helpsystems

SQL Reference Guide

Sequel 11

201907122115

Table of Contents

Understanding SQL

The key to getting the most out of Sequel lies in your ability to understand and use Structured Query Language (SQL). Although you can create and run many queries using SequelShowcase's prompting capabilities, you can only scratch the surface of Sequel's potential without a thorough knowledge of SQL.

SQL is both powerful and simple. With it you can express your ideas quickly and easily. SQL eliminates the need for you to create a computer program that specifies exactly which information to retrieve, where to find it, and how to present it. Since you are less involved in telling the computer how to do something, you can spend more time telling it what you want done.

This section will teach you about the SQL query statement and how you can use it to retrieve information from your database. It will teach you about the structure of the statement and show examples that demonstrate how you can create your own Sequel requests.

Each of the queries in this section can be performed using the files and views in the SEQUELEX library. You can run the examples from the user interface or by using the DISPLAY command directly from command entry. As you do, you will begin to understand SQL and how it works. Soon, you'll be able to create your own SQL statements that return the information you need from your database.

SQL Query Statement

Nearly every Sequel activity involves an SQL query statement. The query statement gets information from the database and delivers it to you for display, printing, reporting, or downloading to another file.

You specify:

what information you want to see,
where the system will find it, and
how you want it presented (display, printout, etc.)

The query statement allows you to create different *views* of your data. What you see depends on your needs. It gives you capabilities for:

calculating
ordering
summarizing
selecting and omitting records

The SQL query statement is also called the SELECT statement because it begins with the word "SELECT".

The complete query statement has a general form like this:

```
SELECT [DISTINCT] item-list
      FROM file-list
      JOIN [join-specifications]
      WHERE search-conditions
      GROUP BY field-list
      HAVING search-conditions
[UNION [ALL] subselect ...]
      ORDER BY field-list
```

Although it may look a little intimidating at first, you'll soon see just how easy it is to understand and use. In practically no time at all, you'll be using only the parts of the statement that you need to accomplish your requests!

Appearance is Up to You

Capitalization and spacing in the statement are entirely up to you. Sequel is not case dependent. Upper and lower case letters make no difference in how the statement is interpreted. Using upper and lower case letters can be an effective way to make an SQL statement more clear to someone who tries to read it. Except for the display presented by the Change View (CHGVIEW) command, Sequel will automatically format the SQL statement when displaying it to you. To increase its readability, SQL keywords will be presented in upper case; field and file names will be shown in lower case letters.

Capitalization does make a difference when fields are compared to character values. The case (upper or lower) of the fields used in a character comparison must match character for character. You need to remember that the character value JOHNSON does not match the character value Johnson because they do not have the same exact character (including case) in every position.

Sequel Clauses

The query statement has several parts, known as clauses. Although there are eight of them, most are optional. The first two— SELECT and FROM are required. Others may be required, depending on the circumstances and the context of the query. The clauses must appear in a specific order or an error will occur. Clauses that are not needed are simply omitted from the SQL statement.

Each query requires at least one subselect. The subselect tells the 'what and where' of the query. In essence, it defines a set of rows and columns to be returned. A complete query can involve up to 32 subselects. Each one can have up to six clauses, but only the SELECT and FROM clauses are ever required. Clauses within a subselect must appear in this order:

SELECT	what fields and calculations you want
FROM	where in the database to find them
JOIN	what fields link the files together
WHERE	what conditions must be met by the records
GROUP BY	how records should be bundled together
HAVING	what conditions must be met by grouped records

If the query includes more than one subselect, each of them must be separated by a UNION clause.

If you want the records returned to you in a particular sequence, you can use an ORDER BY clause at the very end of the statement. It defines how the records should be ordered.

The eight clauses of the SQL query provide outstanding function for working with the information in your database. If you are not already familiar with SQL, you will be amazed at what can be accomplished with a few simple directives!

Each of Sequel's clauses are summarized below. In-depth descriptions, detailing all of the functions of each of them, begin on page 51.

SELECT

The SELECT clause indicates what information you want to see. Following the word SELECT is an *item-list* that indicates the fields, calculations, and/or literal values that will be included in your view of the data.

If the word DISTINCT follows the SELECT keyword, only unique rows will be returned by the view. Duplicate rows will be suppressed.

FROM, JOIN

The FROM and JOIN clauses are used to tell the system where the data should come from and how the system should acquire it. The *file-list* specified in the FROM clause can contain up to 32 file names.

If there is more than one file listed in the FROM clause, the *join-specifications* in the JOIN clause can indicate how the files should be linked together. The JOIN clause isn't required. The joining specifications can be included in the WHERE clause instead of the JOIN clause if you want.

Note: SELECT and FROM clauses are required on every SQL statement.

WHERE

The WHERE clause allows you to specify which records should be included in the view. You indicate a series of *search-conditions*. If all of them are satisfied, a record qualifies for the view. If a record fails one of your tests, it is omitted from the view.

If no WHERE clause is specified, all records from the file(s) specified in the FROM clause will be included.

The WHERE clause can provide an alternative method for specifying joining criteria, although most people find that placing the *linking* specifications in the JOIN clause and the *selection* specifications in the WHERE clause makes the statement more readable.

GROUP BY, HAVING

The GROUP BY and HAVING clauses work together. The GROUP BY phrase causes records to be arranged into groups that have the same values for the *field-list* named in the clause. HAVING can then be used to choose which groups are included in the view.

Grouping records into sets can be very useful when you want to create "summary only" queries. In a grouped query you can count records, calculate totals or averages, or find the highest or lowest values within each record set. As a rule, when you create a grouping query your SELECT clause will include:

the fields in the GROUP BY *field-list*, and

one or more summary functions that count, total, etc. the records within each set

The HAVING clause applies one or more search-conditions to the summarized result of the grouping. As with the WHERE clause, if no HAVING conditions are specified all the groups will be selected.

Note: The HAVING clause cannot be used unless the GROUP BY phrase is also specified, but a grouping query need not include a HAVING clause.

UNION and UNION ALL

UNION is used to merge the sets of records created by two subselects. A subselect (SELECT, FROM, JOIN, WHERE, GROUP BY, HAVING) statement appears on each side of the UNION or UNION ALL phrase. When the view is run, rows will be acquired from each subselect and merged together. Duplicates will be eliminated (UNION), or passed through to you in the final result (UNION ALL).

ORDER BY

ORDER BY is used to sort the records. The clause specifies a *field-list* that indicates which fields in the view should be used for ordering. It also tells whether columns should be placed in ascending or descending order and whether numeric values are to be placed in signed or absolute order.

Field Qualification

Database field references in the various SQL clauses can include or omit a file qualifier. Field qualification establishes which file should be used for the source of a particular field. Notice in the following example that the PRDNO field is qualified in the SELECT and JOIN clauses.

```
SELECT prdno.1,descp,quano,actsp,quano*actsp NAME(amount)
      FROM ordline, partmast
      JOIN prdno.1=prdno.2
      ORDER BY prdno
```

The JOIN clause uses the product number from the first file (ORDLINE) to access records with matching part numbers in the second file (PARTMAST).

If you don't qualify a field, Sequel searches for it by looking in your SELECT clause and then in each file of your FROM clause until it is found. Though rarely required, field qualification can help make your statement more understandable. By qualifying a field name with its file name (or number) you make it easy for someone else to "read" your query.

Note: You should qualify any reference to a field that can be found in more than one of the files in your FROM clause.

You qualify a field by following its name with a period (.) and the file or correlation name of one of the files specified in the FROM clause.

Sequel allows both field.file qualification and file.field qualification. Sequel validates your field specification by first attempting to use the second part as a file qualifier. If unsuccessful, it tries again, this time using the first part as a file qualifier. If unsuccessful in both cases, Sequel will issue an error message.

You can also qualify a field using its file number. Use this method of qualification by following the field name with a period and the index of the file (1,2,3,etc.) in the FROM clause. Using file number qualification requires less typing and often makes SQL statements more readable. For example:

```
CUSNO.CUSTMAST
CUST.CUSNO
ORDHEAD.ORDNO
ORDNO.H
CUSNO.1
PRDNO.3
```

If you are creating a joining query and a file is used more than once in the FROM clause, you cannot use *file name* qualification. Using the file name won't indicate *which instance* of the file you mean. You must use file number or correlation name qualification. The explanation of the FROM and JOIN clauses in the following sections will provide more information on joining files together.

External Values

Several "special" fields can be used in your query to acquire information about the current operating environment.

CURRENT DATE
CURRENT TIME
CURRENT TIMESTAMP
CURRENT TIMEZONE

These fields return the current System i system values for date, time, and time zone. The data type and length of the returned values match the data type implied by the name; date, time, or time stamp. CURRENT TIMEZONE is returned as a time duration a packed decimal (6,0) value indicating the difference in hours, minutes, and seconds between the computer's local time and Universal Time Coordinated (UTC). This value is the same as the system value QUTCOFFSET.

When the query is run, the appropriate value is retrieved from the current machine information. The value is accessed only one time and will not change as the query progresses. Each row retrieved by the query will have the same value for the indicated field.

USER

The name of the user profile executing the query is returned as a varying length string up to eighteen characters long. The value is the same for each row retrieved by the query.

Note: The external user profile name will be referenced instead of any database field named USER unless the database field specification includes qualification with a file or correlation name. Thus, `SELECT user, dbfile.user FROM dbfile` will select both the user profile and the database field.

CURRENT SERVER

The name of the DRDA (Distributed Relational Database Architecture) node that is processing the Sequel requests is returned as a varying length string up to eighteen characters long. The value is the same for each row retrieved by the query. The CURRENT SERVER name must have been previously defined for the system using the Work With Relational Database Directory Entries (WRKRDBDIRE) command. Refer to the IBM *Distributed Relational Database Guide* for more information about DRDA.

ROWID
ROWNUMBER

This field returns the record number of the physical record being accessed. The ROWID/ROWNUMBER field may include a file qualifier. It can be used in any clause of the query statement, including the WHERE and JOIN clauses.

File qualifiers for ROWID/ROWNUMBER are not allowed in the JOIN clause. If you want to refer to a ROWID value from a specific file, you must qualify the ROWID field in the SELECT

clause, give it a different name, and then reference the correctly named field in the JOIN clause. See example 2 below.

A database field of the same name will be preferentially accessed instead of the row number value. Unlike the USER example above, field qualification cannot be used to differentiate between the external and database value. The record number of a file containing a ROWID field can only be accessed by specifying ROWNUMBER. The record number of a file containing a ROWNUMBER field can only be accessed by specifying ROWID.

The ROWID value is especially useful when using *direct files*. These files reference a row of a related file by record number, rather than value. Direct files can be joined together by referencing a database field containing a record number as the "from-field" and the ROWID of the target file as the "to-field" in a join clause.

ROWID can also be used to select specific records from a file. The following example selects every tenth record in the CUSTMAST file.

```
SELECT rowid,*.1 FROM custmast
      WHERE rowid MOD 10 = 0
```

Example

The example below references all of the special external values and returns them along with the rows from the CUSTMAST file. The second example joins two files by record number, rather than field value.

```
SELECT ROWID, USER, CURRENT DATE, CURRENT TIME,
          CURRENT TIMESTAMP, CURRENT TIMEZONE,
          CURRENT SERVER,*.1
      FROM custmast


SELECT ROWID.1 NAME(row1), cusno.1, cname.1,
          ordno, cusno.2 NAME(cus2), ROWID.2 NAME(row2)
      FROM custmast, ordhead
      JOIN row1=row2
```

Null Values

In some cases, a field value is not known or is unavailable (or simply doesn't make sense) for a given record. Placing a value (like blank or zero) into these fields is misleading and can sometimes be quite troublesome. The System i allows a special "place holder" to be set into a field when its value is unknown, unavailable, or inappropriate. This is called the *null value* and may be used to indicate just such a circumstance.

For instance, consider an employee file containing one record for each employee of the organization. The `MGRNO` field is supposed to indicate an employee's manager by indicating the manager's employee ID. This works well in every case except the CEO or Chairman. Since they have no superior, the null value is a perfect choice for the manager field as it clearly indicates that a specific value is unavailable or inappropriate.

Likewise, the salary field for a new employee might be undetermined when their record is entered into the database. A specific number will be known (and entered) sometime in the future. A value of zero would be as inappropriate as any other specific value because the employee is surely not working for free! The null value is precisely what is needed to indicate that the value is still unknown.

Sequel's output commands will show a special value to indicate that a field contains the null value. DISPLAY, PRINT, and REPORT output will show "n/a" ("¬" for fields less than 3 characters) for fields that are null.

Not all fields allow null values to be present. In order to accept a null value, the file must have been created after Version 2 Release 1.1 of the operating system was installed and the DDS keyword necessary to allow nulls (ALWNULL) must have been specified for each null capable field.

When EXECUTE is used to create an output file or OPNSQLF creates an open data path, the allow null (ALWNULL) attribute of each selected field is preserved in the created format. An expression involving one or more fields with an ALWNULL attribute creates a field with the ALWNULL attribute. That is, expressions involving fields that allow null values will produce fields that also allow null values. In addition, several functions (i.e. SUM, AVG) create columns that have the ALWNULL attribute. The ALWNULL attribute can be set explicitly by coding DFT(NULL) for a column within the SELECT clause. See page 1-55 for details.

Note: An ALWNULL(*NO) parameter value specified on the EXECUTE and OPNSQLF commands will override the ALWNULL attribute for fields in the created record format. If ALWNULL(*NO) is specified on an EXECUTE or OPNSQLF request, none of the fields in the format will have null capability, even though the view would normally create null capable fields.

If null values appear in fields that are used for ordering, records containing null values will be at the end of an ascending list. That is, the null value has the highest value in the collating sequence; higher than the letter "Z" and the number "9".

Null Values in Record Selection Tests

The presence of null values complicates record selection by introducing the possibility of an "unknown" result in comparison tests. Without null values, each test in the WHERE or HAVING clause of the query will return either a *true* or *false* result. Since a null value implies a degree of uncertainty, a third type of result, called *unknown*, occurs when null values are considered. If either (or both) of the comparison operands has a null value, the test will return *unknown* instead of *true* or *false*.

Note: The complete WHERE or HAVING clause must still evaluate "true" before a given record can be selected by the query.

For instance, records with a null salary value will not be chosen by either of the following selection tests:

```
WHERE salary>50000
WHERE NOT salary>50000
```

The result of the "greater than 50,000" test is *unknown* for each null salary value. In the first case, records with null salary values are not selected because the comparison is not yield a *true* result. Records with a null salary value are not chosen in the second case because the comparison would have to yield *false* in order for the NOT operator to create a *true* result.

Testing a field or expression for the null value can be accomplished using the IS NULL or IS NOT NULL comparison. Refer to the description of the WHERE clause beginning on page 1-64 for more information and examples.

Calculations and Expressions

Sequel allows you to create calculated results in your views and to use expressions in selecting records from the database. Calculations can be used in specifying both the data to be returned (SELECT clause), and in choosing which records will be included (WHERE and HAVING) as well.

Sequel calculations (expressions) are formulated using mathematical notation and allow a wide range of operators and functions. A calculation can be as simple as a single numeric or character constant. They can also become quite complex; involving many operands, operators, and functions.

This section of the manual will describe each of the operators and Sequel functions available to you. It will show examples of SQL statements that use the functions so you can see how they look. If you are not already familiar with the basic structure of a Sequel statement, you may want to skip ahead to the sections beginning on page 1-51 that describe each clause in detail.

Derived Fields

Using a math or function operator within the SELECT clause creates a *derived field* within the view. This field will have attributes (data type and length) that depend upon the nature of the operations involved.

The query processor automatically creates fields so that they are large enough to hold the result of the expression. You can override the total length, or the number of decimal places (if the result is numeric). Sequel will truncate character results and drop insignificant decimal digits (to the right of the decimal point) if the field is too small to hold the result. Data mapping errors will occur if a numeric result has too few places to the left of the decimal point and significant digits are lost.

The derived field is assigned a default name of "DERIVED_xx" by Sequel. The xx is a sequential number that begins at 01 for the first field and is incremented for each calculated field in the view. This field name can be referenced in another portion of the SELECT or in the WHERE clause. You will usually want to rename the field using the NAME attribute, and to specify length or editing attributes discussed later in this section.

Constants

Constants, sometimes called literals, can be placed in expressions as long as they conform to traditional format. Strings must be surrounded by quotation marks (single or double) and embedded quotation marks of the same type as the enclosing marks must be doubled. For example, `"Bob's Diner"` and `'Bob''s Diner'` are equivalent representations of: Bob's Diner.

Numeric constants are represented in conventional decimal or floating point notation.

Date, time, and timestamp constants must be enclosed in quotation marks and must appear in either a valid SAA format or in the preferred format of the query. See the section beginning on page 87 for more about the various date formats.

Constants need not be included in expressions in order to appear in the view. Simply placing the constant within the field list will cause it to be presented when the view is run. For example,

```
SELECT "Name:" colhdg(" "),cname,"State:" colhdg(" "),cstte
      FROM custmast
```

will create a view with showing "Name:" to the left of each customer and "State:" to the left of each state field.

Operators and Functions

Sequel allows you to use several numeric and character operators in constructing a wide variety of expressions. Within an expression, all fields and constants must have the same data type (character or numeric) and their data type must be consistent with the operators being used.

An expression is processed in left to right order subject to the rules of precedence. Within an expression, calculations inside parentheses are evaluated first, exponentiation is done next, then multiplication, division and remainder, and finally addition and subtraction.

Refer to the tables below for a list of the allowed operators and their meaning.

Numeric Operators		Character Operators	
+	Addition	CAT	Concatenation
-	Subtraction	||	Concatenation
*	Multiplication		
**	Exponentiation		
/	Division		
MOD	Remainder		

Exponents

The exponential operator (**) raises a number to a power. A floating point number will result from the operation. The LEN attribute can be used to force the result into standard decimal notation.

Some examples:

```
10**3 = 1000     (10 * 10 * 10)
10**2 = 100      (10 * 10)
10**1 = 10
10**0 = 1
10**(-1) = 0.1
10**(-2) = 0.01
5**3 = 125
4**2 = 16
0**3 = 0
0**0 = 1
```

MOD - Remainder

This operator returns the remainder of a division of the first operand by the second. It is especially useful when you are trying to extract part of a numeric field. When used in conjunction with subtraction and division, the MOD operator can be used to create the numeric equivalent of the substring (SUBSTR) function.

Note: You may find that using the DIGITS function in conjunction with the SUBSTR function is easier than using the MOD operator for numeric extractions. The difference between them is that the MOD expression returns a numeric result, while the SUBSTR result is always character. (SUBSTR results can be "re-cast" as numeric with the ZONED function)

The examples below show the use of MOD to extract parts of a MMDDYY date.

```
122509 MOD 100 = 09
122509 MOD 10000 = 2509
122509-(122509 MOD 10000)=120000
122509 MOD 10000 - 122509 MOD 100=2500
```

The first example extracts the last two digits - the remainder that results when the date is divided by 100. This yields the year portion of the date.

The second sample returns the last four digits of a number. In this case, the day and year part of the date. This is the remainder when the date is divided by ten thousand.

The third example uses subtraction to remove the DDYY part from the date field, leaving MM0000. This value can be divided by ten thousand in order to acquire just the month value.

The final example shows how to extract just the day part of a date. Notice that the central two digits are retrieved by subtracting the year part (date MOD 100) from the DDYY part (example 2). The result can be divided by 100 to yield the day value.

Tip: Use the Convert Date (CVTDATE) function to create a "date" data type field from a numeric field. Use date operations and functions on the result to get the final results that you need. See the section beginning on page 99 for more information.

Arithmetic Operators

One particularly useful and interesting applications for the numeric operators involves combining separate numeric fields containing date information into a single value. Numeric fields can be "concatenated" into a numeric result by using multiplication and addition operators.

The order date is stored in the order header (ORDHEAD) file as three separate fields. We wish to bring them together again and display the date in yy/mm/dd format (with editing) and present records in descending year/month/day order so that newest transactions are listed first. The query below will accomplish both of these tasks.

```
SELECT cooyr*10000+coomn*100+coody EDTCDE(Y) LEN(6,0)
          NAME(date),cusno,cname,cuspo,shipv,trmds,orval
      FROM ordhead,custmast JOIN cusno.1=cusno.2
```

```
ORDER BY date DESC
```

Dates can be rearranged from one format to another by using the CVTDATE function to convert the numeric fields to a date data type field and then using the CHAR function, or by combining arithmetic operators with the MODulus operator.

A well known "programming trick" that involves multiplication by the value 10000.01 does not work with Sequel, because a numeric overflow error, induced by the calculation, causes records to be skipped.

Some examples of date conversion appear below. The LEN attribute is required because each computation creates a result with digits to the right of the decimal point. The second example in each set creates a correct integer result.

Convert MMDDYY into YYMMDD:

```
(Date MOD 100)*10000+Date/100 LEN(6,0)
(Date MOD 100)*10000+(Date-Date mod 100)/100
```

Convert YYMMDD into MMDDYY:

```
Date MOD 10000*100+Date/10000 LEN(6,0)
Date MOD 10000*100+(Date-Date MOD 10000)/10000
```

Concatenation Operator

The CAT operator (also specified as ||) is used to concatenate (bring together) two or more character or numeric fields into a single field. The length of the result field is the sum of the lengths of the individual fields. Blanks at the end of one field are not suppressed prior to concatenation. The CAT operator is equivalent to the CAT function. (see page 1-36)

Examples:

```
"John " CAT "Smith"    = "John Smith"
firstnm || lastnm      = "Cheryl       Close         "
state || " " CAT zip   = "IL 60173"
hour || ":" || minute || ":" || second    = "12:37:42"
SUBSTR(date,1,2) CAT "/" CAT SUBSTR(date,3,2)
      CAT "/" CAT SUBSTR(date,5,4)        = "10/04/2009"
CAT("Customer Number: ", cusno) = "Customer Number: 100200"
```

Sequel Functions

There are several built-in functions that you can use in creating your expressions. Sequel functions can be classified as either grouping or non-grouping functions. *Grouping functions*, also

called column functions, return a value by working on a set of records. *Non-grouping functions* operate on fields within a record in order to return a value.

Functions require either a single operand (e.g. ABS, SQRT, ATANH) or a comma separated list of several operands (e.g. SUBSTR). Function operands can be either field names, constants, or mathematical expressions. Expressions can mix functions with other functions, literals (numbers or alphabetic strings), and fields.

Several examples showing a variety of Sequel expressions are included below.

Character Functions

Sequel provides several types of non-grouping functions that can be used with character fields.

Uppercase conversion	(UPPER)
Lowercase conversion	(LOWER)
Proper case conversion	(PROPER)
Center conversion	(CENTER)
Trim	(LTRIM, RTRIM, TRIM, STRIP, STRIPX)
String location	(POSSTR)
String verification	(VERIFY, INDEX)
Length	(LENGTH)
Varying length	(VARCHAR)
Translation	(TRANSLATE, XLATE)
Conversion3	(ZONED, DECIMAL, INTEGER, FLOAT, CHAR2NUM)
Hexadecimal conversion	(HEX)
Logical (bitwise) manipulation	(LAND, LOR, LXOR, LNOT)
Phonetic functions4	(SOUNDEX, DIFFERENCE)
Split character value	(SPLIT)
Unedit	(UNEDIT)

Uppercase Conversion

Character comparisons are frequently difficult because you may not be sure of the lower and upper case mix within a field. Is the first letter capitalized, or are all letters? Some? None? Often, you may not know.

One solution to this problem is to convert the field to all upper case prior to the comparison. If the comparison field is specified in upper case, the problem disappears!

The query below uses the UPPER function in the WHERE clause to convert the field CNAME to upper case prior to comparing it:

```
SELECT * FROM custmast WHERE UPPER(cname) CONTAINS "KMART"
```

The UPPER function, performs a case conversion from lower case letters (a-z) to upper case (A-Z). It works by performing the bitwise OR operation between the specified field and a string of blanks. It can be used in the SELECT, WHERE or HAVING clauses.

The UPPER function accepts a single field name that must be placed in parentheses following the function. It is perhaps most useful in the WHERE clause where it can be used to convert fields for comparisons, although it can be placed into the SELECT clause as well. For instance:

```
SELECT UPPER(cname) Name(upcase) FROM custmast
      ORDER BY upcase
```

presents the customer list in order according to their upper case sorting order.

Lowercase Conversion

In some applications, character data may be stored in all uppercase. In some instances, you may prefer all or part of the field to be presented in lowercase. Especially in the case of proper names, you may prefer to see the data in mixed case with the first letter capitalized followed by lower-case letters.

The LOWER functions accepts a single field name that must be placed in parentheses following the function. The LOWER function, performs a case conversion from upper case letters (A-Z) to lower case (a-z). It works by performing the bitwise OR operation between the specified field and a string of blanks. It can be used in the SELECT, WHERE or HAVING clauses.

The query below will convert a two-digit state abbreviation to lower case:

```
SELECT LOWER(cstte) name(low) FROM sequelex/custmast
```

In cases where only part of the field should be converted to lower case, LOWER can be used in conjunction with the substring and concatenation functions. If proper names are stored in all uppercase and JANE should be Jane, the following expression could be used:

```
CAT(SST(fname,1,1),LOWER(SST(fname,2,9)))
```

PROPER Function

In some applications, character data may be stored in all uppercase. In some instances, you may prefer the field to be presented in proper case where the first letter is capitalized followed by lowercase letters.

```
PROPER(expression)
```

The PROPER function accepts a single field name that must be placed in parentheses following the function. The PROPER function performs a case conversion on the first letter of each word to upper case letters (A-Z). It can be used in the SELECT, WHERE or HAVING clauses.

```
PROPER("MARY JONES") = Mary Jones
```

CENTER Function

In most applications, character data is left justified. In some instances, you may prefer the data to be centered based on the length of the field.

```
CENTER(charexp, len)
```

The CENTER function accepts a field name and the length of the resulting field. Leading blanks will be added to force the value to be centered. It can be used in the SELECT, WHERE or HAVING clauses.

```
CENTER("PCE Corp.",25) = '        PCE Corp.       '
```

TRIM Functions

Sequel provides a capability to remove repetitive occurrences of a given character from the leading, trailing, both ends, or from the middle of an alphanumeric column. The functions, known collectively as the trim functions, operate on either fixed or varying length string columns and return a varying length string.

The syntax for three of the trim functions is the same:

```
LTRIM(expression,character)
RTRIM(expression,character)
TRIM(expression,character)
```

Each of the functions removes the specified *character* from the column created by *expression*, working from the left, right, and both ends of the string respectively. If a character is not supplied, a blank is assumed and the leading and/or trailing blanks will be removed from the expression.

