sent in the DBInit call in the ddevName parameter, and is not the name that a user should see. In Apple's case, the 'STR ' resource is 'SQL'; and in Butler's case, the string is 'BUTLER'. Note that while Apple's database extension has the same file name as the string, but it does not have to (especially since the user could have renamed the file). In Butler's case, the file name is shipped as 'Butler Access'. Client applications must get this string resource from the ddev file to be able to use the extension; it is the sole method by which the operating system can find the correct ddev to use.

## Finding hosts

While finding database extensions is a simple task, hosts present a more involved problem. The current state of affairs is an outgrowth of the original System 6 implementation of SQL by Apple, which involves a rather cryptic text file called 'SQL Preferences' (formerly 'Hosts.cl1'). This file is located in Preferences folder under System 7 and defines not only the available hosts, but the communication parameters used to access them.

In a multiple database extension environment, this approach is not ideal. At the very least, the format of the 'SQL Preferences' file will have to change to indicate what hosts are supported by what database extension, if all host information, regardless of database extension, is to be kept there. Alternatively, and, we argue, preferably, database extensions should use whatever method they choose to maintain host connection information, and client applications should interrogate the extension to find out the names of the hosts it supports.

The solution that we have adopted is to include a string list ('STR#') resource in the ddev, with an id of 256 and named “Host Names”. This string list contains all the currently defined host names that the ddev knows about, and is automatically maintained by the ddev. All that is required by client applications is opening the ddev file and getting the string list to obtain the current hosts.

As an immediate yet temporary fix until client applications use the methods described in this document to connect to Butler SQL, an interim approach to finding hosts is to include the host names that Butler SQL supports in the 'SQL Preferences' file. Specific connection information is not needed in this file. The downside to this approach is that the host list obtained from the 'SQL Preferences' file will not be extension-specific, and a user could potentially select a host that a particular database extension cannot access.

## Data server profiles

In order to effectively communicate with a data server, client applications need certain information about the types of commands and capabilities supported by the database extension and data server. The method implemented by Butler SQL is to include a 'STR ' resource, with id 256 and named “Profile”, that has a series of Y/N flags similar to the profile string returned by the DESCRIBE DBMS command. The properties described by each flag need a great deal of cooperative effort, but based on requests by several client application vendors, the following flags are currently defined in the Butler database extension:

### Column Information

- Requires server logon
- Requires DBMS logon
- Requires database logon
- Supports outer join (not supported by SQL)
- Supports null sessions
- Supports DESCRIBE CURRENT DBMS command
- Requires user name for OPEN DATABASE
- Supports DBBreak
- Supports left/right joins (not supported by SQL)

## Getting host information

This section discusses the implementation of reading the hosts out of a string list in the database extension. The aim of this discussion is to go through the steps needed to implement this design and show how one would go about enumerating through the ddevs in the extensions folder and retrieving the hosts from the string lists in each of the ddevs.

## Code segments

These are the main code segments which are used to manipulate the database extensions and the host string lists that are in the resource forks of the extensions. The code is written in procedural MPW Pascal. The analysis of the problem calls for:

- GetDDEVtable to be called to build a list of the ddevs,
- GetIndDDEV would then be called to enumerate through the list to show the user each ddev filename and then called again to fetch that filename,
- GetDDEVhosts would then build the host list, and
- GetIndHost would be called to display and fetch the user's choice.

```
FUNCTION        GetDDEVtable: HNameList;
PROCEDURE       GetIndDDEV(aDDEVList: HNameList;
                index: INTEGER;
                VAR aDDEVfileName: TNameStr):
FUNCTION        GetDDEVHosts(aDDEVfileName: TNameStr ) :
                HNameList;
PROCEDURE       GetIndHosts(aHostsList : HNameList;
                index : INTEGER ;
                VAR aHostName : TNameStr );
FUNCTION        MapDDEVname ( aDDEVfileName : TNameStr )
                : Str63;
FUNCTION        FindItemAt(aNameList : HNameList ;
                aNameStr: TNameStr):INTEGER;
```

## GetDDEVtable

```
FUNCTION        GetDDEVtable : HNameList;
```

This function runs in both system 7 and in system 6 environments with the following effects.

The function GetDDEVtable calls the procedure EachDDEV, which enumerates through the extensions folder looking for extensions of the file type 'ddev', and executes the sub-procedure AddADDEV for each database extension that is eligible to be considered. Therefore, we build up a handle of evenly spaced filenames of each of the database extensions which is passed back to the caller of the function.

