Caché MultiValue Query Language (CMQL) Reference

Version 2018.1
2024-11-07

InterSystems Corporation 1 Memorial Drive Cambridge MA 02142 www.intersystems.com

Caché MultiValue Query Language (CMQL) Reference
PDF generated on 2024-11-07
InterSystems Caché® Version 2018.1

For Support questions about any InterSystems products, contact:

InterSystems Worldwide Response Center (WRC)
Tel: +1-617-621-0700
Tel: +44 (0) 844 854 2917
Email: support@InterSystems.com

Table of Contents

About This Book **1**

CMQL Commands, Keywords, and Query Logging **3**
- CMQL Commands 4
- CMQL Keywords 5
- CMQL Query Logging 6

CMQL Clauses **9**
- AVERAGE clause 10
- AVG clause 11
- BETWEEN clause 13
- BREAK.ON clause 14
- BREAK.SUP clause 16
- BY clause 18
- CALC clause 21
- COL.HDG clause 22
- COL.SPACES clause 23
- CONV clause 24
- CONVERSION clause 25
- DISPLAY.LIKE clause 26
- DISPLAY.NAME clause 27
- ENUM clause 28
- EVAL clause 30
- FIRST clause 32
- FMT clause 33
- FOOTER clause 34
- FOOTING clause 35
- FROM clause 37
- HEADER clause 38
- HEADING clause 39
- IF clause 41
- LPTR clause 42
- MAX clause 43
- MIN clause 45
- PCT clause 47
- PERCENT clause 48
- PREFETCH clause 49
- REQUIRE.SELECT clause 50
- SAMPLE clause 51
- SAMPLED clause 52
- SAMPLING clause 53
- SAVING clause 54
- SELECT.ONLY clause 55
- TOTAL clause 56
- WHEN clause 58
- WHERE clause 60
- WITH clause 61

About This Book

This book provides reference material for CMQL (Caché MultiValue Query Language). CMQL is the Caché MultiValue implementation of an SQL-like query language for MultiValue data.

This book contains the following sections:

- A list of CMQL commands
- A list of CMQL keywords
- How to perform CMQL query logging
- CMQL query clauses

There is also a detailed Table of Contents.

Other related topics in the Caché documentation set are:

- *Using the MultiValue Features of Caché*
- *Operational Differences between MultiValue and Caché*
- *Caché MultiValue Commands Reference*
- *The Caché MultiValue Spooler*

For general information, see *Using InterSystems Documentation*.

CMQL Commands, Keywords, and Query Logging

CMQL Commands

A list of commands that support CMQL clauses.

Description

The following commands support CMQL query clauses:

- BSELECT
- COUNT
- LIST
- LIST.ITEM
- LIST.LABEL
- REFORMAT
- SELECT
- SORT
- SORT.ITEM
- SORT.LABEL
- SREFORMAT
- SSELECT
- STAT
- SUM

The following commands perform CMQL queries on the VOC file. They support the WHEN conditional clause. They do not support the WITH clause (or IF or WHERE conditional clauses):

- LISTF
- LISTPA
- LISTPH
- LISTS

For further details on these commands, refer to the Caché MultiValue Commands Reference.

CMQL Keywords

A list of keywords used in CMQL clauses.

Non-Functional CMQL Keywords

The following CMQL keywords may be included in a query for readability, but are ignored by CMQL and perform no function:

A, AN, ANY, ARE, FILE, FOR, IN, INVISIBLE, ITEMS, OF, PRINT, THAN, THE.

The Attribute 2 value for these keywords is TA.

CMQL Clause Keywords

The following are CMQL clause keywords:

AVERAGE, AVG, BETWEEN, BREAK.ON, BREAK-ON, BREAK.SUP, BY, BY-DSND, BY-EXP, BY-EXP-DSND, BY.DSND, BY.EXP, BY.EXP.DSND, CALC, COL.HDG, COL.SPACES, COL.SPCS, CONV, CONVERSION, DISPLAY.LIKE, DISPLAYLIKE, DISPLAY.NAME, ENUM, EVAL, FIRST, FMT, FOOTER, FOOTING, FROM, HEADER, HEADING, IF, LPTR, MAX, MIN, PCT, PERCENT, PREFETCH, REQUIRE.SELECT, SAMPLE, SAMPLED, SAMPLING, SAVING, SELECT.ONLY, TOTAL, WHEN, WHERE, WITH.

CMQL Subclause Keywords

The following are CMQL subclause keywords:

- Used with the EVAL clause: AS.
- Used with the SAVING clause: NO.NULLS, UNIQUE.
- Used in the TOTAL clause (and other numeric calculation clauses): GRAND-TOTAL, GRAND.TOTAL.
- Used in the WHEN clause: ASD, ASSOCIATED.
- Used in the WITH clause (and its synonyms) and described in the WITH clause:

  Equality operators: #, AFTER, BEFORE, EQ, EQUAL, GE, GREATER, GT, LE, LESS, LT, NE, NOT.

  Other keywords: AND, EACH, EVERY, ID.ONLY, IS.NULL, IS.NOT.NULL, LIKE, MATCHES, MATCHING, NO, NOT.MATCHING, ONLY, OR, SAID, SPOKEN, UNLIKE, WITHOUT.