Alternatively, you may choose to specify the STRIP function. It offers the same function as the trim functions, but with an alternative syntax. It is included for compatibility with SQL/400 syntax.

```
STRIP(expression,type,character)
```

The *type* is optional but must be specified if a character is supplied. If specified, the type value indicates the method of character removal and must specify L, LEADING, T, TRAILING, B, or BOTH. As with the trim functions, if the final operand (the trim *character*) is omitted, a blank is assumed and the leading and/or trailing blanks will be removed from the expression.

The STRIPX function provides the capability to remove repetitive occurrences of a given character from anywhere in the field.

```
STRIPX(expression, character)
```

Both arguments of the expression are required. The expression is the character field or string and the *character* indicates character to be omitted.

Examples:

Function reference and result	Effectively Removes
`RTRIM("1000","0") = "1"`	trailing zeros
`RTRIM("10010","0") = "1001"`	trailing zeros
`LTRIM(".....Test  ",".") = "Test"`	leading dots
`RTRIM(LTRIM(".....Test  ",".")) = "Test"`	leading dots then trailing blanks
`TRIM("  John Doe  ") = "John Doe"`	leading and trailing blanks
`STRIP("0001234500",B,"0") = "12345"`	leading and trailing zeros
`STRIP("  John Doe  ") = "John Doe"`	leading and trailing blanks
`STRIPX("John Doe"," ") = "JohnDoe"`	all blanks in the string

String Location

Sequel gives you a way to find the position of a character sequence within a string. The POSSTR* (position of string) function returns the location of the first character of a search string within a source expression. If the search string cannot be found, the function result is zero. The syntax of POSSTR is:

```
POSSTR(search-expression, find-expression)
```

The POSSTR functions has two arguments. The first one is the expression to be searched, the second is the find expression. Either argument can be a literal string, a field reference, or an expression derived from combinations of fields and literals.

`POSSTR("Little red wagon","red") = 8`	
`POSSTR(address,state)`	returns the location of the state field within the address field
`POSSTR(UPPER(cname)),"LTD")`	returns the location of the characters LTD within the name field, regardless of their case (upper/lower) in the field

String Verification

The INDEX and VERIFY functions are similar to the POSSTR function in that they search a string. They are different in that they return results based on whether the string contains any of a class of characters provided by the search string. INDEX finds the location of the first character that is in the class string. VERIFY returns the location of the first character that is not in the class string. If no characters in the search string satisfy the criteria, the result is the length of the string plus one.

```
INDEX(search-expression, class-string)
VERIFY(search-expression, class-string)
```

Like POSSTR, both functions have two arguments. The first one is the expression to be searched, the second is the class string. The first argument can be a literal string, a field reference, or an expression derived from combinations of fields and literals. The second must be a string literal; field references and expressions are not allowed.

```
VERIFY("6452ABC,"0123456789.") = 5
```

Finds the first character not in the class (not a numeric digit).

```
INDEX(RTRIM(address)," ,.")
```

Returns the location of the first blank, comma, or period

within the address field. If no blank, comma, or period can be found, the result is the length of the address string.

Varying Length String

The VARCHAR function creates a varying length field and allows you to specify its maximum and allocated length. Because the output from the DISPLAY, PRINT, and REPORT shows all varying length results as fixed length columns with the maximum defined length, the usefulness of this function is limited to EXECUTE and OPNSQLF commands that require varying length results. Syntax for the function is:

```
VARCHAR(expression,max,alloc)
```

The expression must yield an alphanumeric result. The function will return a column with a maximum length matching the specified maximum. If an allocated length is specified, it will also be used in the column definition.

Examples:

```
VARCHAR(cname,25,10)
VARCHAR("Sample Result", 20)
```

Length Function

The LENGTH function returns the length of a character expression. The result of the function is a 4 byte binary value. If the expression creates a fixed length field, the function result will be constant for each retrieved row. If the expression creates a varying length field, the result of the LENGTH function will reflect the actual size of the expression result.

Examples:

Function and result	Explanation

`LENGTH(DIGITS(DECIMAL(123,5,0)))=5` Literal value 123 is converted to packed decimal (5,0) then to character string of length 5

`LENGTH(LTRIM(DIGITS(DECIMAL(123,5,0),"0"))=3` 5 character (above) has leading zeros removed leaving string of length 3

String Translation

Two translation functions are available that allow you to translate characters within a string expression. The XLATE function lets you reference a System i translation table. The TRANSLATE function lets you specify the characters to search for and the characters you want them translated to.

```
TRANSLATE(expression, [to-string, [from-string, [pad-character]]])
XLATE(expression, translate-table)
```

Using TRANSLATE, characters in the expression are translated one at a time by searching the from string. If the character is found, the corresponding character in the to string is substituted. If the to string is shorter than the from string and a corresponding character does not exist, the pad character is used as a replacement character. The result is a character string of the same length as the initial string.

`TRANSLATE(" 123 45 6", "0", " " , " ") = "012304506"`
changes blanks "inside" a numeric field to zeros

The XLATE function translates characters in a source expression, using a translation table. Specify the expression and the translation table name as function arguments. The table can be fully qualified with a library name or *LIBL, or left unqualified. The translation table is located when the view is created and compiled into the view. It is not referenced at runtime.

`XLATE(charfld, QSYS/QASCII)`
translates characters in the charfld field to ASCII using the system translation table.

Hexadecimal Conversion

Occasionally it is important to be able to determine the bit pattern value of a field or character expression. The SEQUEL HEX function converts a alphanumeric value to a hexadecimal (base 16) string. The result is a string that is twice as long as the character expression. The string is <u>not</u> prefaced with an 'X' to denote its nature as a hexadecimal string.

Examples:

```
HEX("ABCDEFG") = "C1C2C3C4C5C6"
HEX("123")="F1F2F3"
```

Logical (bitwise) Functions: LAND, LNOT, LOR, LXOR

Four bit manipulation functions are also available for character operations. While not generally useful, they can be applied in specific cases where needed. As implied above, the UPPER function is simply a special case of the OR function.

The LAND, LOR, LNOT, and LXOR functions accept character operands and perform the bitwise AND, NOT, OR and exclusive or (XOR) on them.

Up to 98 operands can be processed in a single function request. At least two are required. Separate the operands with a comma. If the operands are not all the same length, the shorter ones are padded on the right with enough blanks to make them the length of the longest operand prior to performing the operation. The result of the function is a character field with a length equal to the length of the longest operand specified in the function.

Examples:

```
LOR(cname," ") = uppercase cname value
LOR("  12","0000") = "0012"
LAND("12345",x"FFFFF0FFFF") = "12045"
HEX(LNOT(x"01234567890ABCDEF")) = "FEDCBA9876543210"
```

Phonetic Functions

The SOUNDEX and DIFFERENCE functions return values based on the phonetic sound of a character value. They can be used on the SELECT, WHERE or HAVING clauses.

SOUNDEX returns a 4 character value representing the English "sound" of the expression. Similar sounding expressions will have similar values. SOUNDEX can be used on the WHERE to select records with similar sound or the result of the SOUNDEX function can be used on the ORDER BY clause to sort records based on their sound.

The return value begins with the character of the expression, and has a 3-digit value representing the sound of the remaining characters. Variations on the spelling of "Smith" all return the same value:

```
SOUNDEX("Smith")="S530"
SOUNDEX("Smithe")="S530"
SOUNDEX("Smyth")="S530"
SOUNDEX("Smythe")="S530"
```

To select only the records that sound like "Smith", SOUNDEX can be used in the WHERE clause:

```
WHERE SOUNDEX(field)="S530"
```

Example:

To return only the Kmart customers that include Kmart Eastern Region, Kmart Western Region and Kmart Midwest:

```
SELECT cname
      FROM sequelex/custmast
      WHERE SOUNDEX(cname)="K563"
```

DIFFERENCE returns an integer, 0 through 4, which represents the relative phonetic difference between two character expressions. The more alike the two expressions are spelled, the higher the number DIFFERENCE returns. If the character expression are spelled very similarly, DIFFERENCE returns a 4. For two character expressions with little in common phonetically, DIFFERENCE returns 0.

Example:

DIFFERENCE is useful for searching a file when the exact spelling of an entry is not known. To find the customer from the SEQUELEX/CUSTMAST file whose name sounds like "Detzen":

```
SELECT cname
      FROM sequelex/custmast
      WHERE DIFFERENCE(cname,"detzen")=4
```

The record for Dietzgen Donuts, Inc. will be returned.

SPLIT Function

The SPLIT function separates a character value based on the location of the first position of a string. If no part of the character string can be found, the original character value is returned.

```
SPLIT(search-expression, find expression, int)
```

The SPLIT function has three arguments. The first one is the expression to be searched, the second is the find expression, and the third is optional to denote which occurrence of the find expression the separation should begin. The first two arguments can be a literal string, a field reference, or an expression derived from a combination of fields. The third argument should be a number.

```
SPLIT("Little red wagon"," ") = "Little"
SPLIT("Little red wagon"," ",2) = "red"
SPLIT("Little red wagon"," ",3) = "wagon"
```

A length (LEN) should be assigned to each derived field created with the SPLIT function. Otherwise the result field will be very long.

CHAR2NUM Function

The CHAR2NUM function will right adjust a left justified string by adding the appropriate number of leading zeros. This function is useful when numeric values are stored in a character field and the data has leading and/or trailing blanks. It is generally inside of the DECIMAL function.

```
CHAR2NUM(expression, length)
```

The CHAR2NUM function has two required arguments. The *expression* indicates the character field or string to be converted. The *length* represents the character length of the result field.

```
CHAR2NUM(" 12  ",5) = "00012"
```

UNEDIT Function

The UNEDIT function receives a varying length character string and removes the non-numeric characters from it. The result is a varying length string up to 33 characters long containing only a leading minus sign (if the input had a leading/trailing '-' or a trailing CR) and the numeric characters found in the input and a decimal separator if the original string contained one.

```
UNEDIT(character expression [, decimal separator])
```

The first argument is a character string up to 100 positions long. The second optional argument can be specified to control the decimal separator character. If not supplied, the current job's decimal separator will be used.

```
UNEDIT("123AB456CD78EF") = "12345678"
UNEDIT("-123AB456.78CD",".") = "-123456.78"
```

Numeric Functions

Sequel provides a wide variety of non-grouping functions that work with numeric fields, literals and expressions. Most are beyond the scope of normal business calculations and will be used only in highly specialized applications.

Rounding	(ROUND, FLOOR, CEIL)
Absolute Value	(ABS)
Conversion to character	(CHAR, DIGITS)
Conversion to numeric	(ZONED, DECIMAL, INTEGER, FLOAT, SMALLINT, BIGINT)
Exponent/Logarithm	(EXP, EXP10, LOG, LN, SQRT)
Trigonometric	(ACOS, ASIN, ATAN, ATAN2, ATANH, COS, COSH, COT, SIN, SINH, TAN, TANH)
Intra-record total/average	(SUM, AVG)
Random	(RAND)
Sign	(SIGN)
User defined	(ACCUM, EDIT, PCTCHG, SIGN, UNPACK)

Rounding Functions

Unlike the report writer, calculations performed by a Sequel view are not automatically rounded into the result. Sequel has a several rounding functions that make it easy to round calculation results.

```
ROUND(expression, decimal-digits)
FLOOR(expression, decimal-digits)
```

```
CEIL(expression, decimal-digits)
```

The ROUND function rounds an expression result to the number of places that you specify. You can round to the right of the decimal point by using a positive number of digits from 1 to 8. Round to the left of the decimal point by using a negative number. If you don't specify the number of decimal places, a zero will be assumed and an integer result will be returned. For instance,

```
ROUND(12.499,1) = 12.5
ROUND(12.499) = 12
ROUND(12.499,-1) = 10
```

CEIL and FLOOR functions round up or down to the number of places that you specify. Unlike ROUND which uses a "halfway point" of 5, these functions work on any non-zero value below the indicated precision. CEIL rounds away from zero, and FLOOR rounds towards zero. For instance,

```
CEIL(12.0001) = 13          FLOOR(12.9999) = 12
CEIL(12.0001,1) = 12.1      FLOOR(12.9999,1) = 12.9
CEIL(-12.0001) = -13        FLOOR(-12.9999) = -12
CEIL(342.99,-1) = 350       FLOOR(342.99,-1) = 340
```

CEIL, FLOOR, and ROUND work by using binary integers. Therefore the length of the result value is limited to nine total positions.

Absolute Value

The absolute value function (ABS) accepts a single numeric field, constant, or expression and returns its unsigned value. The data type and length of the result matches that of the argument. Refer to the following examples:

```
ABS(1000) = 1000
ABS(-500) = 500
ABS(0)    = 0
```

Conversion to Character

There are two ways to convert a numeric expression to a character result. The CHAR function should be used when you want a left justified result with leading zeros removed and a decimal point inserted when needed. Use the DIGITS function when you want a fixed length result with leading zeros and no decimal editing.

```
CHAR(expression)
DIGITS(expression)
```

The CHAR function (also used to convert date, time, and timestamp values to character) accepts two arguments: the numeric expression, and an optional decimal point character. If a decimal point is not specified the job's default decimal point will be used. The length of the result string is the total length of the numeric field, plus one for the sign, and one more for the decimal point (if the field is not an integer).

Examples:

```
CHAR(1234.56) = "1234.56"
CHAR(-12345) = "-12345"
CHAR(987.654,"?") = "987?654"
```

The DIGITS function accepts a single numeric field (binary, packed, or zoned) and returns a character string representing its unsigned value. No sign or decimal point is included in the result. The length of the result string is the total length of the numeric field if the field is packed or zoned. If the numeric field is a 2 byte binary field (or a integer literal less than 10000), the result is 5 characters long. If the numeric field is a 4 byte binary field (or a integer literal greater than 9999), the result is 10 characters long. Zeros fill the leading positions of the result as necessary.

Examples:

```
DIGITS(1234) = "01234"
DIGITS(12345) = "0000012345"
DIGITS(DECIMAL(12345,5,0)) = "12345"
DIGITS(-500) = '500'
DIGITS(3.14) = "314"
```

Conversion to Other Numeric Forms

Both character and numeric expression results can be "cast" into a specific type of representation with any of several Sequel functions. This can be especially valuable when fields created by the EXECUTE command must have a specific data type or internal coding.

`DECIMAL(expr[,len[,dec]])`	$1 \leq len \leq 31$, $0 \leq dec \leq len$	default 15,0
`ZONED(expr[,len[,dec]])`	$1 \leq len \leq 31$, $0 \leq dec \leq len$	default 15,0
`INTEGER(expr[,len])`	len = 2 or 4	default 4 bytes
`FLOAT(expr[,len])`	len = SINGLE or DOUBLE	default DOUBLE
`SMALLINT(expr)`	len = 8	default 8 bytes
`BIGINT(expr)`	len = 18	default 18 bytes

Each function forces the expression result to have the indicated format and the specified length (and precision) attributes. Reduction in the precision of the expression truncates (does not round) the non-integer portion of the value. The difference between DECIMAL and ZONED format lies in their internal coding; no external difference is apparent to the user.

SMALLINT, INTEGER, and BIGINT support 8 byte binary values including translating the 8 byte binary values to a PC formatted result.

The expression specified as the first operand of the function must create a valid numeric value.

The length (and precision) may be omitted. The default values indicated above will be used if specific values are not provided. Refer to the following examples.

Function reference and result	Effect
`ZONED(SUBSTR(charfld,1,5),5,3)`	First 5 characters of charfld are converted to zoned (5,3) value ("12345ABCD" becomes 12.345)
`ZONED(DIGITS(123456),10,6) = 0.123456`	DIGITS converts literal integer 123456 to a 10 place character before ZONED function casts it as a (10,6) numeric
`DECIMAL(mtdsls/totsls,7,4)`	result is truncated to packed (7,4) value
`INTEGER(mtdsls/totsls)`	Result is truncated to an integer
`FLOAT(numfld,SINGLE)`	Numfld is converted to single precision floating point (1000 becomes 1.0000000E+003)

The LEN attribute (page 53) should not be specified in addition to the casting function, as it will cause a final transformation of the result to packed decimal format. For instance,

```
SELECT ZONED(amtdu/crlim,10,5) NAME(pct) LEN(10,5)
```

will force the result into packed decimal form rather than the zoned decimal form requested by the ZONED function.

Exponent and Logarithm Functions

These functions require one numeric argument. The result is always a floating point number, but it can be converted to fixed decimal notation using the LEN attribute.

The exponential operator (**) can be used to raise a number by a factor. Two special functions offer an easy way to let you raise 10 and the irrational value *e* (2.71828..) exponentially.

```
EXP10(3) = 10**3 = 1000
EXP(3)   = 20.085537...
```

The logarithm function "reverses" exponentiation, and returns the root of a number. Sequel provides base 10 and base e logarithm functions.

```
LOG(1000)     = 3
LN(20.085537) = 3
```

The square root function (SQRT) returns the number which must be multiplied by itself to result in the argument value.

```
SQRT(16)  = 4
SQRT(100) = 10
```

Trigonometric Functions

These functions are useful primarily for engineering applications. They return double precision floating point results. The argument values and results are always in radians. Each function requires a single numeric argument and accepts either a field value, literal (constant), or an expression.

The function result is null capable if the argument is null capable. The result is null only if the argument is null.

```
SIN(expression)
```

Sine of radian argument. The SIN and ASIN functions are inverse operations. The result is in the range of -1 to 1.

```
COS(expression)
```

Cosine of radian argument. The COS and ACOS functions are inverse operations. The result is in the range of -1 to 1.

```
TAN(expression)
```

Tangent of radian argument. The TAN and ATAN functions are inverse operations.

```
COT(expression)
```

Cotangent of radian argument.

```
ASIN(expression)
```

Arc sine of the argument returned in radians. The ASIN and SIN functions are inverse operations. The value of the expression must be in the range of -1 to 1. The result is in the range of -p/2 to p/2.

```
ACOS(expression)
```

Arc cosine of the argument returned in radians. The ACOS and COS functions are inverse operations. The value of the expression must be in the range of -1 to 1. The result is in the range of 0 to p.

```
ATAN(expression)
```

Arc tangent of the argument returned in radians. The ATAN and TAN functions are inverse operations. The result is in the range of -p/2 to p/2.

```
ATAN2(y,x)
```

Calculates the arc tangent of y/x. If both parameters of ATAN2 are 0, the functions returns 0. ATAN2 returns a value in the range-p to p radians, using the signs of both parameters to determine the quadrant of the return value.

Requires V4R5M0 of OS/400.

```
SINH(expression)
```

Hyperbolic sine of radian argument.

```
COSH(expression)
```

Hyperbolic cosine of radian argument.

```
TANH(expression)
```

Hyperbolic tangent of radian argument. The TANH and ATANH functions are inverse operations.

```
ATANH(expression)
```

Hyperbolic arc tangent of the argument returned in radians. The TANH and ATANH functions are inverse operations. The value of the expression must be in the range of -1 to 1.

Example:

```
SELECT leg1,leg2,
          SQRT(leg1**2+leg2**2) NAME(hypotenuse)
                COLHDG("Hypotenuse") LEN(3,0),
          ASIN(leg1/hypotenuse) NAME(angle1)
                COLHDG("Angle 1" "Radians")
          ASIN(leg2/hypotenuse) NAME(angle2)
                COLHDG("Angle 2" "Radians")
          ACOS(-1)/2-angle1-angle2 NAME(deviation)
      FROM triangles
```

The query above examines records in the file called TRIANGLES and computes the hypotenuse ($c^2=a^2+b^2$) and the two interior angles for the right triangles given the two leg lengths.

The length of the hypotenuse is defined to be three digits long with zero decimal places. The angles are returned in radians (double precision) by the ASIN function and the view then determines how accurate the angles are based upon the fact that the two interior angles in a right triangle must add up to 90 degrees (pi/2 radians). The value for pi (3.1415...) is acquired by using ACOS(-1).

Intra-Record SUM and AVG Functions

For convenience, Sequel includes two functions that allow you to total and average fields within a record. They are equivalent to arithmetic operations that you could use to accomplish the same purpose.

Note: Do not confuse these functions with the grouping functions. Although they have the same name, they work in an entirely different manner.

The intra-record sum and average functions allow you to specify a list of numeric fields (or expressions) which should be totaled, or averaged. Sequel translates the function request into a series of arithmetic operations.

In the two sets of examples below, the first example shows how to use the intra record function, SUM in the first set, AVG in the second. The second example shows the equivalent (and probably preferred) method which can be accomplished using standard arithmetic operators.

```
SUM(jan,feb,mar,apr,may,jun,jul,aug,sept,oct,nov,dec)
jan+feb+mar+apr+may+jun+jul+aug+sept+oct+nov+dec

AVG(jan,feb,mar,apr,may,jun,jul,aug,sept,oct,nov,dec)
(jan+feb+mar+apr+may+jun+jul+aug+sept+oct+nov+dec)/12
```

RAND Function

The RAND function returns a random number between 0 and 1. RAND returns the same sequence of random number for a given seed value each time it is run. It can be used on the SELECT, WHERE or HAVING clauses.

To achieve the most random sequence of numbers, issue:

```
RAND(MICROSECOND(CURRENT TIMESTAMP))
```

SIGN Function

The SIGN function returns the sign of the numeric expression. A 1,0 packed decimal field will be returned with the value of -1 if the expression is negative, 0 if is zero and +1 if is positive. The result is null if the expression evaluates to null. To achieve the most random sequence of numbers, issue:

```
SIGN(expression)
```

Example:

```
SIGN(-500) = -1
SIGN(500) = 1
SIGN(0) = 0
```

ACCUM Function

The ACCUM function accumulates values across retrieved rows. It is especially useful to create a running total or count within record groupings.

```
ACCUM(expression[,charexp])
```

ACCUM has one required argument and one optional argument. The first one is the numeric expression to be accumulated. The second denotes the character field to be used for the field grouping.

ACCUM can be used to create a running total of the amount due (AMTDU) by state, customer type, and grand total. By default, ACCUM returns a floating point number. The DECIMAL function can be used to display a more meaningful number.

```
SELECT ctype, cstte, cname, amtdu,
          DECIMAL(ACCUM(amtdu,cstte),11,2) NAME(statetot),
          DECIMAL(ACCUM(amtdu,ctype),11,2) NAME(typetot),
          DECIMAL(ACCUM(amtdu),11,2) LEN(11,2) NAME(total)
      FROM sequelex/custmast
      ORDER BY ctype, cstte
```

STATETOT results in a running or accumulative total for each state. TYPETOT creates an accumulative total for each customer type and TOTAL is a running total of all records.

EDIT Function

EDIT creates a character result while maintaining the specified edit code or edit word.

```
EDIT(dec,len,int,charexp)
```

The EDIT function accepts a single numeric field, length and decimal position of the numeric field, and the desired editing specified with an edit code or edit word and returns a character result.

```
EDIT(1234567,9,2,"J$") = $1,234,567.00
EDIT(1112223333,10,0,"   -   -    ") = 111-222-3333
```

PCTCHG Function

PCTCHG computes the percentage of change between two numbers.

```
PCTCHG(exp,exp)
```

The PCTCHG has two arguments. Each must be a numeric expression. The first argument is the denominator value. This function streamlines calculating percent of change and is the equivalent of `(X-Y)/Y*100`.

```
PCTCHG(200,100) = 50-
PCTCHG(100,200) = 100
```

UNPACK Function

The UNPACK function converts a packed or binary value inside a string to a numeric value. This function is especially useful when data is stored in a program described file and the numeric data is stored in a character field in packed decimal format or in binary integer form. The UNPACK function makes it possible to extract the numeric values from their stored form.

The UNPACK function as two forms. Pass three arguments (character data, length, decimal) to indicate that a packed decimal value needs to be extracted. Pass one argument (character data) to indicate that binary integer data needs to be extracted. UNPACK will extract either a two byte or four byte value, depending on the length of the argument.

```
UNPACK(charexp[, len[, dec]])
```

Example:

CFLD is a 9 position character field with the following value:

```
      12 34 5F 98 7F 07 5B CD 15

UNPACK(SST(cfld,1,3),5,2) = 123.45
UNPACK(SST(cfld,4,2),3,0) = 987
UNPACK(SST(cfld,6,4))     = 123456789
```

The first two examples identify the substring of the character field and the attributes of the decimal value that will be extracted. The last example specifies only a substring which identifies the first argument as binary integer.

VAL2WRD

This UDF converts numeric values into word strings.

`VAL2WRD(value, 'currency string', 'mask codes')`

The three parms are defined as follows:

Value - Any numeric amount (30,9) from 0.00000001 to 999999999999999999999.999999999. This can be a numeric literal, number variable or numeric field.

Currency String - A character string (30) enclosed in single quotes for the name of the currency used. The default is blank which infers no currency is specified in the output.

Mask Codes - Broken into Types A, C, G, H, J, L, M, P, S and X. Each type is followed by a colon and a value. Each type can be separated by a space, comma, period or not separated at all. Use as many types as needed. Make sure they are enclosed in a single set of quotes.

Mask Types:

A: The full mask used to represent the value of the whole number if equal to zero, enclosed in double quotes. A plus sign, following the second double quote indicates the remaining masks should be used even for a 0.00 value.
Default blank

C: Case of the words.
C = Capitalized
L = All lower
U = All upper.
Default as no conversion.

G: Country group for word format. This represents the way the number is represented in specific locations.
1 = US format (*typically no commas, and no 'and'*)
2 = UK format (*uses commas, and 'and'*)
Default 1

H: Half Adjust Precision
2 = Precision to 2 decimal places (8.575 = 8.58)
Default 2

J: Word group delimiter to separate groups of numbers, such as in 7,200
" " (*default for country group 1-US*) produces - Seven Thousand Two Hundred
"," (*default for country group 2-UK*) produces - Seven Thousand, Two Hundred

L: Specify location of currency word
A = After the precision value (*default for country group 1-US*)
B = Before the precision value (*default for country group 2-UK*)

M: Precision Mask
The full mask used to represent the precision value enclosed in double quotes.
Default "##/100" (where # represents a number in the precision such as, 0.87 = 87/100)

P: Precision = .0 handling
0 = Ignore
1 = Use the mask in type M
2 = Use the word "only".
Default 0

S: Number Word Scale. For values exceeding 999 million, some countries use different words to describe the values. These are known as short scale and long scale. The table shows some of the differences. This mask type affects the use of words past "million".

Value		**Short Scale** *(US, UK, Canadian-English, and most English speaking countries)*	**Long Scale** *(Canadian-French, France and most other non-English speaking countries)*
1,000,000,000	10^9	one billion	one thousand million
1,000,000,000,000	10^{12}	one trillion	one billion
1,000,000,000,000,000	10^{15}	one quadrillion	one thousand billion
1,000,000,000,000,000,000	10^{18}	one quintillion	one trillion
1,000,000,000,000,000,000,000	10^{21}	one sextillion	one thousand trillion

S = Short Scale. Used in the US and UK. (1,000,000,000=one billion)
L = Long Scale. Used in Europe. (1,000,000,000=one thousand million)
Default scale is S

X:nx Filler to close the word string
n = value of 1 - 9 denotes the number of repetitions of the character X
x = filler character (such as * or =)
Default - filler is ignored

Examples

Each example uses an amount of 1234.99 and is followed by the generated result.