## GetIndDDEV

```
PROCEDURE    GetIndDDEV(aDDEVList : HNameList ;
              index : INTEGER ;
              VAR aDDEVfileName : TNameStr);
```

The procedure GetIndDDEV is used to index through the DDEVlist handle and return the filename of the ddev in the TNameStr aDDEVfileName. We would use this procedure to enumerate through the list of DDEV filenames so we could display them in a list on the screen.

## GetDDEVHosts

```
FUNCTION    GetDDEVHosts(aDDEVfileName : TNameStr ) :
            HNameList;
```

The function GetDDEVHosts takes a filename of a database extension and returns a handle to a list of host names. The host names are stored in a STR# resource of the name 'Host Names' and ID 256. GetDDEVHosts calls DoWhileResourceOpen, which opens the resource fork and then calls back to the sub-procedure GetHostsList, which enumerates through the string list building up a handle of host names of type HNameList which is passed back to the caller of GetDDEVHosts.

## GetIndHosts

```
PROCEDURE    GetIndHosts(aHostsList : HNameList;
              index : INTEGER ;
```

```
                                VAR aHostName : TNameStr );
```

This function is similar in form and function to GetIndDDEV. GetIndHosts will index through the host list to the appropriate value and return the host name in the TNameStr aHostName.

## MapDDEVname

```
FUNCTION        MapDDEVname ( aDDEVfileName : TNameStr )
                : Str63;
```

MapDDEVname calls DoWhileResourceOpen to open the resource fork and then calls the sub-procedure GetDDEVStr, which gets the STR resource id 128 where the name of the ddev is stored. This value is 'SQL' in Apple's ddev but is coded to 'BUTLER' in the Butler Access ddev. The ddev name from this resource is passed to DBINIT or CL1INIT.

## FindItemAt

```
FUNCTION        FindItemAt(aNameList : HNameList ;
                aNameStr: TNameStr):INTEGER;
```

FindItemAt is included to complement the GetIndHosts and GetIndDDEV and can be used to return the index of the ddev name or host name in the appropriate list.

## TNameList Data Structure

Both the ddev list and host list utilize the TNameList data structure to store and manipulate the items in the list. Each item is of the data type TNameStr. This data structure was used because it not only provides dynamic allocation, but does not allocate unnecessary handles and helps the heap from being fragmented. We provide procedures for creating, disposing, adding, and finding out how many items are in the list.

```
TNameStr = Str31;
TNameList = ARRAY [0..MAXINT] of TNameStr;
PNameList = ^TNameList;
```

```
                    HNameList = ^PNameList;
FUNCTION            CreateNameList: HNameList;
PROCEDURE           DisposeNameList(aNameList : HNameList);
FUNCTION            AddToNameList(aNameList : HNameList;
                    aName : TNameStr):OSERR;
FUNCTION            NumOfNames(aNameList :HNameList ) : Long-
                    Int;
```

## Configuring Database Extensions

This section describes a resource (CNFG, id = 0) in the database extensions shipped with Butler SQL that allows certain parameters within the extension to be configured for certain needs. It should be noted that, in general, the shipping configurations of the database extensions should not be changed; it is only under very unusual circumstances that any of these settings need modification. If in doubt, do not modify and call EveryWare Development Corp.

**Note**

The format of this resource is subject to change at any time, and while every effort will be made to ensure that existing fields will not change, this is not guaranteed. Client application developers writing code to the specifics of this resource format must also be careful to not modify reserved parts of the structure, or risk future incompatibility. For easy (and safe!) modification, a template also exists in each database extension so that any changes can be made using ResEdit™.

The database extension configuration resource has the following Pascal structure:

```
TConfigFlagsRec = RECORD
   quietCTB: Boolean;    { no CTB dialogs }
   reserved1: Boolean;
   quietPPC: Boolean;    { no PPC dialogs }
   reserved2: Boolean;
```

```
allowWNE: Boolean;    { perform WNE in synch loops }
reserved3: Boolean;
longItems: Boolean;   { support long items }
reserved4: Boolean;
ctbWriteLen: Longint; { CTB i/o block limit }
execWNE: Boolean;     { WNE after DBExec for CL/1 }
reserved5: Boolean;
END;
```

The first two fields control whether or not dialogs can appear during a connection for Communications Toolbox (CTB) or program linking (PPC) connections. If the field is true, then no dialogs will appear during the connection. You may want to set a connection "quiet" if the dialogs are interfering with a particular application's window drawing and refreshing.

### allowWNE

The allowWNE field controls whether the database extension calls the _WaitNextEvent trap while it is waiting for data to return from Butler, allowing other applications on the client machine to get processing time. It is particularly important that this flag be true when the client and Butler are on the same machine, otherwise a connection will hang!

### longItems

The longItems field defines whether or not the database extension supports long item (>32K) DBGetItem and DBSendItem calls. These are patches made to the Database Access Manager by the Butler extensions, and if the flag is true then the patches are available to client applications.

### ctbWriteLen

The ctbWriteLen field specifies the maximum block size to be used during a CTB read or write call. This is needed because some connection tools cannot handle arbitrarily large I/O calls (such as the ADSP tool).

The last currently defined field controls whether or not a _WaitNextEvent call is made immediately after sending a DBExec to Butler. This is a very specific flag, applicable only to connections made using the CL/1 API,

and is included because of some peculiar behavior of the Omnis 7 SQL externals when run on the same machine as the Butler server. Note that if this flag is set to true, then the allowWNE flag must also be set to true.

# Data Access Language

**Note**

This section is applicable mainly to those writing client applications in low-level programming languages such as C and Pascal. You do not need the information presented here if you using existing client applications or writing applications in a higher-level language such as HyperCard.

## How it's done

In the case of the DAL, all calls to DAM go through the _Pack13 trap, with a routine selector code in D0. The glue for the routines in DatabaseAccess.p expand the routine call to put the appropriate selector into D0 and invoke _Pack13. The Butler Access database extension patches the _Pack13 trap when it receives the DBOpen message (a private message from DAM to database extensions). The patch is removed when DAM sends the DBClose message.

The patch code intercepts _Pack13 calls before DAM gets a hold of them and interprets two new selector codes: –1, for DBGetLongItem, and –2 for DBSendLongItem. These values were chosen because they stand the best chance of not already being defined and used publicly or privately by DAM since all the other DAM selectors appear to be positive numbers. The patch code then saves off the long lengths in a private storage area, sets a private flag to signal that a "long" call was made, and then adjusts the stack and selector code so that it appears as though a "short" call was being made and passes on to DAM. Once DAM calls Butler Access to process the call, the flag is interrogated to determine if a long call was actually being made, and uses the saved off long length instead of the now

invalid value passed by DAM. When the call is finished, the patch code again gets control, fixes up the stack and return value and returns to the calling program.

This code could break if the selector codes –1 and –2 were at some point used by DAM. Also, this is a tail patch—a programming technique not recommended by Apple.

The DAL routines are simply copies of the "short" routines with long length parameters. They also pass the length in a different field in the cl1CB control block structure since the length field is only a word; the exinf1 field is used instead. The routine selectors have been changed to tell the driver that the length is not where it should be. As with the DAM patches, Apple may eventually use these selector codes for other purposes, in which case the code will no longer work.

## The long item interface

The current interfaces by which client applications can communicate with Butler, namely the Data Access Manager (DAM) and the older DAL (previously called CL/1) driver both suffer a serious limitation in that data items sent to or received from the server are limited to 32K. This is because both interfaces allow for only a two-byte data length parameter.

To allow for larger data items, the DAM and DAL interfaces have been tweaked. This section describes these new interfaces, how to use them, and how they were changed, since future incarnations of the official interfaces may break them.

## C and Pascal code listings

Here are the two new routines, in both Pascal and C, if you can't get them any other way:

(file cl1longapi.p)

This file contains the source code for extensions to the DAL 1.3 API to allow for the sending and receiving of long (>32K) data items.

```
UNIT CL1LongAPI;
INTERFACE
USES
     CL1_API;
FUNCTION CLGetLongItem(sessid: Longint; timeout: Integer;
     VAR typ: Integer; VAR len: Longint;
     VAR places, flags: Integer;
     buffer: Ptr): Integer;
FUNCTION CLSendLongItem(sessid: Longint; typ: Integer;
     len: Longint; places, flags: Integer;
     buffer: Ptr): Integer;
IMPLEMENTATION
{$I cl1longapi.inc1.p}
END;
```

## cl1longapi.inc1.p

The next two functions are virtually identical to their "short" counterparts, with the exception that the lenp parameter is a long integer so that items that are greater than 32K bytes in size may be passed.

The routine selectors have been defined as 23 for CLGetLongItem and 24 for CLSendLongItem. The .CL1E5 driver must support these selectors for these routines to be usable.