`VAL2WRD(amount,' ',' ')`

One Thousand Two Hundred Thirty Four 99/100

`VAL2WRD(amount,'Dollars',' ')`

One Thousand Two Hundred Thirty Four 99/100 Dollars

`VAL2WRD(amount,'Dollars','G:2')`

One Thousand, Two Hundred and Thirty Four Dollars 99/100

`VAL2WRD(amount,'Dollars','X:5*')`

One Thousand Two Hundred Thirty Four 99/100 Dollars *****

`VAL2WRD(amount,'Pounds','G:2 M:”##p” X:5*')`

One Thousand, Two Hundred and Thirty Four Pounds 99p *****

Alphanumeric Functions

Whereas other Sequel functions operate only on character or numeric values, the alphanumeric functions below allow either character or numeric arguments.

Greatest Value	(GREATEST)
Smallest Value	(LEAST)
Convert Null to Value	(VALUE)
Substring	(SUBSTR, SST)
Concatenation	(CAT, BCAT, TCAT)

GREATEST
LEAST

GREATEST and LEAST select the highest (or lowest) values from within a list of fields, constants, or expressions. Elements within the list must have the same data type - character and numeric types cannot be mixed.

GREATEST and LEAST are especially useful in avoiding certain kinds of errors, called *data mapping errors*, that result from a division by zero, or an overflow condition.

The mathematical result of division with a zero divisor is undefined. If your retrieval causes a division by zero, a data mapping error occurs, and the result field will have an undefined value.

Data mapping errors also occur when a field, constant, or calculation result is too large to fit in the column you defined in the view. This can happen quite easily if you specify a result field length that is too small.

Both of these problems can be circumvented by using GREATEST and/or LEAST functions.

Suppose for example, we want to compare each customer's monthly sales amount against their annual sales. We wish to see month to date sales expressed as a percentage of the year to date total. The example below does this:

```
SELECT cusno,cname,mtd$c,ytd$c,
          100 * (mtd$c/ytd$c) Len(3,2) Name(Percent)
      FROM sequelex/custmast
```

If you run the query above, you may notice two different kinds of data mapping errors, each of which are identified by the reason code given on the error message. You can obtain a list of the reason codes and their meanings by positioning your cursor on a mapping error message and pressing the Help key.

One percentage value was omitted because its year to date sales field (YTD$C) has a zero value. The result of the division is mathematically undefined, so the percentage value is unknown. Other values are omitted because the resulting percentage could not fit into a (3,2) field which has a maximum value of 9.99.

The following reformulation solves the problem of dividing by zero.

```
SELECT cusno,cname,mtd$c,ytd$c,
          LEAST(999.99,100*
            mtd$c/GREATEST(ytd$c,1)*ytd$c/GREATEST(ytd$c,1))
            Len(5,2) Name(Percent)
      FROM sequelex/custmast
```

We avoid the overflow problem by increasing the size of the result field to (5,2) and using the LEAST function to return the smaller of the expression result or the largest value that can fit in the field. The LEAST function thus limits the value which will appear in the view.

We can circumvent a division by zero by instead dividing by the expression GREATEST(ytd$c,1). This forces a division by one when the year to date value equals zero. This presents its own problems because the result (MTD$C/1) now equals the month to date sales in those cases where the year to date sales are zero.

To return zero when the year to date sales is value is zero and the correct percentage otherwise, we can make a slight modification. We must add another expression to multiply this result by the result of another division. The second division yields a value of one (YTDC/YTDC) when year to date sales are non-zero, and zero (YTD$C/1) when the year to date sales are zero. The complete expression is as follows:

```
SELECT (mtd$c/GREATEST(ytd$c,0.01))*
          (ytd$c/GREATEST(ytd$c,0.01)) NAME(percent) LEN(5,2)
      FROM sequelex/custmast
```

The query above caused division by zero to return a zero result. We can just as easily force a very high result by using a query like the one below instead.

```
SELECT cusno,cname,mtd$c,ytd$c,
          LEAST(999.99,100*(mtd$c/GREATEST(ytd$c,0.1)))
          Len(5,2) Name(Percent)
      FROM sequelex/custmast
```

Notice that the second division (that turned the result into zero) has been eliminated and the argument in the GREATEST function has been reduced to one-tenth. Dividing the month to date value by this very small value will cause a very large result. Even if the month to date value is as small as 1, the result will be 10. When we multiply this value by 100 to create the percentage, we will create a number larger than the field can hold. Now the LEAST function will limit the result to a value of 999.99!

Translating NULL Values

Sometimes, it is more useful to translate null values to specific results that are not null than to receive a null result from the query. The VALUE function accepts a list of expressions and returns the first non-null result within the list.

`VALUE(expr,expr,expr,...)`

Each expression within the list must return a consistent data type (i.e. all numeric, all character, all date, etc.).

Examples:

```
VALUE(credit, debit, 0)
VALUE(dbdata, "Data Missing")
```

Substring Function

The SUBSTR (or SST) function allows you to break a character numeric field (or the result of a character expression) into smaller pieces. The function requires three parameters that must be separated by commas. The parameters indicate the field, starting position, and length, respectively. Each parameter is allowed to be an expression that creates the appropriate type (character, numeric) of result or numeric field.

`SUBSTR(ADDRESS,15,8)` will extract eight characters from the ADDRESS field beginning with the fifteenth character.

The substring function can be combined with the concatenation function to "edit" a character date field.

```
SELECT CAT(SUBSTR(odcdat,1,2), "/", SUBSTR(odcdat,3,2),
          "/", SUBSTR(odcdat,5,2)) Colhdg("Crt" "Date"),
          odobnm,odobtp,odobsz len(7,0),odobtx
      FROM sequelex/dspobjd
      WHERE SUBSTR(odcdat,5,2)="98"
```

This query shows records in an outfile created by the display object description (DSPOBJD) command. The creation date field (character) is broken apart using substring, and reassembled using concatenation with "/" marks in order to make it more readable. The WHERE clause uses the substring function to select only records with a creation date of 1998.

The substring function is especially powerful when combined with other functions. For instance, separating the first word of a company name field can be done like this:

```
SELECT SUBSTR(cname,1,POSSTR(cname," "))
      FROM sequelex/custmast
```

In this example, the POSSTR function returns the location of the first blank, and this result is used to determine the length of the substring.

Concatenation Functions

Sequel concatenation functions simplify the process of combining the results of character or numeric expressions, especially those containing fixed length fields with trailing blanks. Three functions are provided:

```
CAT(expression,expression,...)
TCAT(expression,expression,...)
BCAT(expression,expression,...)
```

These functions do not provide any extra capability, but they greatly simplify the SQL statement by allowing you to eliminate the trim and blank concatenation steps that would otherwise be required.

The CAT function is equivalent to the concatenation operator (CAT or ||) and combines the results of the alphanumeric expressions without removing leading or trailing blanks. The result of the function is a fixed length field with a length equal to the sum of the lengths of the expressions involved.

The TCAT and BCAT functions trim the trailing blanks from the first expression, leading blanks from the last expression, and both leading and trailing blanks from all intervening expressions prior to concatenating them. If BCAT is specified, a single blank is inserted between all expressions prior to concatenation. The result of the function is a varying length field with a maximum length equal to the sum of the lengths of the expressions involved.

The following examples assume these field values: (~~b~~ represents a blank)

```
FNAME                       ~~bb~~Janebb
LNAME                       ~~bb~~Doe~~bb~~
MINIT                       ~~bb~~S.~~bb~~
CAT(FNAME,LNAME)            "~~bb~~Jane~~bbbb~~Doe~~bb~~"
TCAT(FNAME,LNAME)           "~~bb~~JaneDoe~~bb~~"
BCAT(FNAME,LNAME)           "~~bb~~Jane~~b~~Doe~~bb~~"
CAT(FNAME,MINIT,LNAME)      "~~bb~~Jane~~bbbb~~S.~~bbbb~~Doe~~bb~~"
TCAT(FNAME,MINIT,LNAME)     "~~bb~~JaneS.Doe~~bb~~"
BCAT(FNAME,MINIT,LNAME)     "~~bb~~Jane~~b~~S.~~b~~Doe~~bb~~"
```

Date and Time Functions

Sequel provides several functions that create and manipulate fields that have the date/time/timestamp data type. A comprehensive description of the date and time capabilities of Sequel, begins on page 85.

Creating Date/Time/Timestamp Fields

Fields with date, time, and timestamp data types can be created by using the DATE, TIME, or TIMESTAMP functions. You can also use the CVTDATE function to convert numeric or character fields that contain date values to a date field.

```
DATE(expr)
TIME(expr)
TIMESTAMP(expr[,expr])
```

Expression results must be in a recognizable format with an appropriate separator. That is, they must conform to either SAA, USA, ISO, JIS, or EUR form, or must have the preferred date/time format and separator indicated in the DTSTYLE parameter.

The second operand of the TIMESTAMP function specifies the time value to be placed in the resulting timestamp. If the second operand is not specified, the first operand must specify the entire timestamp.

Examples:

```
DATE("10/30/2009")                                  USA form
DATE("09-12-25")                  (DTSTYLE(*YMD '-') specified)
DATE("11/30/09")                  (DTSTYLE(*MDY '/') specified)
TIME("13.52.23")                                    ISO form
TIME("05:00 PM")                                    USA form
TIMESTAMP("2009-1-31","12:33 AM")
TIMESTAMP("2009-04-30-17.01.10.999999")
TIMESTAMP("20090430170110")
```

Creating a valid date expression from fields containing numeric or character values can be quite tedious. The Convert Date (CVTDATE) function automatically creates a date expression from numeric or character fields. The DATE function is automatically applied to the expression result in order to create a field with a date data type.

```
CVTDATE(expression,type)
```

The expression must contain a valid date in either numeric or character form. The length and format of the expression is given by the type operand. It must be one of the values in the following table.

Type	Date Form	Example
MDY	mmddyy	123109
MDY1	mmddyyyy	12312009
DMY	ddmmyy	311209
DMY1	ddmmyyyy	12312009
YMD	yymmdd	091231
YMD1	yyyymmdd	20091231
CYMD	cyymmdd	1091231
JUL	yyddd	09365
JUL1	yyyyddd	2009365
CJUL	cyyddd	109365

Date types CYMD and CJUL require a *century digit* preceding the year value. The century digit must have a zero value for years between 1900 and 1999. It must have a one value for years between 2000 and 2999.

Date types without a century representation (MDY, DMY, YMD, JUL) are converted in such a way that year values between 40 and 99 will fall between 1940 and 1999, and year values between 00 and 39 will be converted to years 2000 to 2039.

If the expression supplied to the function contains an invalid date value, a mapping error will result.

The CVTDATE function can also be used to convert date values that are represented as separate fields.

```
CVTDATE(yy,mm,dd)
CVTDATE(yyyy,mm,dd)
CVTDATE(cc,yy,mm,dd)
```

Converts three or four fields or expression containing year, month, and day values to a value with a date data type. The expressions must be specified in the order shown above. A century value (18, 19, 20, etc.) may optionally precede the year specification. Values may be supplied in either numeric or character form with the exception that 2 digit year fields can only be numeric when a century field is not supplied. To work around this restriction, simply cast the field as a number:

```
CVTDATE(ZONED(yy,2,0),mm,dd)
```

The combination of values must specify a valid date or a mapping error will result.

Examples:

```
CVTDATE(odcdat,MDY)
CVTDATE(cooyr,coomn,coody)
```

Creating a valid time expression from fields containing numeric or character values can likewise be simplified by using the Convert Time (CVTTIME) function. It automatically creates a date expression from numeric or character fields. The TIME function is automatically applied to the expression result in order to create a field with a time data type.

```
CVTTIME(hh,mm,ss)
CVTTIME(hhmmss)
```

CVTTIME has two formats. The first form accepts 3 character or numeric values. Each must be a valid two digit number representing the hour, minute, and second of the time value to be created. The alternate form is a 6-digit form (character or numeric) containing a time value in hhmmss format.

Retrieving Date/Time/Timestamp Data

Date, time, and timestamp data will be returned in the form indicated by the DTSTYLE parameter. You can use the CHAR function to convert it to a fixed length character string having a specific format.

```
CHAR(expr[,type])
```

The type must be one of the recognized date/time types. If it is not specified, JOB is assumed. If the expression returns a date, the type must be USA, ISO, EUR, JIS, SAA, MDY, YMD, DMY, JUL, JL1, or JOB. If the expression returns a time, the type must be USA, ISO, EUR, JIS, or HMS. If the expression returns a timestamp, the type must be SAA or TS1.

```
CHAR(CURRENT DATE,USA)
CHAR(CURRENT DATE+1 DAY,ISO)
CHAR(timefld,EUR)
CHAR(CURRENT TIME,USA)
CHAR(tstmp,SAA)
```

Portions of a date, time, or timestamp duration or value can be extracted too. The functions below extract the indicated part of the expression and return it as a numeric integer. The expression must have the data type indicated by function (Date, Time, or Timestamp).

```
YEAR(expression)
MONTH(expression)
DAY(expression)
HOUR(expression)
```

```
MINUTE(expression)
SECOND(expression)
MICROSECOND(expression)
```

Each function returns the identified portion of the value or duration created by the expression. The result is always an integer.

A date expression can be converted into an integer that represents its offset from the beginning of the system calendar, or the first day of the year. There are several of these types of functions:

```
DAYS(expression)
DAYOFWEEK(expression)
DAYOFYEAR(expression)
QUARTER(expression)
WEEK(expression)
```

The DAYS function can be especially useful for returning the number of days between two dates or timestamp values.

The expression must be a date or timestamp value, or a string representation of a date. The result is the number of days since January 1, 0001.

The example below finds the number of days between the current date, and the date value contained in the ODCDAT field, which is converted from its MDY format)

```
DAYS(CURRENT DATE)-DAYS(CVTDATE(ODCDAT,MDY))
```

A date expression can be converted into numeric value. There are several of these types of functions:

When the date expression equals December 31, 2009, the following will be returned:

```
CYYDDD(expression)   = 109365
CYYMMDD(expression)  = 1091231
DDMMYY(expression)   = 123109
DDMMYYYY(expression) = 12312009
MMDDYY(expression)   = 123109
MMDDYYYY(expression) = 12312009
YYMMDD(expression)   = 091231
YYYYDDD(expression)  = 2009365
YYYYMMDD(expression) = 20091231
```

Validate Date/Time/Timestamp Data

Three validation functions are available to test whether a date, time, or timestamp value is valid and has a valid type and length based on its format.

```
VALID_DATE(value, “type”)
VALID_TIME(value, type)
```

```
VALID_TSTP(value)
```

VALID_DATE

VALID_DATE accepts an incoming date value and a date type and returns TRUE (1), FALSE (0), INVALID LENGTH(-1), or INVALID TYPE(-2) as an integer result if the date value is a valid date according to the RPG TEST function.

```
VALID_DATE(value, “type”)
```

The valid date types and lengths are:

***MDY, *YMD, *DMY** (6,7,8,9): For use with 2 digit year forms with or without separators. 6 and 7 digit values are assumed to be without separators. 8 and 9 digit values are assumed to have a leading century digit.

***CMDY, *CYMD, *CDM**Y (7,9): With or without separators, having a leading century digit.

***JUL** (5,6,7,8): With or without a separator. 5 and 6-digit values are assumed to be without separators. 7 and 8-digit values are assumed to have LONGJUL (yyyy) format.

***LONGJUL** (7,8): With or without a separator. 4-digit year.

***USA, *ISO, *EUR, *JIS** (8,10): For use with 4-digit year forms with or without separators.

Note: Any other value for date type returns INVALID with result -2. An invalid data input length returns INVALID with result -1. The incoming date value can be a character or numeric (packed) value.

An alternative VALID_DATE format is available with three different forms.

`VALID_DATE(y, m, d)` - Year value may be 4 digits or 2 digits.

`VALID_DATE(c, y, m, d)` - Year value must range from 00-99; century must be a 2 digit value. (19, 20, etc.)

`VALID_DATE(date_value)` - Included for completeness sake. The date_value must be a valid system date type value.

VALID_TIME

VALID_TIME accepts an incoming time value and a time type and returns TRUE(1), FALSE (0), INVALID LENGTH(-1), or INVALID TYPE(-2) as an integer result if the time value is a valid time according to the RPG TEST function.

```
VALID_TIME(value)
```

In this form, the function accepts a single character or numeric time value. *HMS format is assumed.

```
VALID_TIME(value, "type")
```

In this form, the function accepts a single character time value. One of the following type values (formats) are required:

***EUR, *ISO, *JIS, *HMS** (6,8) - With or without separators.

Note: Any other value for time type returns INVALID with result -2. An invalid data input length returns INVALID with result -1. The time value must character.

Two alternative VALID_DATE formats are avialable:

`VALID_TIME(h, m, s)` - Accepts three separate numeric values.

`VALID_TIME(time_value)` - Included for completeness sake. The time_value must be a valid system time type value.

VALID_TSTP

VALID_TSTP accepts an incoming timestamp value and returns TRUE (1), FALSE (0), or INVALID LENGTH(-1) as an integer result if the value is a valid timestamp according to the TEST function. The value may be 19, 26 (with separators), 14, or 20 (with separators) positions.

```
VALID_TSTP(value)
```

An alternative VALID_TSTP format—VALID_TSTP(timestamp_value)—is included for completeness sake. The value must be a valid system timestamp type value.

Grouping Functions

Grouping functions process a single column from a series of records in order to create a result. Seven functions are available: SUM, AVG, MIN, MAX, SDEV, VAR, and COUNT.

The COUNT(*) function requires no arguments and returns the number of records included in the group.

The other grouping functions require one argument. The MIN and MAX functions accept either a character or numeric argument, but all other functions require numeric information. The argument can be a field, constant, or expression.

Grouping functions can operate on either *all* the values in the rows included by the query, or only the *distinct* values returned by the query. For instance, these two function references should return different results:

```
SELECT COUNT(*), COUNT(DISTINCT cstte) FROM custmast
```

The first column will return the number of rows selected by the query. The second column will indicate only the number of distinct state values in the selected rows.

Refer to the section beginning on page 75 for a more complete explanation and several examples of SQL grouping.

Examples:

```
SELECT SUM(amtdu) FROM custmast

SELECT cstte,COUNT(*) FROM custmast GROUP BY cstte

SELECT class,COUNT(*),SUM(csord),
          MIN(csord),MAX(csord),
          SDEV(csord),VAR(csord)
      FROM partmast GROUP BY class
```

The first example produces a single row which contains the total amount due in the customer master file.

The second query creates a group for each state, and reports the state and the number of customers in it.

The third example creates a group for each class of products in the product master and reports customer order (CSORD) activity for the class as a whole. The class, number of products in the class, total order volume, smallest and largest order amount, and the standard deviation and variance among the group are reported for each class in the product master.

Conditional Results - CASE Expressions

Conditional results are created with a CASE expression. A CASE expression can occur anywhere that another expression can be used—in the SELECT, WHERE, or HAVING clauses. A CASE expression can be used to define a new column, to perform a test against another expression result, or even to be used as a further component of another expression.

The CASE expression has two forms. The first, simpler form, lists an expression to be evaluated, then one or more WHEN-THEN clauses, and an optional ELSE clause. It looks like this:

```
...CASE expression
      WHEN expression THEN expression
      WHEN expression THEN expression
           .
           .
      ELSE expression
END...
```

The complete expression is bounded by the words CASE and END. The expression at the top is evaluated and compared against the WHEN expression in each subsequent clause. If they are equal, the corresponding THEN expression is evaluated and returned as the result of the CASE. If no equal WHEN expression is found, the expression defined by the ELSE clause is evaluated and returned. If no ELSE clause has been included in the CASE expression, the result is the null value.

Consider the example below:

```
SELECT prdno, descp, CASE ittyp WHEN "M" THEN "Make"
          WHEN "B" THEN "Buy" ELSE "Type "|| ittyp END
      FROM partmast
```

This simple example shows the product number, description and type from the part master file. The item type field is a coded value. The CASE expression translates an "M" value to a result of "Make", and a "B" value to a result of "Buy". If the item type is neither "M" or "B", the case expression creates a result that concatenates the word "Type" with the item type value. The results could look like this:

Product Number	Product Description	Type
BMXCARTON	BMX Carton	Buy
BMX100	BMX Formula 3 Power Cycle	Type F
102	BMX 3 Rear Wheel (Std.)	Make
103	BMX 3 Seat Back	Make
FLAG	Flag Features	Type F
104	BMX 3 Handle Bar	Make

The more complex form of the CASE expression lets you specify separate conditional expressions and non-equal tests in each WHEN clause. This form is similar to the earlier form and looks like this:

```
...CASE
      WHEN search-condition THEN expression
      WHEN search-condition THEN expression
           .
           .
      ELSE expression
END...
```

The entire search condition follows each WHEN. You can see that the format above is just a simplification of this one. Our earlier example could be rephrased in this format and would still give the same results. The complex expression would look like this:

```
SELECT prdno, descp, CASE WHEN ittyp="M" THEN "Make"
          WHEN ittyp="B" THEN "Buy" ELSE "Type "|| ittyp END
      FROM partmast
```

The more complex form of the CASE expression brings a tremendous level of flexibility to the SQL statement. You have complete freedom over the search conditions, and can use any of the simple tests (no subqueries allowed) that can be used in the WHERE clause (see page 64).

The CASE expression can be used not only to transform "coded" columns into more understandable ones, as we did above, but also to return results of completely different expressions, based on the results of test. The only restriction is that each THEN clause must return the same type of data—either character or numeric.

CASE can also be used to prevent divide by zero and overflow (higher value than fits into the result) errors. For instance, we might want to calculate the percent of credit used by our custom-

ers. We know that some customers have a zero credit limit (which would cause a divide by zero error), and others currently owe more than their credit limit, and would create a "percent used" value over 100%. Using CASE, we might do this:

```
SELECT cname, amtdu, crlim,
          CASE WHEN crlim=0 THEN 0
                WHEN amtdu>crlim  THEN 0.9999
                ELSE amtdu/crlim
          END*100 NAME(pctused) LEN(4,2) EDTWRD("0 . %")
      FROM custmast
      ORDER BY pctused DESC
```

The request returns the customer name, current amount due, and allowed credit limit. It then calculates the "used up" part of the credit and ensures that neither a divide by zero or an overflow will occur. The first WHEN clause tests the credit limit. If it is zero, the CASE result will be zero. The next test determines whether the amount due is greater than the allowed credit. If so, it returns a result slightly less than 1, as an indicator of an overflow condition. Otherwise, the percentage is calculated by division in the ELSE clause.

The case expression thus returns zero, 0.9999, or the correct percentage value. The result is multiplied by 100 and edited with a percentage edit word for easy readability. Finally, rows are ordered in descending order by percent of credit used so that customers above their credit limit appear first on the display or report.

HTML Functions

Sequel provides several functions to facilitate the development of views intended for use with SWI. HREF, HYPERLINK, IMG and IMGREF will build an HTML HREF tag and an HTML IMG SRC tag respectively. These functions simplify the process of building drill down views and views that display pictures using SWI and a browser interface.

HREF

This function builds the HREF tag needed to create a link to another view, report, script, or query object—a ‘drill-down’ link. Each operand can be a field name, a literal, or an expression. The first two operands are required. The first one is the item to appear on the page as an underlined "linkable" item. The second operand names the object to be run. If supplied, the third operand names the library for the object.

Subsequent operands are pairs of expressions. The first item in the pair names a variable used by the view, script, or report. The second item specifies its value.

```
HREF(item,"object"[,"library"[,"variable1",value1
      [,"variable2",value2,...]]])
```

Item - An expression (numeric or character) that will appear as the underlined link.

Object - A character expression result to appear in the OBJ= parameter.

Library - An optional character expression result to appear in the LIB= parameter.

Variable1, value1 - Define the variable name (character expression) and value for each variable to appear. Numeric values are automatically converted to character.

The following example causes the value (1129.67) to appear underlined on the HTML page and the query string to reference object ARQRY.

```
HREF(amtdu,"ARQRY")
```

A HTML tag like the following will be generated

```
<a href="SEQUEL?obj=ARQRY">1129.67</a>
```

This example causes the customer number to appear as the link, and the corresponding cname value to be passed in the query string to the **NAME** variable on the **SEQUELEX/CUSTLIST** object.

```
HREF(cusno,"custlist","SEQUELEX","name",cname)
```

A HTML tag like the following will be generated:

```
<a href="SEQUEL?obj=custlist&lib=SEQUELEX
&%26name=NBCO+Corporation+Inc.">100200</a>
```

HYPERLINK

HYPERLINK is a dual purpose function provided by Sequel.

Used in Viewpoint and SWI:
HYPERLINK creates hyperlinks in your Viewpoint view results to launch Windows or Web-based objects (on the Internet or a network drive) like Excel spreadsheets, PDF documents, Sequel objects, Web sites, or any other type of application. You can run your views locally in Viewpoint or in the Sequel Web Interface using databind=N or databind=A.

Create the underlined link field by using the HYPERLINK function in conjunction with the required hyperlink keyword in the column heading. Key elements are bold:

```
HYPERLINK(URL, CHARDATA) NAME(FLDNAME)
  COLHDG("Column" "Heading" "HYPERLINK")
```

Here is important information about the key elements:

- **URL** is a field or expression that contains the Web or network drive address of the link to the document or object. It must be a character value that provides a valid URL or UNC path such as:
  `"http://www.sequel-software.com"`
  or
  `"file://///POWERi/path/filename.xls"`
- **CHARDATA** is a field or expression that supplies the data that appears underlined in the view results. It must be a character value.

- For Viewpoint only: The **COLHDG** parameter must contain the value "**HYPERLINK**" (or "hyperlink") as one of the column heading entries.
- For SWI, with databind=A, the link will automatically open in a new window or new tab depending on browser settings.

Used in Classic SWI:
The HYPERLINK function can be used with an HREF expression (for creating drill-down links) to cause the link to open in a new window like so:

```
HYPERLINK(HREF(link,"object","library","variable1",value1))
  NAME(FLDNAME) COLHDG("Column" "Heading")
```

Note: Neither HREF nor HYPERLINK(HREF()) give a result that will work in SWI Explorer.

Used in (new) SWI Explorer:
The HREF function (for creating drill-down links) is not supported in the SWI Explorer environment, so the link must be built manually like so:

```
CAT('<a href="http://POWERi/secure/SEQUEL?obj=OBJECT
  &lib=LIBNAME&%26variable=',VALUE,'">',ITEM,'</a>')
```

This is a general example for running a ‘drill to’ view from SWI that runs in the same window as the ‘drill from’ view.

In order to run the ‘drill to’ and open results in a new window, simply add the HYPERLINK function like so:

```
HYPERLINK(CAT('<a href="http://POWERi/secure/SEQUEL
  ?obj=OBJECT&lib=LIBNAME&%26variable=',VALUE,'">',ITEM,
  '</a>'))
```

IMG

This function builds the IMG tag needed to reference an image file through SWI. The first must be a varying length character expression, a literal, or a field name that specifies the location of the image file. This value will be appended to the current URL and processed by the HTTP configuration file to locate the image. The second operand specifies an integer which is the pixel height of the image on the page.

```
IMG(source[,height])
```

Source - A character expression result that will be appended to <img src="SEQUEL/ to form the path to the image object.

Height - An optional numeric expression result that will appear in the height definition of the image.

Example:

To use a jpg image of your logo:

```
IMG("logo.jpg",40)
```

A HTML tag like the following will be generated:

```
<img src="SEQUEL/logo.jpg" height="40">
```

To concatenate the customer number into the image file:

```
IMG(cat("images/C",trim(char(cusno)),".jpg"),100)
```

A HTML tag like the following will be generated causing the customer number to be converted to character, trimmed, and concatenated into the name of the image file:

```
<img src="SEQUEL/images/C100200.jpg" height="100">
```

IMGREF

This function is an enhanced version of the IMG function. It allows for the inclusion of additional attributes for image links such as alt, width, longdesc, and so on.

```
IMGREF(source[,height[,optional tags]])
```

Source - A character expression result that will be appended to **<img src="SEQUEL/** to form the path to the image object.

Height - An optional numeric expression result that will appear in the height definition of the image.

Optional Tags - Typically this is an attribute (alt, width, etc.) set equal to a quoted value.

Example:

To use a jpg image of your logo along with alternate text:

```
IMGREF("logo.jpg",40, 'alt="HS Logo"')
```

A HTML tag like the following will be generated:

```
<img src="SEQUEL/logo.jpg" height="40" alt="HS Logo">
```

The same link with a second optional tag for width:

```
IMGREF("logo.jpg",40, 'width="40" alt="HS Logo"')
```

Will generate this tag:

```
<img src="SEQUEL/logo.jpg" height="40" width="40" alt="HS Logo">
```

URLSTRING Function

This function will build the HREF tag needed to create a link to another view, report, script, or query object. URL format doesn't allow most "special" characters. That means that often, a HREF function in a Sequel statement might create results that can't actually be used. The URLSTRING function can be used to translate a character string that may contain "special" characters into an acceptable URL form.

```
URLSTRING(charexp)
```

Most likely, the URLSTRING function will be used within the HREF function.

```
href("linkname","objectname","libraryname",
      "varname",URLSTRING(charcolumn))
```

If a `charcolumn` value of `"a@b#c$d%e&"` is supplied, Sequel will create a resulting value of:

```
<a href="SEQUEL?obj=objectname&lib=libraryname&%26varname=
      a%40b%23c%24d%25e%26">linkname</a>
```

DB2 Multisystem Functions

These functions are useful primarily in applications of the DB2 Multisystem feature. They return information about how the database has been distributed among more than one System i location.

The HASH function returns the partition number associated with one or more values. The result is an integer with a value between 0 and 1023. If any of the arguments are null, the result is zero. The result cannot be null.

```
HASH(expression,expression,expression...)
```

The NODENAME function returns the relational database name of where a row is located. The argument is an integer that corresponds to a file number in the request. If the argument identifies a non-distributed table, the value of the CURRENT SERVER special register is returned. The result is a variable length character string. The result cannot be null.

```
NODENAME(expression)
```

The NODENUMBER function returns the node associated with a returned row. The argument is an integer that corresponds to a file number in the request. If the argument identifies a non-distributed table, the value 0 is returned. The result is an integer and cannot be null.

```
NODENUMBER(expression)
```

The PARTITION function returns the partition number of a row obtained by applying the hashing function on the partitioning key value of the row. The argument is an integer that corre-

sponds to a file number in the request. If the argument identifies a non-distributed table, the value 0 is returned.

```
PARTITION(expression)
```

SELECT Clause

This clause indicates what columns you want to see. You use this portion of the SQL statement to supply a list of fields, expressions, and literals which should appear in the final view of the data. Each field referenced in this statement must be located in or derived from one or more of the files listed in the FROM clause.

In addition to indicating the information to be included in the view, the SELECT clause also specifies the attributes of the fields. Editing information, length and column heading can be specified. If attribute information is not included for a particular field (column) in the view, SEQUEL will make an appropriate choice based on its source definition or the type of calculation involved.

DISTINCT Phrase

The word DISTINCT can be placed immediately following the word SELECT. This forces the SQL processor to eliminate duplicate rows from the result. It is similar to the UNIQUEKEY parameter but uses the entire row as the criteria for uniqueness rather than some fraction of the ordering list. DISTINCT cannot be used in conjunction with the UNIQUEKEY parameter, nor can DISTINCT be used in a subselect that includes a DISTINCT view logical file in the FROM clause.

```
SELECT DISTINCT cstte FROM custmast
```

will return the list of states represented by customers in the customer master file. Each state will be listed only once.

Select All Fields

There are two ways of specifying the SELECT clause. The first involves simply placing an asterisk after the SELECT keyword. This is a shorthand notation that causes all the fields in each file indicated by the FROM clause to be placed into the view. No calculations will be performed and the field attributes of the columns in the view will be the same as those in the underlying files. For instance:

```
SELECT * FROM custmast ORDER BY cusno
```

will choose all the fields and all the records from the file named CUSTMAST and present the information in order by customer number. The ORDER BY clause is optional.

Fields from the file will be placed in the view along with their current attributes—each will have the same definition as it currently has in the file.

Note: Each field in the view must have a unique name. This can be accomplished by choosing not to select more than one field with a given name, or by renaming duplicate fields with the NAME attribute.

This rule can cause problems when using the "SELECT *" form of the select clause if a given field name is located in more than one file in the FROM list. The "SELECT *" form of the SELECT clause cannot be used in this case and each field must be listed separately.

Field naming conflicts can be resolved simply by renaming one or both fields. Fields can be renamed using the NAME attribute discussed later.

When the SQL statement references more than one file, you can indicate that all fields from a specific file should be included in the view by qualifying an asterisk with the correct table identifier (e.g. *.1, *.cust)

Select Individual Fields

If you want to see only some of the fields from the files you have chosen, or if calculations or literals are required in the view, then each column must be specified explicitly. The "SELECT *" form is not allowed unless the qualified form of the all-fields operator (i.e. *.file or file.*) is used. If more than one column is needed in the view, fields and expressions must be separated with commas. Spaces between items in the list are not required.

For example, we can choose the name and address information for customers in the master file by specifying:

```
SELECT cusno,cname,cadd1,cadd2,cadd3,cstte FROM custmast
```

Obviously, this simple statement will show six columns of information for every record in the file.

```
SELECT prdno.1,descp,quano,actsp,quano*actsp NAME(amount)
      FROM ordline, partmast JOIN prdno.1=prdno.2
      ORDER BY prdno
```

This SQL statement will retrieve information for each order line in the order file (ORDLINE) and calculate an extended amount by multiplying the quantity ordered field (QUANO) by the price (ACTSP). The information will be retrieved in product number order.

```
SELECT CURRENT DATE,*.1 FROM custmast
```

This example returns each row in the customer file. The current date will appear as the first column of the result, customer fields follow it.

Creating Reusable Hidden Results

The Work Data (WDATA) function allows you to create a "hidden" result that will not appear in the query output, but can be referenced by subsequent expressions within the query. Both the NAME() and LEN() attribute can be applied to the result. In order to create a non-output field, the WDATA function must be the outermost function in the expression.

The WDATA function is especially valuable if you need to repeat a calculation several times within a query, as it allows you to specify the calculation only once and thereafter simply refer to the named result of the WDATA function. It can be used to order the data in a sequence that does not otherwise appear on the display. However, if you plan to create a report over the view, sort fields should not be hidden.

```
SELECT WDATA(cooyr*10000+coomn*100+coody) NAME(coymd) LEN(6,0),
          coomn*10000+coody*100*cooyr NAME(comdy) LEN(6,0) EDTCDE(Y),
          cusno,cuspo,ordno,ostat
      FROM sequelex/ordhead
      WHERE coymd>970101 OR coymd BETWEEN 960101 and 960630
          AND coymd<>960401
```

Field Attributes

Field attributes can be used in the SELECT clause to specify formatting information about the data when it is presented to the user. The attribute list follows the field name or expression definition. Five keywords (NAME, LEN, EDTCDE, EDTWRD, and COLHDG) can be used. If not otherwise modified, Sequel uses the existing field definition for simple fields, and creates its own default definitions for derived fields. Keywords in the list are separated with one or more blanks. The order of the keywords is unimportant.

Note: Separate each field specification in the SELECT item list with a comma, but use one or more blanks to separate field names (or expressions) from their corresponding attributes.

Field Naming - NAME Attribute

The name of the field within the view can be changed using the NAME attribute. The name keyword can be specified for any or all fields in the view. Simply type the NAME keyword and enclose the new name for the field in parentheses.

Follow standard naming rules to supply a new name (up to 10 characters) which should be used to reference the field within the view. Once a name has been specified, it can be used elsewhere in the statement. If no column heading has been supplied with the field definition, the field name will be shown on the displayed and printed output.

Field Length - LEN Attribute

The LEN attribute allows you to change the length of a field. The default length is defined in the underlying file, or is calculated by the query processor in order to ensure that significant digits are not lost. Often this length is too long and can be shortened without consequence.

Specify the total length of the field in parentheses following the LEN keyword. Character fields can be up to 32767 characters long.

If the field is numeric, indicate the total length (up to 31 digits), a comma, and the number of decimal positions. The number of decimal positions must be less than or equal to the total length of the field. If you want a number to appear in floating point notation, use the special words SINGLE or DOUBLE to indicate the desired precision.

Note: Specifying a length that is too short causes *data mapping errors*. When using DISPLAY, PRINT, and REPORT functions, the result field will have an undefined value. Other functions such as EXECUTE and insert will cause the record containing the data mapping error to be skipped. If this happens, Sequel will send you a message telling you about it. You can then revise your view to make the field length longer.

Column Heading - COLHDG Attribute

The COLHDG attribute allows you to specify one, two or three column headings which will appear when the view is run. The strings may be up to 20 characters long. They will be displayed and printed on output of the view. Sequel prints up to three lines of headings by filling lines from the bottom up.

A blank column heading can be created by specifying a blank enclosed in quotation marks.

If no COLHDG is specified, the DDS specified column heading for the field will be used. If none exists, the field name will be used instead.

Examples:

```
COLHDG("Heading 1" "Heading 2" "Heading 3")
COLHDG(" ")
COLHDG("ZIP" "Code")
```

Field Editing - EDTCDE and EDTWRD Keywords

Field editing can be specified only for fixed length (not floating point) numeric fields. Unless you specify otherwise, fields will be edited using their DDS specified edit code or word. Fields without edit codes will be edited with the system defined "J" edit code.

Edit codes can be specified using the EDTCDE keyword. Type the character that identifies the edit code or edit description that you want used to edit the values in this field. You can specify any one of the following numbers or letters for edit codes: 1 through 4, A through D, J through Q, and X, Y, or Z. If edit descriptions have been created on your system, you can use numbers 5 through 9 as well.

The various edit codes are used to cause zero suppression and to specify formatting of the trailing negative sign, whether commas appear in the result, and whether zero values print or appear as blanks. Refer to the following charts.

Edit Codes and Their Meaning

Edit Code	Print Commas	Zero or Blanks	Negative Symbol
1	Yes	Zero	None
2	Yes	Blank	None
3	No	Zero	None
4	No	Blank	None
A	Yes	Zero	CR
B	Yes	Blank	CR
C	No	Zero	CR
D	No	Blank	CR
J	Yes	Zero	Trailing -
K	Yes	Blank	Trailing -
L	No	Zero	Trailing -
M	No	Blank	Trailing -
N	Yes	Zero	Leading -
O	Yes	Blank	Leading -
P	No	Zero	Leading -
Q	No	Blank	Leading -

Edit Code	Description
W	Year prefixed date edit. Field (5 to 8 digits) separated by "/"
X	Unedited. Leading zeros not suppressed
Y	Date edit. Field (6 to 7 digits) separated by "/"
Z	Leading zeros are suppressed. No sign shown

A floating currency symbol can be requested by placing the system defined currency symbol (QCURSYM) after the edit code. Specifying EDTCDE(A$) after a numeric field will cause a comma separated value to appear with a leading currency symbol and a trailing "CR" if the value is negative.

Edit words can be specified by enclosing the editing string within quotation marks following the EDTWRD keyword. There must be exactly one blank in the edit word string for each numeric position in the field.

Note: The *Data Description Specifications* manual has a complete description of the format, construction, and use of edit codes and edit words. Refer to it for further reference.

Default value - DFT Attribute

The DFT attribute allows you to control the default value that will appear in the field whenever it is unspecified on an INSERT or a write operation for the file, or when default values are supplied due to a partial outer join. (see page 60) **It is only applied when an output file is created via the EXECUTE command**. Specifying the DFT attribute within a query does not change the

default values supplied by the database when that query is run. The default values that will be supplied by the database during the query cannot be overridden by a DFT value.

The default value must appear in parentheses following the DFT keyword. Character default values must appear in quotation marks (single or double). Numeric values must not be quoted. Date/Time values must be specified in the preferred format of the field. (i.e. default for a *ISO field must appear in "*yyyy-mm-dd*" format)

The usual default value for null capable fields can be changed by indicating a specific value for the DFT attribute. DFT(NULL) specifies that the null value should be supplied instead of a specific value and forces the field to allow null values. If ALWNULL(*NO) is specified on an EXECUTE or OPNSQLF command, none of the fields within the record format will be null capable, regardless of their DFT attribute value.

Coded Character Set ID - CCSID Attribute

The CCSID attribute allows you to control the coded character set ID for character fields. If a CCSID value is not specified, the CCSID for the field will match the CCSID of the underlying field (if this is an expression result), or will be the executing job's CCSID. Refer to IBM's *National Language Support* manual (SC41-5101) for more information about CCSIDs.

FROM Clause

The FROM clause indicates the *location* of the data to be retrieved. At least one file must be specified in the FROM clause. The clause can include file, library, member and format names. Only the file name is required.

Specify the file name first (optionally qualified by the library), then the member name and then the format name. A correlation name can follow each file specification in order to rename the file and provide more understandable field qualification.

Both System i and System/38 file qualification methods are allowed: file.library and library/file name. The member and format names are optional, but the member name must be specified if the format name is given. Separate multiple file specifications in the *from-list* with commas, and place member and format names in parentheses after the file name.

Examples:

```
FROM custmast
FROM custmast.SEQUELEX
FROM sequelex/custmast
FROM custmast(cmp1) cust
FROM custmast,ordhead.*libl h,*libl/ordline(*first viewfmt) l
```

The first example above will reference the first member of the file `CUSTMAST` which will be found on the job's library list. Examples two and three will specifically reference the `CUSTMAST` file in the `SEQUELEX` library. The fourth specification references the `CMP1` member of the `CUSTMAST` on the job's library list, and allows the file to be qualified within the query with the correlation name `CUST`. The final example joins three files, using the library list to locate `CUSTMAST`, `ORDHEAD`, and `ORDLINE`. The `VIEWFMT` record format in the `ORDLINE` file will be used. Correlation names (`H,L`) are specified for both `ORDHEAD` and `ORDLINE`.

Extended file, library, and member names can be used within a Sequel file specification by enclosing the name in double quotation marks. This allows you to specify names containing special characters such as period(.) and lowercase letters. Case (upper or lower) is important when specifying quoted names as the System i will distinguish between "CUST.DTA" and "cust.dta". For example:

```
FROM datalib/"ORD.HDR" hdr
FROM "data.1:3"/"set_a" dseta
FROM "CUST.MAST"(cmp1) cust
```

Note: A correlation name must be specified each time an extended file name is used. You cannot use an extended file name for field qualification. Instead, you must qualify the field by using either the correlation name or file number of the associated file.

Files specified in the FROM clause must be database files. Physical, logical (including join) and distributed data management (DDM) files may be specified. SQL/400 tables, views, and indexes can be used. IDDU defined files may also be referenced. Sequel will access the IDDU data dic-

tionary for the file, record and field definitions of file that are program described and linked to a data dictionary.

Although only one file name is required, the FROM clause can include up to 32 different files to be used in the query.

Special values (*LIBL, *FIRST, *LAST and *ONLY) can be used in place of specific library, member, and format names. *LIBL indicates the library list, *FIRST and *LAST indicate the first (and last) member in the file, *ONLY chooses the only format in the file. The special values will be resolved during each execution. Changes made to the file's location, member list, or format name will not necessarily stop the query from working.

For instance, if *LIBL is specified (or assumed) for the library name, your library list will be searched when the view is run in order to determine which file should be used for the data.

Defaults are taken for elements omitted from the file specification as follows:

*LIBL is chosen for library name,
*FIRST is chosen for the member name, and
*ONLY is chosen for the format name

Logical Files

Specifying a file in the FROM clause means that all the fields and records in that file can be used in the other clauses of the SQL statement. If a logical file is used, perhaps only a subset of the entire physical format will be available, or fields may be mapped to other definitions. All derived fields created in logical formats, such as concatenated, renamed, or translated fields are also available to the query.

In the same way, any select/omit criteria applied to the logical file will also apply to the Sequel view. As with all files, records retrieved through a logical file are not necessarily ordered according to the key path of the file. The sequence can be explicitly specified using the ORDER BY clause.

Note: Each file specification in the FROM clause can indicate only one format. If a multi-format logical file is used, you must indicate which format Sequel is to access. The special format name value *ONLY does not apply and cannot be used with multi-format logical files.

Database Overrides

If database file overrides are present when the view is defined or when it is run, they will affect the files and/or members which are used in the retrieval. Be aware that unusual results can occur if a view is created when an override exists, and later run without the override.

The query processor is not capable of accepting the MBR(*ALL) parameter on the override command. If you wish to execute a retrieval using all data members in a physical file, create a logical file over it and allow the logical file member to be built over all the data members in the physical. Use this logical file in your FROM clause.

JOIN Clause

If more than one file is specified in the FROM clause, the JOIN clause can also be specified. It tells the query facility how to link the files specified in the FROM clause. The JOIN clause provides the same information that is supplied by the JFLD keyword used when creating join logical files with DDS, or when performing a join with the Open Query File (OPNQRYF) command.

SEQUEL does not require the JOIN clause. Linkages between files used in the query can be specified either in the JOIN clause or the WHERE clause. There are some advantages to using the WHERE clause for the join specifications. Specifically, the WHERE clause allows you to:

join files based on expression results for either (or both) parts of the join specification.
omit join specifications to create a Cartesian product

Note: If no joining specification is included for a given file, a full Cartesian product will be created; each record in the primary will be joined to each record in the secondary. If each file has only 1000 records each, a two file Cartesian product will result in 1000*1000 (1,000,000) rows!

The reserved word JOIN may optionally be followed by WITH or BY. Using the WITH or BY preposition is not required, but may enhance the readability of the query. The following three examples are equivalent:

```
SELECT cname,ordno,orval FROM custmast,ordhead
      JOIN cusno.1=cusno.2

SELECT cname,ordno,orval FROM custmast,ordhead
      JOIN WITH cusno.1=cusno.2

SELECT cname,ordno,orval FROM custmast,ordhead
      JOIN BY cusno.1=cusno.2
```

Up to 120 join specifications may be provided in the JOIN clause. Each join specification is of the form: *from-field* join-op *to-field*. The join operator can be any relational (<,>,=,<=,>=,<>) operator. Multiple join specifications are separated with the keyword AND. No commas should be placed in the join specification list.

The *from-field* and *to-field* may be database fields (optionally qualified) or expression results (named in the SELECT clause) that are made from a combination of database fields and constants. The attributes (data type, and length) of the two fields in a join relationship need not be identical.

Several examples of joining have already been given. Here are some more:

```
SELECT cname, ordno, orval
      FROM custmast, ordhead
      JOIN cusno.1=cusno.2

SELECT prdno.1, descp, SUM(quano) LEN(7,0) COLHDG("Tot" "Qty")
```

```
      FROM partmast, ordline
      JOIN prdno.partmast=prdno.ordline
      GROUP BY prdno.1, descp

SELECT cname, ordno.3, line#, descp, quano, actsp, quano*actsp
           NAME(lineamt) COLHDG("Line Total") LEN(7,2) EDTCDE(J$)
      FROM custmast, ordhead, ordline, partmast
      JOIN cusno.custmast=cusno.ordhead
           AND ordno.ordhead=ordno.ordline
           AND prdno.ordline=prdno.partmast
      ORDER BY cname, ordno, line#
```

The last query is an elaboration of an earlier version and joins four files together. The resulting "order inquiry" view combines order information, information from the customer file, and product information.

The join specifications clearly link all four files together, progressing from file to file until all have been linked. The join specification begins by linking the first file (CUSTMAST) to the second (ORDHEAD). Then the order line file is accessed by equating the order number in the order header file with the order number in the order line item file. Finally, the part master is accessed using the product number from the line item file.

The query above can also be expressed as shown below. The results are the same.

```
SELECT cname,ordno.3,line#,descp,quano,actsp,quano*actsp
           NAME(lineamt) COLHDG("Line Total") LEN(7,2) EDTCDE(J$)
      FROM custmast,ordhead,ordline,partmast
      WHERE cusno.custmast=cusno.ordhead
           AND ordno.ordhead=ordno.ordline
           AND prdno.ordline=prdno.partmast
      ORDER BY cname,ordno,line#
```

Types of Joining

Three types of joining can be specified with Sequel requests. The type of joining can be specified prior to the word JOIN within the Sequel statement using the reserved words INNER, PARTIAL OUTER, or ONLY DEFAULT. If the join type is absent, INNER is assumed.

The join type can also be specified with the JTYPE parameter of the Sequel command. When the view is created, all implicitly specified joins will have the type defined by the JTYPE value. During execution, the join type of the first (outermost) subselect can be changed by specifying a new JTYPE value.

Note: The JTYPE keyword must be specified as *INNER if the JOIN clause is not specified. You cannot acquire either partial outer or only default join results if the joining specification is placed in the WHERE clause.

Each JOIN clause within your query can specify a different join type, but only one value is allowed as a JTYPE value for the entire view. The type of joining that occurs within a subquery is not restricted to the join type of the outer query. Each subquery may specify a different type of joining request. All join clauses that are not specifically typed will receive the JTYPE value when the view is created.

Inner Join

The most common type of record linking is established by using the inner join. Only records that can be completely linked using the join specifications will be retrieved. If in the query above for instance, there were no orders for a particular customer, or there were no line item records for a given order, or if no product record matched the PRDNO field in the line item file the join would be unsuccessful and the record would be skipped. An inner join returns only records that match all the joined files.

Specify an inner JOIN using INNER JOIN prior to the join criteria, or by using the JTYPE(*INNER) value when the view is created or run.

Partial Outer Join

A partial outer join request causes the query to return all records in the primary file - even if there are no secondary records matching the joining criteria. This means that if a match is unsuccessful, the query should return a record anyway - substituting default values (usually blanks for alphabetic fields and zeros for numeric) for fields that cannot be found. A partial outer join will return each primary record at least once. If secondary records can be found that match the joining criteria they will be included, otherwise, default values are returned for the secondary fields.

A partial outer join can be requested by specifying PARTIAL OUTER JOIN prior to the join specifications, or by using the JTYPE(*PARTOUT) value when the view is created or run.

Exception Join

The third type of join - known as the exception join can be useful in spotting errors and inconsistencies in the database. Whereas the inner join returns only records that satisfy the linking criteria and the outer join returns all records (substituting default values), the exception join returns only the records that do not match the join specifications. For this reason, the exception join is sometimes called an *only-default* join. Only records from the primary file which do not have a matching secondary record will be returned by the exception join.

As with the outer join, fields that cannot be linked are returned with default values (usually blanks or zeros) in them.

Indicate that you want the query to perform an exception JOIN using the phrase ONLY DEFAULT JOIN prior to listing the join specifications. Alternatively you can specify JTYPE(*ONLYDFT) when the view is created or run.

For more information on the three different types of joining available, refer to the *Control Language Programmer's Guide*, and the *Data Description Specifications* manual.

Example

The two tables below can be used to illustrate the difference between inner, partial outer, and exception joining. In each of the examples that follow, the two tables are joined by customer number using a statement like this:

Customer File

Cust Number	Name	Phone
100112	MNB Corp.	312/640-1258
100200	NBCO Corporation	312/457-1822
100300	Obell Group	315/472-6442
100800	State Corp.	815/514-6252
101200	Maple Leaf	312/248-0050
101616	Sports Shop	312/442-5200
102000	Optimum Corp	312/525-9660
102100	Lim-Equipment Co.	513/299-2960
102311	Lawrence Design	312/654-3221
102900	Taehnrich Corp	312/366-3231

Order File

Cust Number	Order Number	Customer P.O. nbr	Retail Value
100112	110010	S-T5-506	16807.97
100112	110017	S-T5-507	1911.98
100112	110018	S-T5-308	201.00
100112	110019	S-T5-409	301.50
100150	110020	XDF60-A1	1613.20
100200	110022	T6-60-21	37500.00
100200	110024	T6-60-23	12613.20
100200	110025	T6-60-24	14918.69
100200	165022	T6-60-24A	1951.98
100300	165030	V33UT631	100.50
102100	110021	4-V01-20	275.95
102150	110028	verbal	1689.20
102311	110023	W9V5-522A	6930.40

```
SELECT cusno,cname,cphon,ordno,cuspo,orval
      FROM custmast,ordhead
      JOIN cusno.1=cusno.2
```

Inner joining links the tables and selects only the rows that completely meet the joining criteria.

Cust Number	Name	Telephone	Order Number	Customer P.O. nbr	Retail Value
100112	MNB Corp.	312/640-1258	110010	S-T5-506	16807.97
100112	MNB Corp.	312/640-1258	110017	S-T5-507	1911.98
100112	MNB Corp.	312/640-1258	110018	S-T5-308	201.00
100112	MNB Corp.	312/640-1258	110019	S-T5-409	301.50
100200	NBCO Corporation	312/457-1822	110022	T6-60-21	37500.00
100200	NBCO Corporation	312/457-1822	110024	T6-60-23	12613.20
100200	NBCO Corporation	312/457-1822	110025	T6-60-24	14918.69
100200	NBCO Corporation	312/457-1822	165022	T6-60-24A	1951.98
100300	Obell Group	315/472-6442	165030	V33UT631	100.50
102100	Lim-Equipment Co.	513/299-2960	110021	4-V01-20	275.95
102311	Lawrence Design	312/654-3221	110023	W9V5-522A	6930.40

The results of an inner join do not depend on the order of the files. Each row in the result has matching records from each file in the FROM clause. Notice that several customers and order number 110028 are missing because there are no records with a matching customer number in the other file.