Since the cl1CB structure only defines the len field as a short value, the exinf1 field is used instead to pass the item length.

```
FUNCTION CLGetLongItem(sessid: Longint; timeout: Integer;
     VAR typ: Integer; VAR len: Longint;
     VAR places, flags: Integer;
     buffer: Ptr): Integer;
VAR
cl1CB : cl1CB;
BEGIN
cl1CB. retstatus := A_NOTCONN;
cl1CB. sessid    := sessid;
cl1CB. colnum    := A_NXTCOL;
cl1CB. buffer    := buffer;
cl1CB. cl1type   := typ;
cl1CB. exinf1    := len;
```

```
cl1CB. timeout   := timeout;
cl1CB. request   := 23; { new selector! }
cl1api (cl1CB);
if cl1CB. retstatus >= 0 then begin
typ    := cl1CB. cl1type;
len    := cl1CB. exinf1;
places := cl1CB. places;
flags  := cl1CB. flags;
end;
CLGetItem := cl1CB. retstatus;
END; { CLGetLongItem () }
FUNCTION CLSendLongItem(sessid: Longint; typ: Integer;
      len: Longint; places, flags: Integer;
      buffer: Ptr): Integer;
VAR
cl1CB : cl1CB;
BEGIN
cl1CB. retstatus := A_NOTCONN;
cl1CB. sessid    := sessid;
cl1CB. buffer    := buffer;
cl1CB. cl1type   := typ;
cl1CB. exinf1    := len;
cl1CB. places    := places;
cl1CB. flags     := flags;
cl1CB. request   := 24; { new selector! }
cl1api (cl1CB);
CLSendItem := cl1CB. retstatus;
END; { CLSendLongItem () }
```

## cl1longapi.h

```
extern CLGetLongItem (long,int,short*,long*,short*,short*,char*);
extern CLSendLongItem (long,int,long,int,int,char*);
```

## cl1longapi.c

The next two functions are virtually identical to their "short" counterparts, with the exception that the lenp parameter is a long integer so that items that are greater than 32K bytes in size may be passed.

The routine selectors have been defined as 23 for CLGetLongItem and 24 for CLSendLongItem. The .CL1E5 driver must support these selectors for these routines to be useable.

**Note**

Since the cl1CB structure only defines the len field as a short value, the exinf1 field is used instead to pass the item length.

```
#include "cl1api.h"
#include "cl1longapi.h"
CLGetLongItem (sessid, timeout, typep, lenp, placesp, flagsp, buffer)
long sessid;
int timeout;
int *lenp;
short *typep, *placesp, *flagsp;
char *buffer;
{
LOCAL struct cl1CB cl1CB;
cl1CB. retstatus = A_NOTCONN;
cl1CB. sessid    = sessid;
cl1CB. colnum    = A_NXTCOL;
cl1CB. buffer    = lenp  ? cfs (buffer) : (char far *) 0;
cl1CB. cl1type   = typep ? *typep : A_ANYTYPE;
cl1CB. exinf1    = lenp  ? *lenp  : 0;
cl1CB. timeout   = timeout;
cl1CB. request   = 23; /* new selector */
cl1api (& cl1CB);
if (cl1CB. retstatus >= 0)
      {
        if (typep)
            *typep   = cl1CB. cl1type;
        if (lenp)
            *lenp    = cl1CB. exinf1;
        if (placesp)
            *placesp = cl1CB. places;
        if (flagsp)
            *flagsp  = cl1CB. flags;
            }
  return cl1CB. retstatus;
```

```
} /* CLGetLongItem () */

CLSendLongItem (sessid, typ, len, places, flags, buffer)
long sessid;
long len;
int typ, places, flags;
char *buffer;
{
  LOCAL struct cl1CB cl1CB;
  cl1CB. retstatus = A_NOTCONN;
  cl1CB. sessid    = sessid;
  cl1CB. buffer    = cfs (buffer);
  cl1CB. cl1type   = typ;
  cl1CB. exinf1    = len;
  cl1CB. places    = places;
  cl1CB. flags     = flags;
  cl1CB. request   = 24; /* new selector */
  cl1api (& cl1CB);
  return cl1CB. retstatus;
} /* CLSendLongItem () */
```

## Data Access Language interface

Many applications communicate with DAL servers using the DAL API. This API works through a driver (.CL1E5) instead of a database extension. The Butler SQL database extension (or Butler Six, under System 6) installs a driver with the same name as Apple's, but which knows how to talk to Butler.