Partial outer joining creates at least one result row for each row selected from the primary file in the FROM clause. The table below includes all the rows from the inner join (above) plus a row

for each customer not included in the inner join (those without an order). Default values are supplied for the order fields that cannot be found through joining.

Cust Number	Name	Telephone	Order Number	Customer P.O. nbr	Retail Value
100112	MNB Corp.	312/640-1258	110010	S-T5-506	16807.97
100112	MNB Corp.	312/640-1258	110017	S-T5-507	1911.98
100112	MNB Corp.	312/640-1258	110018	S-T5-308	201.00
100112	MNB Corp.	312/640-1258	110019	S-T5-409	301.50
100200	NBCO Corporation	312/457-1822	110024	T6-60-23	12613.20
100200	NBCO Corporation	312/457-1822	110025	T6-60-24	14918.69
100200	NBCO Corporation	312/457-1822	110022	T6-60-21	37500.00
100200	NBCO Corporation	312/457-1822	165022	T6-60-24A	1951.98
100300	Obell Group	315/472-6442	165030	V33UT631	100.50
100800	State Corp.	815/514-6252			.00
101200	Maple Leaf	312/248-0050			.00
101616	Sports Shop	312/442-5200			.00
102000	Optimum Corp	312/525-9660			.00
102100	Lim-Equipment Co.	513/299-2960	110021	4-V01-20	275.95
102311	Lawrence Design	312/654-3221	110023	W9V5-522A	6930.40
102900	Taehnrich Corp	312/366-3231			.00

Only default joining returns the rows that cannot be completely matched through the joining criteria. Customers not present in the inner join will be listed in the only default join.

Cust Number	Name	Telephone	Order Number	Customer P.O. nbr	Retail Value
100800	State Corp.	815/514-6252			.00
101200	Maple Leaf	312/248-0050			.00
101616	Sports Shop	312/442-5200			.00
102000	Optimum Corp	312/525-9660			.00
102900	Taehnrich Corp	312/366-3231			.00

Unlike inner join, partial outer and only default join results depend on the order of the files in the FROM clause. The table above shows the results from the join when the customer file is listed first. The table below shows the result when the order file is listed first.

Cust Number	Name	Telephone	Order Number	Customer P.O. nbr	Retail Value
100150			110020	XDF60-A1	1613.20
102150			110028	verbal	1689.20

Each row in the order table that doesn't have a matching record in the customer file will be listed. Columns in the customer records now supply default values (zeros and blanks) for the result.

WHERE Clause

The WHERE clause indicates which records from the underlying files are to be chosen during query execution. In essence, it provides an expression which can be evaluated either *true*, *false*, or *unknown* for each record retrieved by the data manager. If it evaluates to a *true* condition, the record is accepted and presented to the user. Otherwise, the record is rejected and another is retrieved from the database.

Fields, expressions and constants can be used within the WHERE clause. A derived field, created in the SELECT clause and given a name with the NAME attribute, can be used in a WHERE comparison by using its assigned name.

Expressions (calculations) are created in the same manner as with the SELECT clause and can be used at any point in the comparison. Refer to the calculation section of the SELECT clause described above for more information on constructing expressions. All non-grouping functions are valid in the WHERE clause. Grouping functions are not valid.

The WHERE clause is composed of one or more comparison tests, also called *search conditions*. Each search condition involves testing fields or expressions against other fields or expressions using comparison operators. As mentioned above, the result of each test is either *true*, *false*, or *unknown*. The results are processed by any boolean operators (AND, OR, NOT, XOR) in the clause in order to determine whether the complete condition is *true* for the record.

Complex Conditions - AND, OR, NOT, XOR

Expressions and conditions within the WHERE clause can grow to become quite complex. Several comparisons can be made within a single statement by using AND, OR, NOT, and XOR (exclusive or - either one or the other but not both) between comparisons. Boolean precedence tests any NOT conditions first, then the AND conditions and finally the OR and XOR. Parentheses can be used to alter the normal evaluation method or to clarify your clause.

The result of an AND condition is *true* if both comparisons are also *true*.

The result of an OR condition is *true* if either one of the comparisons are *true*.

The result of an XOR condition is *true* if exactly one of the comparisons are *true*.

The result of a NOT condition is *true* if the comparison is *false*.

Here is an example of an SQL statement that does several tests:

```
SELECT cname,cphon EDTWRD("   /   -    "),amtdu,crlim
      FROM custmast
      WHERE amtdu > crlim
           OR amtdu > 20000 AND payyr < 89 AND paymn < 12
```

This query shows customers that have an accounts receivable balance greater than their credit limit, or both a current balance greater than $20,000 and a previous payment prior to December

of 1989. Note that the boolean order of operations causes the AND condition to be evaluated before the OR, as if the second and third and fourth comparisons were enclosed in a set of parentheses, like this:

```
SELECT cname,cphon EDTWRD("   /   -    "),amtdu,crlim
      FROM custmast
      WHERE amtdu > crlim
           OR (amtdu > 20000 AND payyr < 89 AND paymn < 12)
```

A negative search condition can be specified by using the NOT operator before defining the condition. For instance, all customers except those in Illinois, Minnesota, or Missouri can be selected by using NOT:

```
SELECT * FROM custmast
      WHERE NOT cstte IN ("IL","MN","MO")
```

Notice that the NOT operator precedes the entire comparison, not simply the comparison operator. An entire set of comparisons can be negated by enclosing them in parentheses and preceding the parenthesized group with a NOT. Creative use of the four boolean operators (AND, OR, XOR, NOT) and appropriate use of parentheses will allow any selection or omission criteria to be specified, no matter how complex.

Sequel Search Conditions

Each search condition within the WHERE clause has the following elements:

a field, expression, or constant to be compared,
a test (comparison operator),
value(s) to compare the field, expression, or constant against

Any field from any file used in the FROM clause can be tested, along with the result of any expression in the SELECT clause. Field qualification may be necessary in order to uniquely identify the field(s) to be tested.

The elements involved in a search condition must have the same (character/numeric/date) data type. You cannot mix numeric and character items within a comparison test. Though they must have the same data type, items in the comparison do not need to be the same length, or have the same number of decimal places if they are numeric.

Numeric tests are performed according to the signed value of the expressions involved. Character tests between fields with unequal lengths are performed so that the smaller field is extended on the right with blanks. Tests involving date, time, and timestamp values are performed chronologically so that later values are evaluated as "higher" than earlier ones.

Character comparisons are case sensitive. Upper and lower case letters do make a difference. Character tests are performed according to the alphabetic collating sequence. This sequence counts all lower case letters less than upper case, and the letter A less than the letter Z. Numbers are highest in the sequence with 9 counted greater than 0.

SEQUEL allows you to use several kinds of comparison tests. They are detailed in the following sections.

Comparison Operators

=	Equal to
<	Less than
<=	Less than or equal to
>	Greater than
>=	Greater than or equal to
<>	Not equal to

Use these operators to test one field, expression, or literal against another. You can use them to perform character or numeric tests, but remember not to mix character and numeric elements within a comparison.

The result of the comparison will evaluate to unknown only if one (or both) of the operands has the null value. Otherwise, the result will always be true or false.

Some examples follow.

```
NAME >= "G"
```

Selects records with a NAME value after "G" in the collating sequence. This includes: Greg, Mike, and Suzette, but would not include: Donna, or Floyd.

```
QUANO*ACTSP=500
```

Only records where the quantity ordered times the actual selling price equals $500.00 are retrieved.

```
OPBAL-ISSUE+RECPT<ORDPT
```

Select records only if the opening balance less issued amount plus received amount is less than the minimum required (order point).

```
COMPAN="ABC Brick"
```

Select all records matching this criteria. Note that a record with a value "ABC Brick Co." would be omitted as would "ABC brick".

Range Checking - BETWEEN, NOT BETWEEN

Numeric and character based range checking is easier with the ..BETWEEN..AND.. construct. This test will evaluate to *true* only if the first expression is equal or greater than the second and equal or less than the third. The two statements below are equivalent.

```
AMTDU BETWEEN 1000 AND 5000
AMTDU>=1000 AND AMTDU<=5000
```

Either numeric or character comparisons are allowed as long as all three operands are of the same type. The result of the comparison will evaluate to *unknown* only if the operand has the null value. Otherwise, the result will always be *true* or *false*.

The following queries choose all customers with a first letter "M".

```
SELECT * FROM custmast WHERE cname>="M" and cname<="N"
SELECT * FROM custmast WHERE cname BETWEEN "M" and "N"
```

Specifying the same value for the beginning and ending criteria in a BETWEEN test is equivalent to simply specifying an equals relationship.

```
SELECT * FROM custmast
      WHERE cname BETWEEN "Sports Shop" AND "Sports Shop"
```

The statement above is awkward and is more properly phrased as shown below, but returns the same result.

```
SELECT * FROM custmast WHERE cname="Sports Shop"
```

Date Comparison

BETWEEN is also useful when choosing a set of transactions within a specific date range. The INVDAT field in the invoice master file is a numeric field in YYYYMMDD format. We can choose records from August 1 through August 31, 2009 with:

```
SELECT * FROM invmast
      WHERE invdat BETWEEN 20090801 AND 20090831
```

Character Search - CONTAINS

This search operator is useful for searching character fields (and expressions) to determine if they include a specific sequence of characters. You cannot use the CONTAINS operator with numeric fields or expressions.

While either a character field or an expression can be used on the left side of the CONTAINS operator, only a character constant, enclosed in quotation marks, can be specified on the right side.

CONTAINS searches the first expression for an occurrence of the second. For instance, the contents of a name field may be searched to determine if it contains "SMITH". As with all character comparisons, upper and lower case letters are treated as different values.

The result of the comparison will evaluate to unknown only if the expression operand has the null value. Otherwise, the result will always be true or false.

Examples:

```
SELECT * FROM custmast WHERE cname CONTAINS "Sport"
SELECT * FROM partmast WHERE descp CONTAINS "Nut"
```

Pattern Search - LIKE, NOT LIKE

The LIKE operator is similar to CONTAINS, but more powerful. It too is a character operator and allows only a character field or expression on the left and a quoted string on the right.

Unlike CONTAINS, the right side of a LIKE predicate specifies a pattern string which is used to determine a match with the left hand side. Generic and wild card characters can be used in creating the pattern. An asterisk (*) indicates that any number of characters (zero or more) can appear in the searched string; a question mark (?) allows any single character to appear.

For instance, the mask `*Anders?n*` will locate strings containing Anderson or Andersen. The first asterisk allows any number of characters to precede the "A". Likewise, the trailing asterisk is necessary since without it the mask would demand that the desired string be right justified within the field.

The result of the comparison will evaluate to *unknown* only if the expression operand has the null value. Otherwise, the result will always be *true* or *false*.

Notice that CONTAINS is a subset of the LIKE comparison. Any test made with CONTAINS can be translated into a LIKE comparison, but the reverse is not true. For instance:

```
SELECT * FROM partmast WHERE descp LIKE "*Nut*"
```

is equivalent to the last example in the section above.

LIKE can be used to create comparisons that cannot be done any other way. For example:

```
WHERE lstnam LIKE "*Jo*ns?n*"
```

will find any record with a last name value of Johnson, Johnsen, Johansen, Jonson, etc.

The LIKE operator can be prefaced with the word NOT in order to return only the records with a false result of the LIKE test. For example,

```
WHERE lstnam NOT LIKE "*Jo*ns?n*"
WHERE NOT lstnam LIKE "*Jo*ns?n*"
```

are equivalent and will both return records where the lstnam field is not like the pattern mask. Since the NOT operator will only return rows that have a false result, rows with a null value for the `LSTNAM` field will not be returned.

Set Comparisons - IN, NOT IN

The IN operator allows you to find out if a field matches one of a list of values. The first expression is checked against a list of values (up to 50) supplied in parentheses. If a match is found, the predicate value is *true*, and the record is selected. Otherwise the record is rejected. The result of the comparison will evaluate to *unknown* only if the expression operand has the null value.

The IN comparison is similar to the VALUES concept in DDS, and is equivalent to expressing a series of tests separated by OR.

The following SQL statements select only customers in Illinois, Minnesota, or Missouri and presents them in state and zip code order.

```
SELECT * FROM custmast
      WHERE cstte="IL" OR cstte="MN" OR cstte="MO"
      ORDER BY cstte,czipc

SELECT * FROM custmast
      WHERE cstte IN ("IL","MN","MO")
      ORDER BY cstte,czipc
```

Either character or numeric comparisons can be performed with the IN operation. Naturally, all items within the list must have the same data type and it must match the data type of the expression to the left of the IN operator.

```
SELECT * FROM ordhead
      WHERE coody IN (6,7,8)
```

This statement will select records from the order header file in which the order day falls on the 6th, 7th, or 8th of the month. As you gain experience with SQL you will learn that there is usually more than one way to phrase a query. Some are simpler and easier to understand than others.

For instance, assume that a date field (6,0) is stored in MMDDYY format. Selecting records that have a month value of June, July, or August can be done in either of two ways. Obviously the second method is more easily understood than the first.

```
WHERE (date-date MOD 10000)/10000 in (6,7,8)
WHERE date BETWEEN 060100 and 083100
```

The IN operator can be prefaced with the word NOT in order to return only the records with a false result of the test. For example,

```
WHERE cstte NOT IN ("IL","MN","MO")
WHERE NOT cstte IN ("IL","MN","MO")
```

are equivalent and will return only those records with a false result of the test. Because a null value in the cstte field causes an unknown (neither true nor false) result, rows with null cstte values are rejected.

Null Comparisons - IS NULL, IS NOT NULL

The NULL operator allows you to find out if a field or expression contains the null value. The operand can be any data type. The result of the comparison is either *true* or *false*. The statement

```
SELECT * FROM custmast WHERE cstte IS NULL
```

will retrieve only the rows in which the cstte field has a null value.

IS NOT NULL can be used to test whether a field or expression is not null.

```
WHERE cstte IS NOT NULL
```
and
```
WHERE NOT cstte IS NULL
```

are equivalent and return only the rows where the cstte field does not have the null value.

Note: If your database contains null values, IS NULL and IS NOT NULL comparisons should be used in all of your queries. Handling the null values explicitly will clarify the intent of your query and decrease the possibility of unexpected results.

Subquery Comparisons - Basic Comparison

A select statement can be included in a comparison within the WHERE or HAVING clause of another select statement. This second or inner statement is known as a *subquery*. Using subqueries is an effective way to compare values within one file against values of another.

In most cases, subquery statements can be avoided by expressing the query as a join or by using multiple steps that involve the creation of intermediate results. Occasionally however, you may find that the power provided by the subquery operators is exactly what is needed to solve a query problem.

SEQUEL can perform several different types of subqueries. The simplest form involves comparing a field or expression result to the result of a select statement. For instance,

```
SELECT * FROM custmast c
      WHERE oropn <>
           (SELECT SUM(orval) FROM ordhead o
                 WHERE c.cusno=o.cusno)
```

The not equal operator (<>) is used to compare the OROPN field (amount on order) in each customer record to the result of the subquery. The subquery is a grouping query that totals the order value for all of the customer's orders. Notice that the query above uses correlation names to qualify the CUSNO field within the subquery so that only orders for the "outer" customer are included in the total.

Basic subqueries are constructed simply. A field or expression in the where clause is compared by means of a comparison operator (=,<,>,etc.) to the result of a select statement which must be surrounded by parentheses.

Subqueries involving a basic comparison must satisfy two rules:

the subquery SELECT clause must specify exactly one field, and
the subquery must return only one record (or none)

If either of these rules is violated, the query will end with an error. If the subquery returns no records, the null value is returned and the result of the comparison is *unknown* (neither *true* nor *false*).

Subquery Comparisons - Quantified Comparison

A simple variation of the basic subquery comparison allows several records to be included by the subquery. By using the reserved word SOME, ANY, or ALL prior to the subquery, a set of values can be compared against a field or expression result, much like the IN test described on page 1-69. For instance,

```
SELECT * FROM ordline
      WHERE prdno = ANY (SELECT prdno FROM partmast
                              WHERE ittyp="B")
```

Conceptually, the subquery creates an intermediate result listing part numbers in the part master that have a "B" item type value. Rows from the order line file are selected if the product number given in the order matches any item in the list.

Subqueries involving a quantified comparison must, like their basic comparison counterparts, identify only a single column in the SELECT clause. As indicated above however, the subquery is not constrained to returning only one row.

When SOME or ANY is specified (they are equivalent) the comparison will be *true* if the subquery returns at least one values that yields a *true* result for the test. The result is *false* if the comparison returns a *false* result for every row in the subquery or if no rows are returned at all. Otherwise, the result is *unknown*.

When ALL is specified the comparison will be *true* if every row returned by the subquery yields a *true* result for the test or if no rows are returned at all. The result is *false* if it is *false* for any row returned by the subquery. Otherwise, the result is *unknown*.

Using these rules, it is easy to see that the following query will return part records only if the list price for the part is greater than the price paid on any open order for it or if the part is not found in the order file.

```
SELECT * FROM partmast p
      WHERE lstpc > ALL (SELECT actsp FROM ordline o
                              WHERE p.prdno=o.prdno)
```

The example above illustrates the potential confusion between the English meaning of the word "any" and the SQL logic of ANY and ALL. In general, you should try to avoid formulating queries that use SOME, ANY, or ALL by creating subqueries that use IN or EXISTS comparisons instead.

Subquery Comparisons - IN Comparison

The subquery IN comparison provides a simple and familiar way of testing a field or expression result against a list of values. Like the basic and quantified subqueries already described, the IN subquery must identify only a single column in the SELECT clause. This column value from every row chosen by the subquery is compared for an equal match with the field or expression preceding the IN operator. For instance,

```
SELECT cusno,cname FROM custmast
      WHERE cusno IN (SELECT cusno FROM ordhead)
```

The subquery returns a list of customer numbers in the order file. The "outer" query chooses only the customer rows with a customer number in the list. Customers without a record in the order file will not be returned by the query.

The IN subquery must return only one column. Any number of rows can be selected. Like the IN comparison described on page 1-69, the data type (numeric, character, date) of the subquery column must be compatible with the field or expression used in the comparison.

Note: The `IN (subquery)` test is equivalent to an `= ANY (subquery)` comparison. Most people also find it easier to understand.

The result of the IN test will be *true* if there is at least one row in the subquery with a value matching the field or expression given in the comparison. The result is *false* if the comparison is *false* for every row returned by the subquery, or if the subquery returns no rows at all. Otherwise the result is *unknown*.

Subquery Comparisons - EXISTS Test

The EXISTS test is the most powerful subquery operator. Unfortunately, it can also lead to the most complicated query statements. It is used simply to determine if the subquery that follows it returns any rows at all. The specific values returned by the subquery are inconsequential. They are never used in the query.

The result of the test is *true* if one or more rows is returned by the subquery. If the subquery returns no rows, the result is *false*. The result of the EXISTS test can never be *unknown*.

The example below uses an EXISTS test to return only the customers that have one or more records in the order file. It is equivalent to the example of the IN subquery shown in the previous section.

```
SELECT cusno,cname FROM custmast cust
      WHERE EXISTS (SELECT * FROM ordhead ord
                        WHERE cust.cusno=ord.cusno)
```

The Value of Subqueries

As you have already learned, there are often several ways to phrase a particular query request. In some cases, a subquery will be the most natural form, in others it will not. Most subqueries can be rephrased into simpler queries that involve joining. The result is usually easier to understand than the original subquery form.

Sometimes, use of a subquery can provide results that would be more difficult (or even impossible) to achieve with a single level request. Some of these are listed below so that you can gain additional experience and insight into the use of subquery expressions.

Comparison Against a Single Grouped Value

A subquery can be used to return a single grouped result that can be compared against data values in a primary record. For instance, finding the customers with an amount due (AMTDU) above the average for the entire customer file is as easy as:

```
SELECT * FROM custmast
      WHERE amtdu > (SELECT AVG(amtdu) FROM custmast)
```

The subquery creates a grouped result using the AVG function. It can be compared to each record in the customer file. Other grouping functions (MIN, MAX, SUM) can be used for similar purposes.

Comparison Against Smallest, Largest Within a Group

If you want to access the information in a row having the smallest or largest field value within a group of records, there is probably no better way than using a subquery. Assume for the moment that we want to retrieve order information for the largest order placed by each of our customers. The query can be phrased like this:

```
SELECT * FROM ordhead head1
      WHERE EXISTS (SELECT COUNT(*) FROM ordhead head2
           WHERE head1.cusno=head2.cusno
           HAVING head1.orval=MAX(head2.orval))
ORDER BY ordno
```

Though it appears intimidating at first, the query is not difficult to understand. Read it as follows:

> Select a record from the order header file if, by examining the order file and choosing only orders for this customer, the value for the order we have selected matches the largest order value in the group of this customer's orders. Show the results in order by our order number.

Finding Occurrences of a Set of Records Within a File

With a complex subquery involving two levels of nesting, you can find out if one file has every record within a set of records in another file. For instance, suppose you wanted to know all the

customers (if any) with all of our finished goods on order. A query like the one below will provide the answer:

```
SELECT * FROM custmast c1 WHERE NOT EXISTS
      (SELECT * FROM partmast p1
      WHERE ittyp="F" AND NOT EXISTS
           (SELECT line# FROM ordline l1,ordhead h1,
           partmast p2,custmast c2
           WHERE c1.cusno=c2.cusno AND p1.prdno=p2.prdno
                 AND l1.ordno=h1.ordno AND l1.prdno=p2.prdno
                 AND h1.cusno=c2.cusno))
```

The query works by finding all customers (outer query) for which we can find no finished good part (second query) that is not on order (innermost query) for the customer. Using rules of logic (double negative creates a positive), only customers having all finished goods parts on order will be selected.

Mixed Join Types

Using subqueries, you can create a query that involves an outermost join that is different from a second (or third) join type. These queries can probably be rephrased as single level queries involving a single join type (but they might not be as easily understood).

```
SELECT cusno,cname,ordno FROM custmast,ordhead
      INNER JOIN cusno.1=cusno.2
      WHERE ordno IN (SELECT ordno FROM ordline,partmast
                      ONLY DEFAULT JOIN prdno.1=prdno.2)
```

The query above lists customers (and their orders) if any line of the order cannot be matched against an existing part in the part master file.

GROUP BY Clause

One of the most powerful features of SQL is its ability to group records together. A grouping query assembles records into one or more groups, processes each group as a whole, and presents one result record per group to the user. It is a very effective way to summarize the information in your database.

A grouping query is created whenever one of the seven column functions (COUNT, AVG, SUM, MIN, MAX, SDEV, VAR) are used.

A grouping query may or may not include a GROUP BY clause. If no GROUP BY clause is included, the entire file is treated as a single group.

Use of the GROUP BY clause causes the records in the file to be split into sets. The clause consists of a list of comma separated field names. Within each set, records have common values for each of the fields listed in the GROUP BY clause. Each set of records will be reduced to a single "group" record according to the column functions present in the SELECT clause.

After the query processor assembles the records into their respective groups, column functions can be used to perform derivations on the whole group of records. They can be counted. A particular field can be totaled or averaged. The smallest or largest values for a given field can be determined. Statistical variance or standard deviation across the group can be calculated.

In addition, you can also specify whether only distinct values within the set should participate in the function, or whether all values in the set should be included. Thus you can count the number of distinct entities, and compute the sum, average, or statistical variation among the unique values in the column, instead of the entire list of values.

The example below illustrates what happens in a grouping operation.

Ungrouped (detail) data

Customer Number	Order Number	Customer PO nbr	Retail Value
101100	110011	V-T35810	16807.97
101200	110012	73V29-06	16807.97
101616	110016	69T35-05	1911.98
101616	165016	69T35-06	16807.97
102000	110020	TZ432-06	100.50
102100	110021	4-V01-20	275.95
102100	165021	4-V01-23	16807.97
102311	110023	W9V5-522A	6930.40
102311	371323	W9V5-522B	37500.00
102600	371326	Y-22-Y5	6930.40
102800	165028	UV9-061	301.50
102800	247528	UV9-062	16807.97
102800	371286	UV9-063	16807.97
102900	371290	VY537-18	16807.97
102900	371292	VY537-20	16807.97
102900	371299	VY537-27	1951.98
102900	556929	VY537-15	16807.97

Grouped (summary) data

Customer Number	Order Count	Highest Order	Lowest Order	Total Value
101100	1	110011	110011	16807.97
101200	1	110012	110012	16807.97
101616	2	165016	110016	18719.95
102000	1	110020	110020	100.50
102100	2	165021	110021	17083.92
102311	2	371323	110023	44430.40
102600	1	371326	371326	6930.40
102800	3	371286	165028	33917.44
102900	4	556929	371290	52375.89

During the grouping operation, records were separated into sets by customer number. Each set produced one grouped result row. The row provides the customer number associated with each set of records, the number of records in the set, the highest and lowest order number in the set, and the total value of all rows (orders) that are included.

Grouping does not allow you to "mix" detail and summary information. Each grouped record represents information about the entire set. There is no way to include information from a single record within the set (such as the order number itself) because there are many separate values for the column but room for only a single value in the result row.

Note: When grouping is used, each field or expression within the SELECT clause must be:

a) named in the GROUP BY clause, or
b) an aggregate expression

This is the most important rule about grouping queries. An *aggregate expression* is created whenever a grouping function is used, or when a calculation involves only grouping fields. An aggregate expression may involve constants or other aggregate results. If any derived fields are created in the select clause, grouping functions must be used to create them.

```
SELECT cstte,SUM(amtdu) FROM custmast GROUP BY cstte
```

This statement totals the amount due by state, and returns the total amount and the state name. It would be invalid to place the field CNAME into the request since there are many values for CNAME in each group.

If no GROUP BY clause is included in the query, the entire file is treated as a single group. The following query tells several interesting facts about the customer file by returning a single, grouped record:

```
SELECT COUNT(*) COLHDG("Number" "of" "customers"),
           SUM(amtdu) COLHDG("Total" "due") LEN(11,2),
           MIN(amtdu) COLHDG("Smallest" "amount") LEN(9,2),
           MAX(amtdu) COLHDG("Largest") LEN(9,2)
      FROM custmast
```

Confusion can sometimes arise due to the dual nature of the AVG, SUM, MIN, and MAX functions. If more than one operand (field) is supplied to the function, (e.g. AVG(MON, TUE, WED, THU, FRI)), then the values of the listed fields will be averaged, totaled, etc., for each *non-grouped* record in the view. If only one field is supplied (e.g. AVG(MON)), then the function will apply as a *grouping* function to a set of records and return the average, total, etc., over a field value for all the records in the group.

```
SELECT class,SUM(recpt)+SUM(adjst)-SUM(issue) NAME(flux) LEN(7,0)
      FROM partmast GROUP BY class ORDER BY flux DESC

SELECT class,SUM(recpt+adjst-issue) NAME(flux) LEN(7,0)
      FROM partmast GROUP BY class ORDER BY flux DESC
```

```
SELECT class,SUM(SUM(recpt,adjst)-issue) NAME(flux) LEN(7,0)
      FROM partmast GROUP BY class ORDER BY flux DESC
```

The three queries are equivalent. The first example totals the three columns before combining them. The second query calculates the flux (income-outgo) for each record in the group, then accumulates the result. The third works like the second, but uses the intra-record to add RECPT to ADJST. The answer is the same in each case, but the third example is more difficult to understand because it uses the intra-record SUM function. Generally, it is best to use SUM only for grouping operations and use the addition operator (+) for intra-record accumulation.

Grouping Performance

Though it is a powerful feature, use of the GROUP BY clause can sometimes result in lengthy wait times and complaints of "poor performance" of the query. This happens simply because it is necessary to access so many records prior to showing the user their requested data. Because the display program shows 16 or 18 view records at once, it is conceivable that a grouping query may access several thousand records from the database in the process of filling a display page. Indeed, if the select clause includes grouping functions and no GROUP BY clause is used, all records in the file will be read before any data is returned to you; you will receive a single data record containing the results of the query over the entire file!

An additional consideration occurs with regard to concurrent file access during the grouping operation. In order to ensure that grouped results are accurate, the query processor locks the file from update operations during the processing of each group. When the group is finished, the file is unlocked, allowing update requests to complete normally. As a result, concurrent users that require update access to the files referenced by the query may experience temporary performance degradation due to locks against the file. The wait time these users experience will increase in proportion to the number of records included in each grouped row. This predicament can be avoided by taking either of two actions:

Schedule queries that produce a small number of groups from very large files to be run when they are not likely to interfere with interactive users requiring update access to the files.

Use a two step approach that uses EXECUTE to create a temporary file containing the records you want to summarize. You should include a WHERE clause so that the temporary result has only the records you want grouped. Perform the grouping query on the copy, rather than the data which is in use.

HAVING Clause

Once the files have been partitioned into groups, a boolean condition can be applied to them in order to further filter your view. This is accomplished through the use of the HAVING clause. The HAVING clause is similar to the WHERE clause, but it applies to the grouped record rather than the underlying "un-grouped" records.

Note: A HAVING clause is never required, but it cannot appear unless the view requests grouping by specifying a GROUP BY clause.

The tests allowed in the HAVING clause are the same as those allowed in the WHERE clause. Unlike the WHERE clause however, the HAVING clause can include aggregate field references and aggregate expressions. These items can be compared with each other or with constants using the standard comparative functions and operators.

While the HAVING clause could conceivably be as complex as a WHERE specification, some simple HAVING examples produce powerful queries:

```
SELECT ordno,SUM(quano*actsp) NAME(ordval) LEN(7,2)
      FROM ordline
      GROUP BY ordno
      HAVING ordval > 10000
      ORDER BY ordval DESC
```

The example above calculates the order value by gathering together all lines for each order, multiplying the quantity ordered on each line by the selling price, and totaling the result. Only orders exceeding $10,000 will be included in the view.

```
SELECT salno,avg(amtdu) len(7,2),count(*) name(nbr)
      FROM custmast GROUP BY salno
      HAVING nbr > 5 AND avg(amtdu) > 5000
```

This example accumulates the customer amount due by salesman, and shows the salesmen with more than five customers and an average amount due more than $5,000 for those customers.

Grouping Performance

Query performance can be improved by using the WHERE clause to filter out records which will not appear in the final view rather than forcing the HAVING clause to do it. Using the HAVING clause to perform checks that could be done with the WHERE clause causes the computer to do extra work—first grouping a series of records together, performing some calculations and then discarding the result.

Note: The HAVING clause should only include tests on aggregate expressions. Do not use it when the comparison can be performed by the WHERE clause.

The following SQL statements both return the exact same result, but the *second will be faster since the grouping function will not be performed for all the states, only for the three selected*.

```
SELECT cstte,COUNT(*) FROM custmast
      GROUP BY cstte HAVING cstte IN ("IL","MN","MO")

SELECT cstte,COUNT(*) FROM custmast
      WHERE cstte IN ("IL","MN","MO") GROUP BY cstte
```

UNION and UNION ALL Clauses

UNION and UNION ALL create a result by combining rows that are defined by SELECT statements on each side of the UNION or UNION ALL phrase. If UNION is specified without the ALL keyword, the result has all duplicate rows eliminated.

The effect produced by the UNION phrase is similar to the result of a logical file specifying more than one member in the DTAMBRS keyword of the Create Logical File (CRTLF) command. Records from the underlying query results are merged into a single result view.

The select statements on either side of the UNION specification must be *union-compatible*. That is, corresponding columns from each subselect must have corresponding character or numeric type. Length and attributes of the columns need not match exactly. Thus, a Packed(9,2) column is union-compatible with a Binary(4) column (since they are both numeric values) but not with a Character(15) column.

Attributes of the result columns conform to the largest attribute definitions from the corresponding columns supplied by the query definition. The Sequel column names, headings, and edit codes are acquired from definitions placed on the first SELECT clause in the query statement. The query:

```
      SELECT cusno,cname,cstte FROM custmast
UNION ALL
      SELECT vendno,vname,vstte FROM vendmast
ORDER BY vstte
```

will create a list of all customers and vendors and place them in state order. Duplicates are not removed.

The ORDER BY clause requires the field names from the last SELECT clause. The view above is sorted by `VSTTE`. When creating a physical file from a UNION view and fields are specified on the ORDER BY clause, a different requirement exists. Fields used on the ORDER BY clause must have the same names on all SELECT clauses of the UNION view. Simply assign an alias name to the fields to be used on the ORDER BY. The view above would have to be changed:

```
      SELECT cusno,cname,cstte name(state) FROM custmast
UNION ALL
      SELECT vendno,vname,vstte name(state) FROM vendmast
ORDER BY state
```

The query below will produce a result showing all customers from three members of the customer master, along with a literal identifying their source member. Customers will appear in customer number order.

```
      SELECT cusno,cname,amtdu,"Chicago"
          FROM custmast(chicago)
UNION ALL
      SELECT cusno,cname,amtdu,"New York"
          FROM custmast(newyork)
UNION ALL
      SELECT cusno,cname,amtdu,"Los Angeles"
          FROM custmast(losang)
ORDER BY cusno
```

ORDER BY Clause

The final clause in the query allows you to specify the order of the records when they are presented.

Note: Unless an ORDER BY clause is used by the query, records in the view will be returned in system defined sequence—an order chosen by the system which yields maximum performance for the query. System defined order may change from one execution of the query to the next.

Several fields can be supplied on the ORDER BY clause, as long as the total length represented by the fields does not exceed 256 positions (bytes). The specification for each field can indicate whether it is to be sequenced in ascending (ASC) or descending (DESC) order. If the ordering is not specified, ascending order is assumed. Numeric values can be treated as unsigned numbers by using the ABS ordering keyword after the field name. Multiple ordering specifications must be separated with a comma. Each field in the ordering specification must be identified by its name or its index in the SELECT clause.

Examples:

```
ORDER BY cname
ORDER BY amtdu DESC
ORDER BY cusno,ordno,line#
ORDER BY flux ABS DESC,descp
ORDER BY 5 DESC ABS,descp
```

Note: The query can be ordered according to field and/or expression values, but only field specifications may appear in the ordering clause. If the query statement includes UNION phrases, the ORDER BY must identify field names found in the last SELECT clause.

This means that if an ordering based upon an expression is desired, the expression must be defined in the SELECT clause and given a name using the NAME attribute. The derived field can then be used for ordering by indicating its name in the ORDER BY clause.

```
SELECT cname,cphon,cstte,amtdu
      FROM custmast
      WHERE amtdu > crlim
      ORDER BY cstte,amtdu DESC
```

The query shows customers over their credit limit in state order with customers having the largest balances first. The states will be listed in ascending alphabetic order, the customers within each state will be listed in descending order according to amount due.

Unique Key

The ORDER BY clause can work in conjunction with the UNIQUEKEY parameter on Sequel commands when SERVER(*SEQUEL) is specified. The UNIQUEKEY parameter tells the query processor to return only the first record in the series if more than one record has the same

ordering criteria. Either some or all of the fields specified in the ORDER BY clause can be used to establish the comparison criteria. For instance,

```
SELECT * FROM custmast ORDER BY cstte,czipc
```

will normally choose all records in the customer master and return them alphabetically by state, and in increasing zip code order.

However if UNIQUEKEY(1) is used when the view is created or executed, only one record for each state present will be returned. That record will be the one with the lowest zip code since the ORDER BY places the records in that order.

If, on the other hand, UNIQUEKEY(*ALL) or UNIQUEKEY(2) is specified (they are equivalent in this case) when the query is created or executed, both fields in the ORDER BY will be used to determine uniqueness. The result is that one record will be retrieved for each state-zip code combination.

UNIQUEKEY is useful when you want to choose the first or last record in a series. The order file may have several orders for a given customer, but suppose we are interested in finding only the most recent one? By typing the command:

```
DISPLAY SQL('SELECT cusno,cname,ordno,
          coocc*1000000+cooyr*10000+coomn*100+coody
          Len(8,0) Name(date)
      FROM custmast,ordhead JOIN cusno.1=cusno.2
      ORDER BY cusno,date desc')  UNIQUEKEY(1)
```

we place the records in descending date order and pick only the first one (the most recent) for each customer. You can access the oldest order for each customer simply by placing the records in ascending date order, removing DESC from the end of the ORDER BY clause.

The UNIQUEKEY parameter value, like several other command parameters can be specified in the Design View interface through the File\Properties screen.

Ordering Performance

Use of the ORDER BY clause will cause the query processor to use an *access path* to present the data to the user. If an existing access path can be used, the query processor will make use of it. Otherwise, a new one must be created. If a calculated field is used in the ordering, it may be necessary to create a temporary result first, then create the access path over it. Sometimes, the benefit achieved by ordering the data is outweighed by the time it takes to create an access path. In these cases it may make sense to dispense with the ORDER BY clause entirely, accepting whatever order the system chooses to present the data instead.

Working With Date and Time Values

One of the most useful aspects of Structured Query Language (SQL) is its ability to work with date and time information. Sequel incorporates all the standard SQL date and time capabilities. As a result, it is easy for you to create Sequel requests that:

compare date values within your database without regard to their format (MDY, YMD, etc.) or structure (number and type of fields)

use addition or subtraction to change a date by some number of days, months, and/or years - automatically adjusting for end of month and year

find the difference between two date values

order database records according to a date value without regard to its format (MDY, YMD, etc.) or structure (number and type of fields)

present date and time values in a format that matches your personal or cultural preference. (i.e. YMD, MDY, etc.)

Using these capabilities requires that you understand a few principles of date and time information and how Sequel works with it. This section should provide you with all the information you need to get the most out of the date/time values in your database.

Understanding Data Types

Date and time values are "special" kinds of information. Although we frequently refer to them in numeric form (09/18/1959 or 12:00) they obey special rules that do not constrain "true" numbers. For instance, date values must obey rules like:

Month values must be integers between one and twelve

Day values must be integers greater than zero and less than 28, 30, or 31 (depending on the month value)

February 29 is sometimes (but not usually) allowed

Whenever addition or subtraction involving a day (or month) value results in a "carry" operation, the month (or year) value is adjusted up or down accordingly

Time values must obey special rules like:

Minute and second values must be integers ranging from 0 to 59

Hour values can range from 0 to 24

Times between midnight and noon have an AM designation, those after noon and before midnight receive PM designation

The instant known as midnight can be expressed as either 00:00 AM (00:00:00) or 12:00 AM (24:00:00)

Whenever addition or subtraction involving a second (or minute) value results in a "carry" operation, the minute (or hour) value is adjusted up or down accordingly

In order for the System i to enforce these rules, it must "know" which fields represent dates, which fields represent times, and which fields contain other types of information. The nature of the information stored into a field is referred to as the *data type* of the field. The System i supports only five different data types, each with its own rules:

Character string (or **alphanumeric**) fields can contain any values that can be entered into the computer. String information can be combined (concatenated) or separated (substring) to form new alphanumeric fields.

Numeric fields can only contain numbers. Several classes (or sub-types) of numeric values are supported by the System i. They are:

Packed and Zoned decimal
Binary integer (small and large)
Floating point (single and double precision)

These forms differ mainly in the range of values they can represent. Small binary integers for instance can only hold integer values between -32768 and 32767. Packed and zoned decimal values can be up to 31 digits in length. Floating point fields can represent very large and very small numbers through the use of mantissa and exponent (n.nnnE+mmm) notation.

Numeric values can be combined with other numeric values (even though they may have different forms) into expressions that involve mathematical operators and functions.

Date fields can only contain valid date values. They must be values between January 1, 0001 and December 31, 9999

Time fields can only contain valid time values. They must be values between 00:00:00 and 24:00:00

Timestamp fields contain both a date and time component in addition to a microsecond (0.000001 second) portion.

In order to use Sequel successfully it is important to realize two facts about your database. First, that each field has a specific data type that determines rules that will be enforced and values it can hold. Second, that the type of data implies which operations are (and are not) allowed for the field.

Note: Generally speaking, data values of different types cannot be combined in an expression unless they are first converted to the same data type.

For instance, a string value such as "400" cannot be added to the number 400 unless the number is first converted into a number through the use of the INTEGER function. Thus,

```
"400" + 400                is wrong, but
INTEGER("400") + 400          is allowed
```

because the INTEGER function creates a numeric result from the string expression.

Date, Time, And Timestamp Formats

Sequel lets you access date and time values in a variety of different formats. The format can be chosen by each user for each request, although most users will usually prefer the same form for all requests. Eight different date formats are available. Each is known by its identifying type, shown below:

Type	Form	Example
`*MDY`	mm/dd/yy	12/31/09
`*YMD`	yy/mm/dd	09/12/31
`*DMY`	dd/mm/yy	31/12/09
`*JUL`	yy/ddd	09/366
`*USA`	mm/dd/yyyy	12/31/2009
`*EUR`	dd.mm.yyyy	31.12.2009
`*ISO,*JIS`	yyyy-mm-dd	2009-12-31
`*JL1`	yyyyddd	2009366

Likewise, three different time formats can be used. They are:

Type	Form	Example
`*USA`	hh:mm xx *(where xx is AM/PM)*	02:15 PM
`*HMS,*JIS`	hh:mm:ss	14:15:27
`*ISO,*EUR`	hh.mm.ss	14.15.27

All timestamp information is presented in a single 26 character format like yyyy mm dd hh.mm.ss.dddddd (ex. 2009-12-31-14.15.27.345302)

The format that Sequel will use when interpreting date and time literals (in the SELECT, WHERE, and HAVING clauses) and when date and time values are presented (through the DISPLAY, PRINT, and REPORT commands) depends on the "preferred" date/time style that is used when your request is run. The preferred style that you want to use is specified with the DTSTYLE keyword of the various commands. It allows you to specify four separate elements that control formatting. All four elements must always be specified. They occur in this order:

the date format that will be used when interpreting date literals and presenting date values. It must be one of the values listed above.

the date separator character, such as a slash (/), dash (-), or period (.), that should appear between elements of the date field. (Valid only if date format is *MDY, *YMD, *DMY, or *JUL)

the time format that will be used when interpreting time literals and presenting time values.

the time separator character (such as colon (:), or period (.) that should appear between elements of the time field. (Valid only if time format is *HMS)

Date and time formats are independent. You can, for instance, choose to have dates represented in *USA form and time values represented in *EUR form by coding the parameter as `DTSTYLE(*USA *N *EUR *N)`. (`*N` is an OS/400 command notation causing the default value of a parameter to be used). Likewise, acquiring dates in yy-mm-dd format and time values as hh.mm.ss can be accomplished by using `DTSTYLE(*YMD '-' *HMS '.')`.

When a Sequel view is created, the four DTSTYLE elements (date format/separator, time format/separator) used on the Create View (CRTVIEW) command are stored in the view so that they may be used when the view is run. If the view was created through Viewpoint, the values for DTSTYLE will be retrieved from the user's defaults. The DTSTYLE of remote database and *LOCALSYS views (type SQLVIEWM) will always be *ISO.

When the view is run, these values can be used or different values can be supplied, depending on your preference. If the view is run from inside DSNVIEW, the DTSTYLE saved with the view will be used. If the view is run from outside DSNVIEW, the command running the request will control the four date style elements from the DTSTYLE parameter. The PRINT, BCHPRINT, DISPLAY, REPORT, BCHREPORT, RUNSCRIPT and BCHSCRIPT default to *JOB so dates will be presented in the same format as your job date. If the view is run from Viewpoint, the regional settings for date preference for the user on the PC will be used.

In addition to the reserved values mentioned above, several special values are allowed in the DTSTYLE parameter. Specifically,

> `*JOB` can be specified for the format and/or separator values. This causes the preferred format for your job to be used when interpreting literals and presenting data from the record. Each time you sign on to the System i, the system values QDATFMT, QDATSEP, and QTIMSEP are used to establish the initial settings for your job. Once established, they can be changed by specifying new values on the DATFMT, DATSEP, and TIMSEP parameters of the Change Job (CHGJOB) command. If *JOB is specified for any of the DTSTYLE elements on a Sequel request, your current job's values will be used for the corresponding element (date format/separator, time format/separator).
>
> `*FIELD` can be specified for the format and/or separator values. This causes the preferred format of each field's definition to be used when presenting data from the record. The values specified on the DDS keywords DATFMT and DATSEP determine each field's preferred format. If DTSTYLE(*FIELD) is specified, character strings used in date/time expressions must conform to one of the international formats (*USA, *ISO, *EUR, *JIS, *JL1) or to the default format for your job.
>
> `*NONE` can be specified for the separator values if the date or time format is *MDY, *YMD, *DMY, *JUL, or *HMS. Date and/or time literals used within the query must not have any separators. Data values are presented without any separators.

The default or "preferred" format of a field is specified when it is created. Sample DDS that creates a file with every possible date and time format is shown below.

```
5738PW1 VRM  920327            SEU SOURCE LISTING

 SOURCE FILE . . . . . . . TEST/QDDSSRC
 MEMBER  . . . . . . . . . DATEFMTS

 SEQNBR*...+... 1 ...+... 2 ...+... 3 ...+... 4 ...+... 5 ...+... 6 ...+... 7 ...+..
   100                R DATEFMTS
   101                  CHAR         10
   200                  DUSA           L         DATFMT(*USA)
   300                  DISO           L         DATFMT(*ISO)
   400                  DEUR           L         DATFMT(*EUR)
   500                  DJIS           L         DATFMT(*JIS)
   600                  DMDY           L         DATFMT(*MDY)
   601                  DDMY           L         DATFMT(*DMY)
   700                  DYMD           L         DATFMT(*YMD)
   800                  DJUL           L         DATFMT(*JUL)
   900                  DDFT           L
  1000                  TUSA           T         TIMFMT(*USA)
  1100                  TISO           T         TIMFMT(*ISO)
  1200                  TEUR           T         TIMFMT(*EUR)
  1300                  TJIS           T         TIMFMT(*JIS)
  1400                  THMS           T         TIMFMT(*HMS)
  1800                  TDFT           T
               * * * *  E N D  O F  S O U R C E  * * * *
```

The source pictured above can be compiled into a file and records inserted into it. The following Sequel command requests all columns and rows from the file, with both date and time values presented in the preferred format defined in the database.

```
DISPLAY SQL('SELECT * FROM datefmts')
      DTSTYLE(*FIELD *N *FIELD *N)
```

The data could look like this:

CHAR	DUSA	DISO	DEUR	DJIS	DMDY	DDMY	DYMD
Today	01/21/1997	1997-01-21	21.01.1997	1997-01-21	01/21/97	21/01/97	97/01/21
New year	01/01/0001	0001-01-01	01.01.0001	0001-01-01	01/01/01	01/01/01	01/01/01
Christmas	12/25/1997	1997-12-25	25.12.1997	1997-12-25	12/25/97	25/12/97	97/12/25
April Fool	04/01/1998	1998-04-01	01.04.1998	1998-04-01	04/01/98	01/04/98	98/04/01
Leap day	02/29/1997	1997-02-29	29.02.1997	1997-02-29	02/29/97	29/02/97	97/02/29

DJUL	DDFT	TUSA	TISO	TEUR	TJIS	THMS	TDFT
97/021	1997-01-21	03:01 PM	15.01.12	15.01.12	15:01:12	15:01:12	15.01.12
01/001	0001-01-01	00:00 AM	00.00.00	00.00.00	00:00:00	00:00:00	00.00.00
97/360	1997-12-25	04:19 PM	16.19.39	16.19.39	16:19:39	16:19:39	16.19.39
98/091	1998-04-01	04:20 PM	16.20.14	16.20.14	16:20:14	16:20:14	16.20.14
97/060	1997-02-29	04:20 PM	16.20.39	16.20.39	16:20:39	16:20:39	16.20.39

Each format is different because columns are shown in the format identified as the field's preferred form. Date/time columns without a specific format (DDFT, TDFT) receive *ISO formatting. If the same request is run with `DTSTYLE(*USA *N *USA *N)`, each field will be shown in USA format like this:

CHAR	DUSA	DISO	DEUR	DJIS	DMDY	DDMY	DYMD
Today	01/21/1997	01/21/1997	01/21/1997	01/21/1997	01/21/1997	01/21/1997	01/21/1997
New year	01/01/0001	01/01/0001	01/01/0001	01/01/0001	01/01/2001	01/01/2001	01/01/2001
Christmas	12/25/1997	12/25/1997	12/25/1997	12/25/1997	12/25/1997	12/25/1997	12/25/1997
April Fool	04/01/1998	04/01/1998	04/01/1998	04/01/1998	04/01/1998	04/01/1998	04/01/1998
Leap day	02/29/1997	02/29/1997	02/29/1997	02/29/1997	02/29/1997	02/29/1997	02/29/1997

```
DJUL        DDFT        TUSA      TISO      TEUR      TJIS      THMS      TDFT
01/21/1997  01/21/1997  03:01 PM  03:01 PM  03:01 PM  03:01 PM  03:01 PM  03:01 PM
01/01/2001  01/01/0001  00:00 AM  00:00 AM  00:00 AM  00:00 AM  00:00 AM  00:00 AM
12/25/1997  12/25/1997  04:19 PM  04:19 PM  04:19 PM  04:19 PM  04:19 PM  04:19 PM
04/01/1998  04/01/1998  04:20 PM  04:20 PM  04:20 PM  04:20 PM  04:20 PM  04:20 PM
02/29/1997  02/29/1997  04:20 PM  04:20 PM  04:20 PM  04:20 PM  04:20 PM  04:20 PM
```

Now, each field has the same format. Presenting the same request with DTSTYLE(*JOB *JOB *JOB *JOB), each field will be shown using the running job's format. If the job's date format is MDY, the output will like this:

```
CHAR        DUSA      DISO      DEUR      DJIS      DMDY      DDMY      DYMD      DJUL
Today       01/21/97  01/21/97  01/21/97  01/21/97  01/21/97  01/21/97  01/21/97  01/21/97
New year    01/01/01  01/01/01  01/01/01  01/01/01  01/01/01  01/01/01  01/01/01  01/01/01
Christmas   12/25/97  12/25/97  12/25/97  12/25/97  12/25/97  12/25/97  12/25/97  12/25/97
April Fool  04/01/98  04/01/98  04/01/98  04/01/98  04/01/98  04/01/98  04/01/98  04/01/98
Leap day    02/29/97  02/29/97  02/29/97  02/29/97  02/29/97  02/29/97  02/29/97  02/29/97
```

```
DDFT      TUSA      TISO      TEUR      TJIS      THMS      TDFT
01/21/97  15:01:12  15:01:12  15:01:12  15:01:12  15:01:12  15:01:12
01/01/01  00:00:00  00:00:00  00:00:00  00:00:00  00:00:00  00:00:00
12/25/97  16:19:39  16:19:39  16:19:39  16:19:39  16:19:39  16:19:39
04/01/98  16:20:14  16:20:14  16:20:14  16:20:14  16:20:14  16:20:14
02/29/97  16:20:39  16:20:39  16:20:39  16:20:39  16:20:39  16:20:39
```

Notice that values for the "New year" record are missing in the DUSA, DISO, DEUR, DJIS and DDFT columns. The date values in the record reflect January 1, 0001 (see earlier examples). Date values having a year prior to 1940 or after 2039 cannot be represented in MDY, DMY, YMD, or JUL format and so they are dropped from the display or report. Although they can hold any valid date value, columns with a preferred style of MDY, DMY, YMD, or JUL will not be able to present dates outside this range unless the format is changed to one showing the full four digit year value.

It is important to note that all valid dates can be stored in all date fields, regardless of their preferred format. Furthermore, comparisons and expressions involving date/time values are completely independent of the preferred form of the column.

Automatic Conversion Of Character Strings

Your Sequel requests will often need to include specific dates, times, or timestamps in comparison or calculation expressions. If these values are entered as character strings they will be automatically converted to the appropriate data type needed for the expression.

Since date and time values can be shown in many different formats, it makes sense that they can be entered into Sequel statements in many forms as well. Sequel will automatically convert strings entered in a proper format into values that have a date, time or timestamp data type. Date and time values used within a Sequel request must appear as strings (surrounded by quotation marks) in either:

the "preferred" form (format and separator) specified on the DTSTYLE keyword of the Create View (CRTVIEW) command, or

*USA, *ISO, *EUR, *JIS, or *JL1 form

Timestamp values must have with their full 26 character format. Styles may be mixed within a Sequel request, as long as the above rule is not violated. Consider this query request:

```
DISPLAY ('SELECT * FROM filename WHERE
          datecol BETWEEN "12.01.1997" AND "12.31.97" OR
          datecol="04.01.1998"')
DTSTYLE(*MDY '.' *HMS *NONE)
```

Though it is an extreme example, the query will work correctly. If datecol is a column with a date (or timestamp) data type, the query selects only records having a datecol value in December 1997 or equal to April 1, 1998. Notice that each of the date literals used in the query conform to one of the "international" standards (*USA, *ISO, *EUR, *JIS or *JL1) or to the format specified by the DTSTYLE elements. The first literal is presented in the *USA format, the second appears in *MDY form with period separators, and the last literal is in the *EUR format.