Again, the DAL interface that client applications call has word length parameters:

```
function CLGetItem (sessid : longint;
    timeout : integer; var typ, len, places,
    flags : integer; buffer : ptr) : integer;
function CLSendItem (sessid : longint; typ, len,
    places, flags : integer; buffer : ptr) : integer;
```

The new API calls to handle long data items are:

```
function CLGetLongItem (sessid : longint;
    timeout : integer; var typ : integer;
    var len : longint; var places, flags : integer;
    buffer : ptr)
    : integer;
function CLSendLongItem (sessid : longint;
    typ : integer; len : longint;
    places, flags : integer;
    buffer : ptr) : integer;
```

or, in C:

```
CLGetLongItem (sessid, timeout, typep, lenp, placesp,
flagsp, buffer)
long sessid;
int timeout, *lenp;
short *typep, *placesp, *flagsp;
char *buffer;
CLSendLongItem (sessid, typ, len, places, flags,
buffer)
long sessid, len;
int typ, places, flags;
char *buffer;
```

The "glue" for these additional routines is in the form of an object module that must be compiled and linked with your code, in the same way that the other DAL API routines are glued in. Both Pascal and C versions of these routines are available.

As with the DAM patches, these additional routines are available only if the appropriate code is linked in.

# Appendix B • Butler SQL ODBC specifics

The following table displays and describes the SQL and ODBC data types used by Butler SQL.

**Table 17: SQL and ODBC data types**

| Data type | SQL type | ODBC type | Description |
|---|---|---|---|
| BOOLEAN | 1 | | A logical value that can assume the values TRUE, FALSE, or NULL (unknown) |
| SMINT | 2 | SQL_SMINT | A signed 16-bit integer |
| INTEGER | 3 | SQL_INTEGER | A signed 32-bit integer |
| SMFLOAT | 4 | | A signed 32-bit floating-point number |
| FLOAT | 5 | SQL_FLOAT | A signed 64-bit floating-point number |
| DATE | 6 | SQL_DATE | A date consisting of a year, month, and day |
| TIME | 7 | SQL_TIME | A time consisting of an hour (0–23), minute, second, and hundredth of a second |
| TIMESTAMP | 8 | SQL_TIMESTAMP | A date-and-time stamp, with the combined components of the DATE and TIME types |
| CHAR | 9 | SQL_CHAR | A fixed-length character string, with a constant length (number of characters) |

**Table 17: SQL and ODBC data types**

| Data type | SQL type | ODBC type | Description |
|---|---|---|---|
| DECIMAL | 10 | SQL_DECIMAL | A signed decimal number that has an associated precision (total number of decimal digits) and scale (number of digits to the right of the decimal point) |
| MONEY | 11 | | A data type with the same characteristics as DECIMAL, interpreted as a currency amount |
| VARCHAR | 12 | SQL_VARCHAR | A variable-length character string, with an associated length (current number of characters) |
| VARBIN | 13 | SQL_VARBINARY | A variable-length byte string, with an associated length (current number of bytes) |
| LONGCHAR | 14 | SQL_LONGVARCHAR | A variable-length character string, with an associated length (current number of characters) up to 2.3G |
| ❖ See “Butler’s extended data types” on page 37 for information on data types 13–28. | | | |
| PICTURE | 24 | | Identical to VARBIN, except data is expected to be in the Macintosh ‘PICT’ (picture) resource format. |
| SOUND | 25 | | Identical to VARBIN, except data is expected to be in the Macintosh ‘snd ’ (sound) resource format |
| ICON | 26 | | Identical to VARBIN, except data is expected to be in the Macintosh ‘cicn’ (color icon) resource format |
| MOVIE | 27 | | Identical to VARBIN, except data is expected to be in the Macintosh ‘moov’ (QuickTime movie information) resource format |

**Table 17: SQL and ODBC data types**

| Data type | SQL type | ODBC type | Description |
|---|---|---|---|
| DOCUMENT | 28 | | Identical to VARBIN, except that the data stored is a Macintosh file or alias in binary format. |
| CURSOR | | | A value identifying an active SQL rowset |
| OBJNAME | | | A data item whose value identifies a SQL identifier |
| GENERIC | | | An item used to declare a SQL variable that can assume any of the other data types when data is assigned to it. |
| BIGINT | 30 | SQL_BIGINT | Precision: 3<br>Maximum length: 1 byte. |
| TINYINT | 31 | SQL_TINYINT | Precision: 19 (signed) or 20 (unsigned).<br>Maximum length: 20 bytes. |

# 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



## G



## H



## I



## K



## L



## M

## N



## O



## P



## Q



## R



## S

## 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.**

**Copyright © 1992–1996 EveryWare Development Corporation. All rights reserved.**

**2.0 2-96**

## Publishing Tools

**This manual was created on a Macintosh using FrameMaker from Frame Technology Corporation.**

**The Butler SQL software, ButlerClient, ButlerHosts, ButlerTools, ButlerLink Access, Tango, the documentation, and associated materials are © 1992–1996 EveryWare Development Corp. All rights reserved. Butler SQL is a trademark of EveryWare Development Corp. All other trademarks mentioned are the property of their respective owners.**