As already discussed, the DTSTYLE elements control the output formatting for date and time values as well. When the query above is run, date values will be presented in month, day, year format separated by periods. December 1, 1998 will appear as 12.01.98 for example. Time values will appear without any separators. Six o'clock in the evening will appear as 180000.

Duration

Working successfully with date/time data types involves an understanding of another concept known as the **duration**. A duration is simply a number representing an interval of time. It is important to understand that a duration represents an *interval* of time, whereas an expression having a date/time/timestamp data type represents an *instant* of time.

There are three types of durations. All are expressed with a numeric data type. Any numeric value having the correct length and attributes can serve as a duration.

Date durations are packed numeric (8,0) values that represent a number of years, months and days. They have a *yyyymmdd* form.

Time durations are (6,0) numeric values that reflect a number of hours, minutes, and seconds using *hhmmss* form.

Timestamp durations use a (20,6) numeric value to represent a complete interval to microsecond precision. The complete form is *yyyymmddhhmmss.mmmmmm*.

Durations are useful in date/time arithmetic for incrementing and decrementing date/time values by an interval. They are "created" in two ways.

A duration always results whenever two columns of like data type are subtracted. For instance, the expressions

```
datecol1 - datecol2
CURRENT DATE - "11/30/1996"
```

both produce date durations. The resulting values are numeric integers with 8 digits that reflect the number of years, months, and days between the date values indicated. If the `CURRENT DATE` value is May 1, 1998 the result of the second expression would be the number 20,501 - 2 years, 5 months and 1 day.

Durations can also be specified by following a numeric value or expression with one of the following duration suffix words:

```
YEARS/YEAR
MONTHS/MONTH
DAYS/DAY
HOURS/HOUR
MINUTES/MINUTE
SECONDS/SECOND
MICROSECONDS/MICROSECOND
```

This latter form is known as a labeled duration. The following examples create labeled durations that can be used in arithmetic expressions with date, time, or timestamp values.

```
3 MONTHS
(end-start) HOURS
daycnt DAYS
```

The first example creates a date duration of 3 months. It has a numeric value of 300 (leading zeros omitted) since both the year and day portion of the duration is zero. Because it is a date duration, it can be added to (or subtracted from) a date or timestamp value.

The second example subtracts two numbers and creates a time duration. This (6,0) number can be combined in an expression with time or timestamp values in the same way that date durations can work with date values.

The final example suffixes a numeric column (daycnt) with the reserved word DAYS to create a date duration.

External Date, Time, and Timestamp Values

Several reserved fields can be accessed within your Sequel statement to acquire information about the current operating environment. The external values can be used in any part of a Sequel statement that date/time/timestamp column values or literals can appear.

CURRENT DATE
CURRENT TIME
CURRENT TIMESTAMP
CURRENT TIMEZONE

These fields return the current System i system values for date, time, and timezone. The data type and length of the returned values match the data type implied by the name; date, time, or timestamp. CURRENT TIMEZONE is returned as a packed decimal (3,0) value.

When the query is run, the appropriate value is accessed from the current machine information. The value is accessed only one time and will not change as the query progresses. Each row retrieved by the query will have the same value for the indicated field.

Date, Time, and Timestamp Expressions

There are essentially three types of operations that can be performed with date/time values: comparison, arithmetic, and special functions.

Comparisons are allowed between columns and/or conforming literals having a common data type. All comparison operators that are valid for numeric data types are also valid for date data types: =, <, >, <>, <=, >=, BETWEEN, IN. Date/time values cannot be used in a LIKE or CONTAINS predicate.

Creating a query that involves date, time or timestamp comparisons is very easy. Fields can be compared just as any other database field (or expression) would be. Correct results are returned without any extra work involved in rearranging the date format, or compensating for end of month or end of year values. For instance,

```
SELECT cusno,cname,paydate,orderdate,ordno,orval
      FROM custmast,ordhead JOIN cusno.1=cusno.2
      WHERE orderdate>paydate
```

will show customer order information for all orders that have arrived since the customer's last payment.

The query below uses a labeled duration to select records from the customer master file with a last payment date (PAYDATE) more than 3 months prior to today's date.

```
SELECT * FROM custmast
      WHERE paydate+3 MONTHS < CURRENT DATE
```

Remember that comparisons can only be made between compatible data types. The first comparison below is allowed; it compares the order date against an acceptable character date form.

```
SELECT * FROM custmast
      WHERE paydate < "1997-11-01"
```

The next example creates an error because numeric values are not compatible with date, time, or timestamp data types.

```
 SELECT * FROM custmast
      WHERE paydate < 19971101
```

Before the numeric value can be compared with a date value it must be converted to a string having one of the acceptable forms, or to a value with a date data type. This can be accomplished using the Sequel functions that are described beginning on page 99.

Arithmetic Operations

There are three arithmetic operations allowed when using date/time values: subtraction, increment, and decrement.

Subtraction
Date - Date = Date duration
Time - Time = Time duration
Timestamp - Timestamp = Timestamp duration

Increment (allows either operand first)
Date + Date duration = Date
Time + Time duration = Time
Timestamp + any duration = Timestamp

Decrement
Date - Date duration = Date
Time - Time duration = Time
Timestamp - any duration = Timestamp

Subtraction

Date/time subtraction is the simplest date operation, resulting in the duration (the number of years, months and days) between to two operands. Subtracting an earlier date from a later one always yields a positive duration; subtracting a later date from an earlier one always yields a negative duration.

Sometimes, the result of a date subtraction can be confusing. The reason is that the resulting duration looks like any other (8,0) decimal value, but it has a "special" meaning because it represents elapsed time. The query and output below shows the elapsed duration between the current system date and the invoice date stored in a data table.

```
SELECT cname,paydate,CURRENT DATE,CURRENT DATE-paydate
      FROM custmast
```

Name	PAYDATE	Current Date	Elapsed Interval
MNB Corp.	11/04/97	03/12/98	408
NBCO Corporation Inc.	11/12/97	03/12/98	400
Obell Group Sales	12/09/97	03/12/98	303
Lim-Equipment Co.	12/03/97	03/12/98	309
Lawrence Design	11/26/97	03/12/98	316
Reay Corp	10/23/97	03/12/98	420
Rider Corp.	12/29/97	03/12/98	214
Taehnrich Corp	10/11/97	03/12/98	501

The fourth column in the result expresses the number of years, months, and days between the PAYDATE column and the CURRENT DATE value (March 12, 1998) in yyyymmdd form. Thus, there are 4 months, 8 days between the first pair of dates and 5 months and 1 day between the last. It should be obvious that, unless the special meaning of the duration column is known, its content can be quite mysterious!

You can go a long way towards eliminating this confusion through the use of a user defined edit code. User defined edit codes can be especially useful for editing special kinds of numeric information such as telephone numbers, social security numbers, etc. If one has not already been created, you should create a permanent edit code for durations using a command similar to this:

```
CRTEDTD EDTD(5) INTMASK('    &years&  &months&  &days') NEGSTS('-')
```

(Place 4 blanks prior to the first ampersand, and two blanks between each succeeding pair of ampersands)

Depending on your system, edit code 5 may already be in use. Choose an unused number between 5 and 9 for the duration code. Refer to the *CL Programmer's Guide* and *Control Language Reference* manuals for additional information about user defined edit codes.

Once the edit code has been created, it can be used to edit the result of a date subtraction and provide nicely formatted output. Compare the result of the example above with the following statement and example results.

```
SELECT cname,paydate,CURRENT DATE,
           CURRENT DATE-paydate EDTCDE(5)
      FROM custmast
```

Name	PAYDATE	Current Date	Elapsed Interval
MNB Corp.	11/04/97	03/12/98	4 months 08 days
NBCO Corporation Inc.	11/12/97	03/12/98	4 months 00 days
Obell Group Sales	12/09/97	03/12/98	3 months 03 days
Lim-Equipment Co.1	02/03/97	03/12/98	3 months 09 days
Lawrence Design	11/26/97	03/12/98	3 months 16 days
Reay Corp	10/23/97	03/12/98	4 months 20 days
Rider Corp.	12/29/97	03/12/98	2 months 14 days
Taehnrich Corp	10/11/97	03/12/98	5 months 01 days

There is one other important caveat about the use of durations. Even though they have a numeric data type, they must never be treated as "regular" numeric data. Consider the implications of the following query:

```
SELECT cstte, SUM(CURRENT DATE-paydate), SUM(payam)
      FROM custmast GROUP BY cstte
```

The Sequel user intends to accumulate the total payment amount and the total last payment age for customers in each state. Although the request will return a result (with no errors) the result will be wrong! The reason is that the accumulation of durations (yyyymmdd) happens according to the rules of arithmetic - not the rules of the calendar. Notice what can happen when duration values are added together.

1 year, 1 month, 99 days	00010199
plus one more day	+ 00000001
equals 1 year, 2 months and 0 days!	= 00010200

Subtraction Using DAYS Function

The DAYS function helps you determine the number of days between two dates. Instead of producing a duration (years, months, and days) like simple subtraction, it gives you an easy way to find a more usable difference between two dates.

```
SELECT cusno,cname,DAYS(CURRENT DATE)-DAYS(paydate)
      FROM custmast
```

The example above finds the number of days since the last payment made by each customer. The one below, converts a numeric date value in MDY format (the field does not have a date data type) to a date field and subtracts it from the current date to find the difference. The result will contain only rows with an ODCDAT value greater than 45 days old.

```
SELECT * FROM dspobjd
      WHERE DAYS(CURRENT DATE)-DAYS(CVTDATE(odcdat,mdy))>45
```

Accumulating a total difference in dates can be done correctly using the DAYS function. The next example uses the DAYS function to determine the number of days between the current system date and the payment date. These values are subtracted to determine the number of days between them.

```
SELECT cstte, SUM(DAYS(CURRENT DATE)-DAYS(paydate)),
           SUM(payam)
      FROM custmast GROUP BY cstte
```

Because the SUM function is accumulating only the total number of days between the dates, the query will return the correct results.

In some instances the you may want to determine the days between two dates, but without any week end dates. The WEEKDAYS function accepts two dates as input and returns the number of week days between them. Input values must be date data type.

```
SELECT WEEKDAYS(CVTDATE(20120101,ymd1), CURRENT DATE)
      FROM sqlexec
```

The calculation determines the integer representations of the two dates and returns the difference between them after first subtracting the number of weekend days that would otherwise be counted. For example:

- Mon - Tue is 1 day (24 hour period).
- Mon - Fri is 4 days.
- Fri - Sun is 0 days (no full weekdays).
- Fri - Mon is 1 day .
- Sun - Sun is 5 days.

Increment/Decrement

Date/time arithmetic also allows a date/time/timestamp value to be incremented or decremented by a duration. The result is a value of the same data type that has been changed by some interval. Adding or subtracting a number of months and a date has the same effect as turning pages on a calendar. The day portion of the result is unchanged unless doing so would cause an invalid date (such as September 31) in which case it is adjusted to reflect the end of the month.

Each of the examples below uses a duration to increment or decrement a date value. The resulting date value can be presented to the user or used in a comparison with another date.

```
SELECT * FROM custmast
      WHERE CURRENT DATE-30 days>paydate

SELECT * FROM ordhead
      WHERE CVTDATE(cooyr,coomn,coody)+1 month<CURRENT DATE

SELECT *.1,oidate+30 years COLHDG("Maturity" "date")
      FROM security
```

Increment or decrement expressions requiring complex durations with a number of years, months and days can also be created. The next example demonstrates both an increment and decrement in the same statement. The increment uses a complex labeled duration of 2 months and 27 days. The decrement uses a calculated duration obtained by subtracting two dates.

```
SELECT cname, paydate,
      paydate + 2 MONTHS + 27 DAYS, COLHDG("Forward")
      paydate - (CURRENT DATE-paydate), COLHDG("Backward")
      FROM custmast
```

Name	PAYDATE	Forward	Backward
MNB Corp.	11/04/97	01/31/98	06/27/97
NBCO Corporation Inc.	11/12/97	02/08/98	07/12/97
Obell Group Sales	12/09/97	03/08/98	09/06/97
Que Company Inc.	11/20/97	02/16/98	07/29/97
Maple Leaf Cemetery	12/09/97	03/08/98	09/06/97
Sports Shop	12/31/97	03/27/98	10/19/97
Taehnrich Corp	10/11/97	01/07/98	05/10/97

The "Forward" column shows the result of incrementing the payment date by 2 months and 27 days. The month part of the date is incremented first and the result is then incremented by 27 days. Notice that the results might have been different if the payment date had been incremented by 27 days + 2 months (which would have returned 02/01/98 for the first record).

The "Backward" column shows the result of decrementing the payment date by the duration between the system date (March 12, 1998) and the payment date. The subtraction creates a decimal (8,0) value reflecting the number of years, months and days between the payment date and the current date. This duration is then subtracted from the payment date to create a date value prior to it.

Date, Time, and Timestamp Functions

Several Sequel functions are provided to assist with conversion, extraction, and presentation of date, time, and timestamp values.

Extract functions

Eleven functions are provided to recover just a portion of a date, time, or timestamp expression. Input to the function can be a column name, a literal value, or an expression. It must have a date, time, or timestamp data type, or it must be an appropriate duration. An integer representing the requested portion of the value is returned.

Date/Timestamp/Duration

`YEAR(expression)`	returns the year (4,0) part
`MONTH(expression)`	returns the month (2,0) part
`DAY(expression)`	returns the day (2,0) part
`DAYOFWEEK(expression)`	returns numeric 1-7 with Sunday as 1
`DAYOFYEAR(expression)`	returns value 1-366 depending on date
`QUARTER(expression)`	returns value 1-4
`WEEK(expression)`	returns value 1-53

Time/Timestamp/Duration

`HOUR(expression)`	returns the hour (2,0) part
`MINUTE(expression)`	returns the minute (2,0) part
`SECOND(expression)`	returns the second (2,0) part

Timestamp/Duration

`MICROSECOND(expression)`	returns the microsecond (6,0) part

The query below extracts the month part of the last payment date and uses it in a grouping request. All customer records having a last payment date in a given month will be grouped together. The total payment amount for each group will be returned by the query.

```
SELECT MONTH(paydate) NAME(paymonth),SUM(payam)
      FROM custmast
      GROUP BY paymonth
```

A similar query can be used to show a historical payment trend by grouping records according to the number of months since their last payment date:

```
SELECT MONTH(CURRENT DATE-paydate) NAME(paymonth),SUM(payam)
      FROM custmast
      WHERE YEAR(CURRENT DATE-paydate)=0
      GROUP BY paymonth
```

The (CURRENT DATE-paydate) expression creates a duration reflecting the number of years, months and days between the system date and the last payment date in the customer record. The query selects only records that reflect a payment within the last year, then groups the records according to the number of months since the last payment date. Output might look like this:

Payment Age (months)	Value of payments
0	38,504.67
1	367,915.25
2	152,096.16
3	129,218.86

Creating and Representing Date/Time/Timestamp

Other functions can be used to create date, time, and timestamp columns, control their representation, and to determine the number of days since the beginning of the calendar.

`DATE(expression)`
returns a date value from string, numeric value or timestamp.

`TIME(expression)`
returns a time value from a string or timestamp expression

`TIMESTAMP(expression,expression)`
returns a timestamp given a timestamp string, or a date and time string. The second operand of the TIMESTAMP function specifies the time value to be placed in the resulting timestamp. If the second operand is not specified, the first operand must be a 26 position character string that specifies the entire timestamp.

Expression results can be *character strings* as long as they are in a recognizable format with an appropriate separator. That is, they must conform to either USA, ISO, JIS, EUR or JL1 form, or have the preferred date/time format and separator indicated by the DTSTYLE parameter on the request.

You can use the CHAR function to convert date, time and timestamp values to a fixed length character string having a specific format. This function effectively overrides the format specified on the DTSTYLE keyword.

`CHAR(expr[,type])`

The type must be one of the recognized date/time types. If it is not specified, JOB is assumed. If the expression returns a date, the type must be USA, ISO, EUR, JIS, MDY, YMD, DMY, JUL, JL1, or JOB. If the expression returns a time, the type must be USA, ISO, EUR, JIS, HMS, or JOB. If the expression returns a timestamp, the type must be SAA, TS1, or JOB.

We can get the exact output shown on 89 that was created by supplying *FIELD for the DTSTYLE elements, by specifying various formats in the CHAR function.

```
SELECT CHAR(dusa,USA), CHAR(diso,ISO), CHAR(deur,EUR),
          CHAR(djis,JIS), CHAR(dmdy,MDY), CHAR(ddmy,DMY),
          CHAR(djul,JUL), CHAR(tusa,USA), CHAR(tiso,ISO),
          CHAR(teur,EUR), CHAR(tjis,JIS), CHAR(thms,HMS),
          CHAR(tdft,ISO)
      FROM datefmts
```

Regardless of the DTSTYLE elements supplied with the Sequel request, the output from the query statement above will appear as shown in the first example on page 89.

A date expression can be converted into an integer that represents its offset from the beginning of the system calendar. The DAYS function can be especially useful for returning the number of days between two dates or timestamp values.

```
DAYS(expression)
```

The expression must be a date or timestamp value, or a string representation of a date. The result is the number of days since January 1, 0001. The Sequel request below returns the number of days between the current date and the last payment date in the customer record.

```
SELECT cusno,cname,paydate,DAYS(CURRENT DATE)-DAYS(paydate)
      FROM custmast
```

The expression cannot be phrased as DAYS(CURRENT DATE paydate) because the DAYS function does not accept a duration as an argument.

Converting Non-Date Representations

The System i has not always supported date, time, or timestamp data types. As a result, date and time values within your database are probably stored in some alternative form - having either a character string or numeric data type and a month, day, year or a year, month, day representation.

If you know how dates are stored within your database files, you can convert them into date type data within your view. Once you do, you can gain the advantages of date data types, performing comparisons and date arithmetic on them.

Sequel includes a date conversion function that makes it easy to convert the dates in your database into System i date type values. The CVTDATE function will convert any of 10 different types of date formats into a date data type value. The table below lists the date formats that can be converted using the CVTDATE function.

Type	Date Form	Example
MDY	mmddyy	123198
MDY1	mmddyyyy	12311998
DMY	ddmmyy	311298
DMY1	ddmmyyyy	31121998
YMD	yymmdd	981231
YMD1	yyyymmdd	19981231
CYMD	cyymmdd	0981231
JUL	yyddd	98365
JUL1	yyyydddd	1998365
CJUL	cyyddd	098365
separate fields	yy,mm,dd (numeric only)	98,12,31
	cc,yy,mm,dd	19,98,12,31
	yyyy,mm,dd	1998,12,31

The CVTDATE function will accept almost any valid numeric or character date provided that its length and format matches the one of the types shown above. Like other Sequel functions, CVTDATE can be specified in the SELECT, WHERE, or HAVING clause. You indicate the field (or expression) containing the date value and one of the types listed above, and Sequel will convert it to a System i date value. For instance,

```
CVTDATE(orddate,MDY)
```

will convert the field ORDDATE (order date) from its current month, day, year form to a System i date value. Any System i date operation or function will work successfully on the converted value.

Date types CYMD and CJUL use a *century digit* preceding the year value. The century digit must have a zero value for years between 1900 and 1999. It must have a one value for years between 2000 and 2999.

Date types without a century representation (MDY, DMY, YMD, JUL) are converted in such a way that year values between 40 and 99 will fall between 1940 and 1999, and year values between 00 and 39 will be converted to years 2000 to 2039.

If your dates are stored in separate fields, you can still use CVTDATE to convert them. CVTDATE can convert dates stored as three or four fields to a value with a date data type. You supply the field names containing the year, month, and day values and CVTDATE will create the date value from them. The expressions must be specified in year, month, and day order. A century value (18, 19, 20, etc.) may optionally precede the year specification. An example of this type of conversion may look like this:

```
CVTDATE(payyr,paymn,paydy)
```

When the year field is a two digit field, CVDATE will only accept a numeric field for the year. To work around this restriction, simply cast the field as a number:

```
CVTDATE(ZONED(yy,2,0),mm,dd)
```

Examples

The output file from the Display Object Description (DSPOBJD) command includes a "last used" field that records the date that an object was last accessed. The field is a 6 position character value in *mmddyy* form. Suppose that we want to find all objects in the QGPL library that have not been used in the last 2 months. CVTDATE makes it easy.

Begin by creating an output file to contain the QGPL information. Use the command:

```
DSPOBJD OBJ(QGPL/*ALL) OBJTYPE(*ALL) DETAIL(*SERVICE) OUTPUT(*OUTFILE)
          OUTFILE(QTEMP/OBJD)
```

Now you can use Sequel to show objects that haven't been used for two months (or any other length of time for that matter). Request the results at your display by using the following DISPLAY command:

```
DISPLAY SQL('SELECT * FROM qtemp/objd
          WHERE CVTDATE(odudat,MDY) < CURRENT DATE - 2 MONTHS')
```

You probably received several CPD4019 messages at the bottom of your display indicating that records were omitted because they didn't have valid dates in them. If you do a simple view of the records, you will find that the usage date field is sometimes blank (or even zero!). Because Sequel is expression oriented, it is easy to change the statement above to transform invalid date values into valid ones.

The GREATEST function can be used to replace the blank (or zero) values with a valid date value. Try this:

```
DISPLAY SQL('SELECT * FROM qtemp/objd WHERE
          CVTDATE(GREATEST("010140",odudat),MDY) <
          CURRENT DATE - 2 MONTHS')
```

By adding the expression GREATEST("010140",udate) to the statement, we replace blank values (or zeros) with the lowest mmddyy date that can be represented, namely 01/01/1940. Any records with a blank or zero UDATE value will be shown on your display since the invalid date is first translated to 01/01/1940 before being compared with the cutoff date 2 months ago.

Now order the list of objects in the OBJD file in creation date order. If you choose descending date order, the most recently created objects will be listed first. Normally this would be a difficult task since the creation date field (ODCDAT) is in *mmddyy* form. Ordering by ODCDAT will essentially disregard the year since the month part of the date is at the left. Objects created in December will be listed before those created in November regardless of the year they were created.

If the date is converted to a System i date value, ordering can take place properly. Try the following statement:

```
DISPLAY SQL('SELECT CVTDATE(odcdat,mdy) name(CRTDATE),*.1
          FROM qtemp/objd ORDER BY crtdate DESC
```

Other useful queries can be created from the object description data that you have created. Try the ones listed below and invent your own to become more familiar with Sequel's date capabilities.

List the objects with differing source and object creation dates:

```
DISPLAY SQL('SELECT odsrcd,odcdat,odobnm,odobtp,odobat
          FROM qtemp/objd WHERE odsrcd>" " AND
          CVTDATE(odcdat,mdy) <> CVTDATE(odsrcd,ymd)
```

List the objects that weren't included in the last save:

```
DISPLAY SQL('SELECT odsdat,odscmd,odobnm,odobtp,odobat
          FROM qtemp/objd
          WHERE CVTDATE(odsdat,mdy) <
                (SELECT MAX(CVTDATE(odsdat,mdy))
                FROM qtemp/objd)
```

To calculate the number of days between today's date and the invoice date:

```
DISPLAY SQL('SELECT DAYS(CURRENT DATE)-DAYS(CVTDATE(invdte,ymd1)
          NAME(AGE) LEN(5,0) FROM sequelex/invmast')
```

Converting non-Time representations

Sequel includes a time conversion function that makes it easy to convert the times in your database into System i time values. The CVTTIME function converts a single 6-digit or three 2-digit values into a time. Values may be supplied in either numeric or character form.

```
CVTTIME(hh,mm,ss)
CVTTIME(hhmmss)
```

Like other Sequel functions, CVTTIME can be specified in the SELECT, WHERE, or HAVING clauses.

Appendix

Translating Sequel Syntax to Standard SQL

Sequel version 10.0 has the ability to translate Sequel SQL syntax into standard SQL syntax based on a chosen server. This translation allows you to run your Sequel request locally, using the new, faster SQL Query Engine, or against remote databases such as SQL Server, Oracle, and MySQL using the SQL syntax native to these databases. You have complete control, through two execution parameters (SERVER and SYNTAX), on how to translate the SQL against a given database.

The Sequel language is a variation of standard SQL that runs using the OS/400 Classic Query Engine (CQE). The CQE is the original query processor with extensions that allow access to OS/400-specific features like multi-member and multi-format files. The SQL Query Engine (SQE) is the "new" query engine that uses many new optimization features which do not exist in the CQE. Although the SQE promises better performance over many queries, it sacrifices much of the OS/400-specific compatibility inherent to the CQE.

Many want to be able to continue using Sequel syntax because it offers several usability enhancements users want and have grown accustomed to like the JOIN statement, CVTDATE and column headings. At the same time, we want to be able to run Sequel syntax against the SQE, so Sequel objects can take advantage of the faster query processing speed, and we want (wherever possible) to be able to target the same Sequel statements against other remote databases, such as SQL Server, Oracle, and MySQL.

New Host Features for Syntax Conversion

There are two main host changes provided to support the conversion of Sequel statements from our extended Sequel syntax into standard SQL.

- The first is the addition of the SYNTAX parameter to any command that works with an SQL statement (such as CRTVIEW, DISPLAY, DSNREPORT, and others).

  Specifying SYNTAX(*SEQUEL) indicates the SQL statement is written using our extended SEQUEL syntax..

  Specifying SYNTAX(*SERVER) indicates the SQL statement is written in the syntax of the database (SEQUEL, MySQL, SQLServer, Oracle, etc.) specified on the SERVER parameter. No conversion from *SEQUEL to native SQL takes place.

- The second addition is the new server (SERVER) value *LOCALSYS. *LOCALSYS and *LOCAL both use the newer SQE however *LOCAL uses *SQL naming which means qualified names are entered as libname.filename and unqualified files must be in the library with the same name as the user running the query. *LOCALSYS uses *SYS naming-library/file qualification instead of library.file qualification. *SYS naming locates unqualified files using the job's library list.

  New UDF functions (IMG, INDEX, VERIFY, XLATE) have been added to support

*LOCAL/*LOCALSYS use of special functions previously available only when using SERVER(*SEQUEL).

Note: Remote database (SERVER) support will not work until the J2SE 5.0 JDK (57nnJV1 option 7-9) and IBM Toolbox for Java 57*nn*JC1 licensed programs are installed.
(where *nn* = 22 for V5 (Os/400 V5R3, V5R4, etc.), 61 for V6, and 70 for V7)

Review the CRTVIEW and DISPLAY commands in the *Sequel 10 Programmers Guide* for a complete explanation of the SYNTAX and SERVER parameters.

Client Changes Required to Support Sequel Conversion

There are also several changes for the Viewpoint client GUI interface.

- The SYNTAX parameter has been added to the object property dialog in Viewpoint and allows the values *SEQUEL and *SERVER. The default syntax for Sequel is *SEQUEL and all other servers will default to *SERVER. The *SEQUEL syntax allows the use of Sequel SQL conventions like the JOIN clause.
- The new Database Server (SERVER) value *LOCALSYS has been added to the database selection list in the object property dialog of Viewpoint. This server uses system naming (lib/file) in contrast to *LOCAL which uses SQL naming (lib.file). *LOCALSYS requests will use the library list of the Viewpoint job.
- The Expression Editor in the Viewpoint designer will combine the list of server specific functions and the supported Sequel functions.

Conversion Blockers and Manual Resolutions

The conversion routine will accept a Sequel Query and the join type as input, and will output an SQL query. There are a couple items, or "blockers" that prevent conversion from Sequel syntax to standard (*LOCAL) syntax, and will have to be dealt with "as is".

1. Sequel syntax allows the selection of members and formats in the FROM clause.

   ```
   SELECT *
       FROM FILE(MEMBER) FORMAT
   ```

   Standard SQL only allows access from the first member of a file, and does not allow access from any file (LF) that has more than one format. Any Sequel statement that specifies either a member or format name cannot be translated.

   To resolve this situation you can do one of the following:

   - For multi-member access, a user can create an alias (using SQL) that selects from that file:

     ```
     CREATE ALIAS LIBRARY.FILE_MEMBER FOR LIBRARY.FILE(MEMBER)
     ```

     Then, simply select from that alias:

```
SELECT *
     FROM FILE_MEMBER
```

- Most multi-format logicals are simply a UNION of results between two files or two members of the same file. Instead of selecting a single format of a multi-format logical file like so:

  ```
  SELECT *
       FROM MULT_FORMAT_LF(FORMAT)
  ```

  Select the data from the physical file that the format is associated with.

  ```
  SELECT *
       FROM PF_THAT_FORMAT_IS_BUILT_OVER
  ```

  Of course, if there is select/omit criteria in the multi-format logical (such as this select/omit DDS that was part of the same format):

  ```
  O ONAM                      COMP(EQ '          ')
  S OLIB                      COMP(EQ 'QSYS')
  ```

  The selection criteria should be duplicated with a corresponding WHERE clause:

  ```
  SELECT *
       FROM PF_THAT_FORMAT_IS_BUILT_OVER
       WHERE ONAM<>' ' AND OLIB='QSYS'
  ```

2. Sequel allows the selection of ambiguous field names. If the same field name exists in more than one file, it is "ambiguous" to select that field name without qualifying it.

   ```
   SELECT CUSNO, SUM(ORVAL)
        FROM SEQUELEX/CUSTMAST, SEQUELEX/ORDHEAD
        JOIN BY CUSNO.1=CUSNO.2
        WHERE CUSNO=100200
        GROUP BY CUSNO
        ORDER BY CUSNO
   ```

   Here, Sequel will leave ambiguous field names unqualified.

   The Java code that Sequel uses to translate the statement to standard SQL does not have the list of fields from each file available to it, so it cannot qualify the resulting SQL:

   ```
   SELECT CUSNO, SUM(ORVAL) AS DERIVED_01
        FROM SEQUELEX/CUSTMAST C0,SEQUELEX/ORDHEAD 01
        WHERE C0.CUSNO=01.CUSNO AND CUSNO=100200
        GROUP BY CUSNO
        ORDER BY CUSNO
   ```

   and generates an error: *'Error SQL0203 - Name CUSNO is ambiguous.'*

The solution is fairly simple—ensure that all ambiguous field names are qualified.

```
SELECT CUSNO.1,SUM(ORVAL)
     FROM SEQUELEX/CUSTMAST, SEQUELEX/ORDHEAD
     JOIN BY CUSNO.1=CUSNO.2
     WHERE CUSNO.1=100200
     GROUP BY CUSNO.1
     ORDER BY CUSNO
```

Both the green-screen DSNVIEW and Viewpoint designers do this automatically, so almost all views created by users should already have correctly qualified file names.

Note that fields in the ORDER BY clause are not ambiguous as long as they are identified in the SELECT clause. But if you are ordering results based on a field that is not included in the SELECT clause, then you must also qualify that field in the ORDER BY clause if it occurs in more than one file.

3. Sequel allows files to be joined using derived fields (standard SQL does not).

```
SELECT CUSNO.1, WDATA(DIGITS(CUSNO.2)) NAME(cusno2), ORDNO.2
     FROM SEQUELEX/CUSTMAST, SEQUELEX/ORDHEAD
     JOIN BY CUSNO.1=CUSNO2
```

For an inner join (JTYPE(*INNER)), the same SQL statement can be rearranged to use the WHERE clause instead of the JOIN clause since the end result is identical and this statement can be successfully converted.

```
SELECT CUSNO.1, WDATA(DIGITS(CUSNO.2)) NAME(cusno2), ORDNO.2
     FROM SEQUELEX/CUSTMAST, SEQUELEX/ORDHEAD
     WHERE CUSNO.1=CUSNO2
```

For join types other than *INNER, no direct method in SQL is equivalent; instead, an SQL View logical file must be manually created and referenced in the SQL statement.

```
CREATE VIEW SEQUELEX.CUSNO2 (CUSNO2,ORDNO) AS
SELECT digits(CUSNO), ORDNO
     FROM SEQUELEX.ORDHEAD

SELECT C1.CUSNO,CUSNO2,ORDNO
     FROM SEQUELEX.CUSTMAST c1
     LEFT OUTER JOIN sequelex.cusno2 c2 ON c1.cusno = c2.cusno2
```

Note: *With Sequel version R10M05 (and higher), this item will no longer prevent conversion. However, when fields are selected from the secondary file and there is not a corresponding record in the secondary file, the result is a null value (n/a) whereas with *SEQUEL the result is the field's default value (usually blank or zero). Having null values can affect record selection, calculated results, and result in a level check error when creating a physical file.*

4. The parameter UNIQUEKEY can be specified in a Sequel view to return the first record in a series that matches a given ordering value if there is an ORDER BY clause in the view.

This function is not supported for any other server (will not convert), and will generate an error.

5. Remote database functions that use DB2 or *SEQUEL keywords as input will cause conversion problems if the remote database function is used in a *SEQUEL syntax view.

   As an example, the following SQL Server CONVERT function with *SEQUEL syntax can fail:

   ```
   convert(VARCHAR(10),srv_date,101)
   ```

   This is a valid expression, but since VARCHAR is a DB2 scalar function, we must convert it to a CAST expression like so:

   ```
   convert(CAST(10 AS VARCHAR),srv_date,101)
   ```

   The result is invalid input to the CONVERT function.

6. Sequel syntax doesn't allow empty function calls like getdate().

   getdate() is a valid SQLServer function that returns the current date. Remote server functions can generally be used in Sequel syntax statements, unless they require an empty parameter list.

7. Dynamic Drill Down options are not available for GROUP BY views with hidden fields (WDATA).

Sequel Function Compatibility

The table below lists each Sequel function and its compatibility (and ability to translate) to SQL Server, Oracle, MySQL, and DB2.

These functions are the same in Sequel (as documented) and the standard DB2/400 implementation of SQL. Functions that exist or are supported in other remote databases (SQL Server, Oracle, MySQL) are checked with an 'X'. A blank indicates the function is not supported for a given database. If a database has an equivalent function, but with a different name, the name is listed.

(X=supported, blank=not supported, 'name'=supported equivalent)

Sequel Function	SQL Server	Oracle	MySQL	DB2
ABS(num)	X	X	X	X
ABSVAL	X	X	X	X
ACCUM				X
ACOS				
ANTILOG				
ASCII				
ATAN				
ATAN2				
ATANH				
AVG Scalar	X	X	X	X
AVG([DISTINCT or ALL] expr)	X	X	X	X
BCAT		X	X	X
BIGINT(expr)	X	X	X	X
BIT_LENGTH			X	
CASE WHEN,THEN END	X	X	X	X

(X=supported, blank=not supported, 'name'=supported equivalent)

Sequel Function	SQL Server	Oracle	MySQL	DB2
CAT		X	X	X
CEIL		X		
CEILING(expr)	X	CEIL	X	X
CENTER				X
CHAR(expr [,dec char])	X	X	X	X
CHAR(datetime [,date format])		TO_CHAR	X	X
CHAR_LENGTH	LEN	X	LENGTH	X
CHAR2NUM				X
CHARACTER_LENGTH	LEN	X	LENGTH	X
CHR				
COALESCE	X	X	X	X
CONCAT		X	X	X
COS				
COSH				
COT				
COUNT(*)	X	X	X	X
COUNT([DISTINCT or ALL] expr)	X	X	X	X
CVTDATE				X
CVTTIME				X
CYYDDD				X
CYYMMDD				X
CYYMMDD1				X
DATAPARTITIOINNUM				
DATAPARTITIONNAME				
DATE(expr)		X	X	X
DAY(date-expr)	X	X	X	X
DAYNAME				
DAYOFMONTH				
DAYOFWEEK(expr)			X	X
DAYOFWEEK_ISO				
DAYOFYEAR(expr)			X	X
DAYS(expr)				X
DDMMYY				X
DDMMYYYY				X
DEC				
DECIMAL(expr[,len[,dec]])	X	X	X	X
DECRYPT_CHAR				
DECRYPT_SQ				
DEGREES				
DIFFERENCE(expr, expr)	X			X
DIGITS(expr)				X
DIST_KM				
DIST_MILE				
DIST_REF_KM				
DIST_REF_MILES				
DOUBLE	X	X	X	X
DOUBLE_PRECISION	X	X	X	X
DTAARA				
EDIT				
ENCRYPT_RC2				

(X=supported, blank=not supported, 'name'=supported equivalent)

Sequel Function	SQL Server	Oracle	MySQL	DB2
ENCRYPT_SQ				
EXP(expr)	X	X	X	X
EXP10(expr)				
FLOAT(expr[,len])	X	X	X	X
FLOOR	X	X	X	X
FMTMSG				
GETHINT				
GETHINT_SQ				
GREATEST		X	X	X
HASH(expr, expr...)				
HEX(expr)				X
HOUR(expr)			X	X
HREF				X
HYPERLINK				
IFNULL				
IMG				
IMGREF				X
INDEX				
INSERT				
INT				
INTEGER(expr[,len])	X	X	X	X
JULIAN_DAY				X
JUSTIFY				X
LAND(expr,expr...)		BITAND	BIT_AND	
LCASE				
LEAST		X	X	X
LEFT				
LENGTH(expr)	LEN	LENGTHB	X	X
LN(expr)	LOG	X	LOG	X
LNOT(expr,expr...)				
LOCATE				
LOG	X	X	X	X
LOG10				
LOR(expr,expr...)			BIT_OR	
LOWER(expr)	X	X	X	X
LTRIM		X	X	X
LXOR(expr,expr...)			BIT_XOR	
MAX([DISTINCT or ALL] expr)	X	X	X	X
MICROSECOND(expr)			X	X
MIDNIGHT_SECONDS				X
MIN([DISTINCT or ALL] expr)	X	X	X	X
MINUTE(expr)			X	X
MMDDYY				X
MMDDYYYY				X
MNTHNAME				X
MOD				
MONTH(date-expr)			X	X
MONTHNAME				
MULTIPLY_ALT				
NODENAME				

(X=supported, blank=not supported, 'name'=supported equivalent)

Sequel Function	SQL Server	Oracle	MySQL	DB2
NODENUMBER				
NULLIF				
OCTET_LENGTH				
PARTITION				
PCTCHG(expr, expr)				X
POSSTR(expr-1, expr-2)	CHARINDEX	LOCATE	LOCATE	X
POWER				
PREV				X
PROPER				X
QUARTER(date-expr)			X	X
RADIANS				
RAISE_ERROR				
RAND(seed-value)	X	X	X	X
REAL	X	X	X	X
REPEAT				
REPLACE				
ROUND	X	X	X	X
ROWID				
RTRIM		X	X	X
SDEV	X	X	X	X
SECOND(expr)			X	X
SGFMTIME				
SIGN	X	X	X	X
SIN				
SINH				
SMALLINT(expr)	X	X	X	X
SOUNDEX	X	X	X	X
SOUNDEX(expr)	X	X	X	X
SPACE				
SPLIT				X
SQRT(expression)	X	X	X	X
SST	SUBSTRING	X	X	X
STRIP(expr[,type[,char]])				X
STRIPX				X
SUBSTR(expr,start,len)	SUBSTRING	X	X	X
SUBSTRING				
SUM([DISTINCT or ALL] expr)	X	X	X	X
TAN				
TANH				
TCAT		X	X	X
TIME(expr)		X	X	X
TIMESTAMP(expr, expr)	X	X	X	X
TIMESTAMP_ISO				
TIMESTAMPDIFF				
TRANSLATE(expression, [to-string, [from-string, [padcharacter]]])				X
TRIM		X	X	X
TRUNC				
TRUNCATE				
UCASE				

(X=supported, blank=not supported, 'name'=supported equivalent)

Sequel Function	SQL Server	Oracle	MySQL	DB2
UNEDIT				X
UNPACK				X
UPPER(expr)	X	X	X	X
URLSTRING				X
VALID_DATE				
VALID_TIME				
VALID_TSTP				
VALUE	X	X	X	X
VAR([DISTINCT or ALL] expr) or VARIANCE	VAR	VARIANCE	VARIANCE	X
VARCHAR	X	X	X	X
VERIFY				
WEEK(date-expr)			X	X
WEEK_ISO				
XLATE				
XOR				
YEAR(date-expr)			X	X
YYMMDD				X
YYYYDDD				X
YYYYMMDD				X
ZONED(expr[,len[,dec[,dec_char]]])				
ZONED(expr[,len[,dec]])				

Note: DB2/400 is a superset of the DB2 standard and certain functions exist in DB2/400 that are not compatible across all DB2 databases (DB2 UDB (Windows/Linux), DB2 z/os, DB2/400).

Limits within Sequel

Sequel has limits within clauses, functions as well as output amounts. For the most part, these limits will not prohibit you from accomplishing most tasks.

View Limits

These limits may affect you during the design process of views.

View Size

The overall SQL statement can be a maximum of 20,000 positions. To determine the view size, select the view and choose **File>Properties** from the Viewpoint Explorer menu.

Number of Files

Each view can have up to 32 files. In UNION views, if each FROM clause has only one file, 32 UNION clauses are allowed.

Expressions

- Each expression for a derived field on the SELECT clause can be up to 2,000 positions.
- The IN comparison operator used on the WHERE clause can have up to 50 comma separated values.
- The CASE function can contain up to 48 pairs of When/Then conditions.

Variables

- A view can have 50 unique prompt variables. There can be up to 110 variable used in a view. So 14 of the unique variables can be used twice in the view. You might need to place the variable on the SELECT and WHERE clause—this would count as two uses.
- The maximum length of a QSTRING variable is 256, EXPR variable is 1085, NUMBER variable is 29, NAME variable is 10, and DATE variable is 10.
- The Prompt Text value can be up to 32 positions and the Extended Help can be a maximum of 256.
- The maximum number of values displayed with the DBLIST integrity test is 8000 in Viewpoint, 800 is Sequel Web Interface version 5, and is unlimited in Sequel Web Interface version 10.

Results Limits

These limits relate to directing view output.

Display

The maximum allowed width of all displayed columns is 2000.

EXECUTE to PC File

The maximum size of a character field is 2000 positions.

E-Mail

The maximum size for an E-mail attachment is 16MB.

Host Report Limits

Aggregate Calculations

The maximum number of aggregate calculations (such as SUM) that can be created is 200.

Number of Fields

It is recommended the view/SQL statements use a maximum of 100fields on the SELECT clause.

Table Limits

The limits of the table are very robust and should rarely be an issue.

Table Size

The overall size of the table result must be less than 16 GB. The total size is calculated by:

((size in bytes of all dimension fields + maximum number of column groups * size of all column fields) * total number of records generated including break level) + 32K for fudge factor of work fields

Categories

- Maximum number of category fields is 32 with a maximum of 32 column fields per category.
- Maximum number of column groups is 2048 for Viewpoint and 9999 for the host version.

Dimensions

Maximum number of dimension fields is 20 or a total field length of 2000.

Script Limits

Script Size

- Total number of lines allowed in a script is 2000. A single command may require multiple script lines.
- The maximum length for a single command line is 9000 positions including substituted data.

Auditor Limits

QRYSUM File

The QRYSUM file which is populated during the Analyze Audit Data (ANZAUDDTA) process has a field named SMSQL. This field holds the SQL statement of the view. It's size is 2000 positions, so the SQL statement will truncate at 2000 positions. The QRYSUM file is used to audit usage of views and reports.

FAQ

General Questions

1. **How can I determine what version of Sequel is being used?**

   Use the command `SQVER`, or from the Viewpoint Explorer or any design screen select Help\About on the menu.

2. **How can the company name that appears on reports be changed?**

   The company name is stored in the first 40 positions of the ASC#SQ data area. It can be change with the following command:

   ```
   CHGDTAARA DTAARA(SEQUEL/ASC#SQ (1 40)) VALUE('New Company Name')
   ```

   If the new company name is longer than the old name, the additional characters will not be included until the report is edited. If the new name is substantially longer, you may want to re-center the @@CMPNAM field in the report editor.

   You can also prompt the Change Company Name (CHGCMPNAM) comand. Enter values for product name, company name value, and product install library like so:

   ```
   ASCSUPPORT/CHGCMPNAM PRODUCT(SEQUEL) COMPANY(Company Name)
    LIBRARY(LIB)
   ```

3. **Is there a way to limit the number of records an outfile can have?**

   The attributes for outfiles are copied from SQLEXEC file in SEQUEL. To change the initial size use:

   ```
   CHGPF SEQUEL/SQLEXEC SIZE(              )
   ```

   This change would have to be implemented after each Sequel installation.

4. **How can the use of the INSERT, UPDATE and DELETE commands be restricted?**

   First, these commands are shipped to have 'Allow Limited user' *NO. So if the user profile is defined as limited (no command line access), the user will not be allowed to use the commands. There are no menu options that correspond to these commands. They require a command line to be used. To restrict users who have command line access, use OS/400 object authority:

   ```
   EDTOBJAUT SEQUEL/UPDATE *CMD
   ```

   Adjust access for each command as necessary.

Errors

1. **I am using UPDATE on a date data type field and am receiving a CPF5035 'Data mapping error' due to reason 17: the format of the date in a date, time or timestamp literal is not valid.**

   This most often occurs when a date field is updated using the "default" date like 0001-01-01. When using date constants before 1940 or after 2039, you must change the DTSTYLE parameter on the UPDATE command to match the date constant:

   ```
   UPDATE SET((datefield 'DATE("0001-01-01")')) SQL('from sequelex/
   custmast') DTSTYLE(*ISO)
   ```

2. **I am running a view/report and received a CPF5035 'Data mapping error' due to reason 16: A date value is less than the minimum allowed value. What does it mean?**

   Most likely you are using the CVTDATE function to convert a numeric or character field to a date data type. The CVTDATE function can convert only valid values to a date. If there is a zero in your date field, this cannot be converted to a date data type. A valid value can be substituted for the zero value using the GREATEST function:

   ```
   CVTDATE(GREATEST(yourfield,19400101),ymd1))
   ```

3. **I am running a view and received a CPF5035 'Data mapping error' due to reason 9: Division by zero. What does this mean?**

   Division by zero occurred on a record. Division by zero is not mathematically allowed. Conditionally we have to work around the record(s) with zero in the denominator field. The CASE function works well to return a zero result:

   ```
   CASE WHEN denominator=0 THEN 0 ELSE numerator/denominator END
   ```

4. **I am running a view/report and received a CPF5035 'Data mapping error' due to reason 2: A significant digit was truncated. What does this mean?**

   The result of an expression is too large for the length of the field defined. Increase the length of the result field.

5. **I have submitted a job and am receiving a CPF4103 'Device *REQUESTER not found while opening file RUNDSPD in library SEQUEL'.**

   The view contains variables and the job was submitted before Sequel had a chance to prompt you. Try using one of our batch commands: BCHPRINT, BCHEXECUTE, BCHREPORT or BCHSCRIPT. Once you are prompted, you can use F14=Submit to send the job to a job queue.

6. **I am using EXECUTE to create a file and am receiving CPF4131 'Level check on the outfile name'.**

   The outfile you specified already exists and is different than the file Sequel is trying to create. Everything about the file that exists and the file we are trying to create must match including the number of fields, field names and field lengths. You can delete the file (DLTF)

and create it from scratch, use a different name for the outfile that does not exist or determine what is different and change your view.

7. **I am receiving a CPD4306 'Field x in VIEWFMT is not unique'.**

   The same field name is on the SELECT clause more than once. Even if the fields are from different files, it will cause an error. If you need both fields, assign as alias name to one of the fields:

   ```
   SELECT CUSNO.01, CUSNO.02 NAME(CUSNO2)
      FROM SEQULEX/CUSTMAST,SEQUELEX/ORDHEAD
      JOIN CUSNO.01=CUSNO.02
   ```

8. **I am generating spooled output using PRINT or REPORT and received CPA4072 'Reached maximum number of spooled records'.**

   Use CHGPRTF SEQUEL/SQLPRT1 MAXSPL(*NOMAX). There are actually eight printer files, SQLPRT1 through SQLPRT8.

   This change would have to be implemented after each Sequel installation.

9. **UPDATE gives CPD4305 'I/O attribute x'03' field UPD001A format VIEWFMT not valid'.**

   This error occurs when two or more files are referenced on the UPDATE command. Only records from the primary file can be updated. Modify the view or the SQL statement so that the fields being updated are from the first file on the FROM clause.

10. **What does CPI4313 'Join default values cannot be identified' mean?**

    This is an informational message that is sent with all partial outer (*PARTOUT) or only default (*ONLYDFT) joins. It means that a zero or blank used for the records without a match on the secondary file will be hard to distinguish from the actual values of zero or blank.

11. **What does QRY6031 'Character value 0 used in WHERE clause should be numeric' mean when doing something like** `SELECT * FROM SEQUELEX/CUSTMAST WHERE AMTDU>"0"`**?**

    Because AMTDU is a numeric field, any comparison should be made to a numeric value. Therefore remove the quotes around the 0 on the WHERE clause.

12. **We have a user based license and occasionally receive a CPF9E71 'Grace period expired. Requesting user not added.' And a CPF9E19 'Usage limit threshold exceeded for product Sequel.' What do these messages mean?**

    The CPF9E71 means that you are over your usage limit. If you have a six user license, this message would be sent when the seventh Sequel job was attempted.

    The CPF9E19 indicates that you are at your threshold limit which means you are close to running out of available user licenses.

    You can check who is using Sequel by doing WRKLICINF, then use option 8 next to the 0ASCSEQ product. To change the threshold limit, you can use option 2.

13. **I ran a report and received a RPT7010 'calculation name cannot receive summary result while processing record'. What does this mean?**

    The length of the calculation is not long enough to hold a subtotal or grand total amount. Report output will be generate, but some subtotal and the grand total values could be truncated. To fix the problem, go in the report editor, press F9=Calculations, press F5=Field Attributes and increase the length of the field named in the error message.

14. **I am using EXECUTE to a PC document in *XLS or*WKS format and received an MCH0601 unmonitored by QRYO07 and MCH0601 space offset X'00FFFF0F' or teraspace offset. What does it mean?**

    Output is written to a user space first before writing the worksheet stream file. This imposes a 16 meg limit on the amount of data. So reduce the number of records or fields in your selection.

How To Questions

1. **When creating a UNION view in DSNVIEW, how do I prompt the clauses after the UNION and before the ORDER BY?**

   Use F14=Prompt Clause. This function key is available at Intermediate and Advanced assistance levels.

2. **When creating a UNION view, is there an easy way to isolate the sections and run each section on its own?**

   At advanced assistance level, position your cursor to left of the desired SELECT clause and press F13. To return to the view in its entirety, use F12.

3. **I am trying to pass a character value in a character CL variable to the UPDATE command and can't get the quotes right.**

   In a CL program, character values supplied to the values parameter through CL variables must also be enclosed in double quotes. This means the double quote marks must be inserted into the character string contained in the CL variable. For example:

```
DCL    &CTYPE *CHAR 2
DCL    &CTYPE2 *CHAR 4
CHGVAR &CTYPE2 VALUE('"' *CAT &CTYPE *TCAT '"')
UPDATE SET((CTYPE &CTYPE2)) SQL('from
       sequelex/custmast where ctype="CS"')
```

4. **How can I update part of a character field? For instance, my customer telephone field is a character 10 field and I want to change some of the area codes.**

   Build the value as an expression.

```
UPDATE SET((CPHON '"773" CAT SST(CPHON,4,7)')) SQL('FROM SEQUELEX/
CUSTMAST WHERE SST(CPHON,1,3)="312" AND SST(CPHON,4,3)
IN("456","457")
```

5. **How can I combine a first name and last name field and remove the extra spaces?**

   ```
   BCAT(FNAME, LNAME)
   ```

6. **How can I combine the name fields, but make last name, first name?**

   ```
   BCAT(TCAT(LNAME,","),FNAME)
   ```

7. **How can I edit a social security number field with dashes and not suppress the leading zero?**

   If the field is numeric, create an edit word and increase the length of the field to 10,0:

   ```
   SELECT ssn LEN(10,0) EDTWRD("0   -  -    ")
   ```

   If the field is character, use the substring and concatenation functions:

   ```
   SELECT CAT(SST(ssn,1,3),"-",SST(ssn,4,2),"-",SST(ssn,6,4))
   ```

8. **Is there an easy way to select all the fields in a file and build an expression or two?**

   Yes, use SELECT * but qualify the asterisk with a file qualifier and continue the SELECT clause with an expression:

   ```
   SELECT *.1, CRLIM-AMTDU NAME(CRBAL) LEN(9,2)
          FROM    SEQUELEX/CUSTMAST
   ```

9. **How can I place a variable in the title of a report?**

   Simply place the variable (i.e. &date) in the title. If you are designing the report for the first time, you can place the variable on the Title line on the Level Break selection screen. Or if the report has already been created, go the to exit screen of the report and change the title to include the variable. To ensure that the variable does truncate, enclose the title in single quotes and use two ampersands for the variable name (i.e. 'Customer Report for &&date').

10. **How can I add a page break to an existing report.**

    A page break is equivalent to a Skip After 1 line control. Use the line skeleton on the format that you would like the page break. If you would like a page break for each customer, use F15=Select Format and select the customer format, place an S on the underscore mark next to the line number and press enter. On the last line of the screen, set the Skip After to 1.

11. **How can detail be suppressed in a report?**

    Go to the field attributes for the field by placing an asterisk on the first position of the field and change 'Print on overflow' to Y and 'Break>=' to the appropriate break level.

Index

E

F

O

P

Q

R

S