MultiValue Verb Keywords

The following are MultiValue command keywords: ALL, DATA, DICT, DIR, OFF, ON, TEMPL, TO, USING.

The following are display formatting keywords. They are described with the CMQL commands that support them:

COL-HDR-SUPP, COL.HDR.SUPP, COL.SUP, COL-SUPP, COUNT.SUP, DBL.SPC, DBL-SPC, DET.SUP, DET-SUPP, HDR.SUP, HDR-SUPP, ID-SUP, ID.SUP, ID-SUPP, NI.SUP, NI-SUPP, NO.PAGE, NOPAGE, SUPP, VERT, VERTICALLY.

CMQL Query Logging

Creating a log of CMQL queries.

Description

You can create a log of executed CMQL queries. CMQL logging is intended as a diagnostic aid for use when porting MultiValue queries. It should not be used as part of a production application.

The CMQL query log is created by writing subscript entries to the Caché global ^CMQLlog. This log is activated by setting ^CMQLlog = 1 and deactivated by setting ^CMQLlog = 0. Note that global names are case-sensitive.

The operations that are listed in the CMQL log are invocations of CMQL commands, regardless of whether the command completes successfully.

Activating CMQL Logging from Caché MultiValue

To activate CMQL logging from the Caché MultiValue Shell, use the COS, #, or [commands, which permit you to issue an ObjectScript command from within MultiValue:

```
USER: COS SET ^CMQLlog=1
```

For further details on these commands, refer to the Caché MultiValue Commands Reference.

Setting ^CMQLlog=1 activates the logging of queries for the current account (namespace). All CMQL queries are logged, including queries that fail to execute due to an error. This logging remains in effect for all users of the current account (namespace) until ^CMQLlog is explicitly reset to 0 or the global is KILLed. CMQL logging persists across Caché restart.

Setting ^CMQLlog=0 deactivates the logging of queries for the current account (namespace). This means that queries issued when ^CMQLlog is set to zero are not logged in ^CMQLlog. Setting ^CMQLlog=0 does not delete the existing contents of ^CMQLlog. Queries that have been logged in ^CMQLlog remain listed in the log until you issue a `KILL ^CMQLlog` command. Setting ^CMQLlog=0 also does not suspend CMQL routine numbering. Caché MultiValue assigns a sequential integer routine number to every CMQL query issued, whether or not that CMQL query is logged. Issuing a `KILL ^CMQLlog` does not reset this CMQL routine number counter.

Viewing the CMQL Log

You can use the ObjectScript ZWRITE command to view the CMQL query log. The following example displays ^CMQLlog from within the MultiValue Shell:

```
USER: COS ZWRITE ^CMQLlog
^CMQLlog=1
^CMQLlog(66,1)="LIST VOC WITH @ID LIKE Q..."
^CMQLlog(66,2)=""
^CMQLlog(66,3)="405ý3ýýþ"
^CMQLlog(66,4)=3432
^CMQLlog(67,1)="SELECT VOC TO 4"
^CMQLlog(67,2)=""
^CMQLlog(67,3)="404ý480ý4ýþ"
^CMQLlog(67,4)=5996
```

The first line returned shows the activation status of ^CMQLlog. In this case, CMQL logging is active for the USER account.

Each query in the CMQL log is represented by four lines. The first ^CMQLlog subscript is the CMQL routine number counter (in this case, routines 66 and 67). The second ^CMQLlog subscript is the log line number (1 through 4) for that query.

- Log line 1: Contains the text of the query as a quoted string. Note that double quotation marks within the query string are duplicated (""item"") to indicate a literal quotation mark rather than the end of the query string.

- Log line 2: If executing the query directly, contains an empty string. If executing the query using the MVBasic EXECUTE command, contains the execute stack entry, a string such as `$lb("+1^MVBASIC6108.mvi +2")`, where 6108 is the pid number.
- Log line 3: Contains execution information about the query, specified as a dynamic array. Items are separated by value marks (@VM):
  - The first item is the error or completion message number. On error, returns the error code number. On success, returns one of the following: 401=LISTPA, LISTPH, LISTS successful, 403=SORT.ITEM successful, 404=SELECT, SSELECT, BSELECT successful, 405=LIST, LISTF, LIST.ITEM, SORT successful; 407=COUNT successful; 438=SUM successful; empty string=LIST.LABEL, SORT.LABEL, STAT successful.
  - The second item is the number of records selected by the query upon success. If an error occurred, the second item may contain error information.
  - The third item is the select list selected to.

  If a query returns multiple messages, the information for each message is presented in the log line 3 dynamic array, with messages separated by field marks (@FM), with the success message items listed first. One example of this is a LIST.ITEM command that specifies some records in the file and some records that are not in the file. Each item not in the file returns its own field. This is shown in the following example:

```
^CMQLlog(9,1)="LIST.ITEM VOC 'A' 'FRED' 'BASIC'"
^CMQLlog(9,2)=""
^CMQLlog(9,3)="405ý2ýýþ780ýFREDýýþ"
^CMQLlog(9,4)=6108
```

- Log line 4: Contains the process ID (pid) of the process executing the query.

CMQL Clauses

AVERAGE clause

Synonym for AVG.

Description

The **AVERAGE** clause is a synonym for the AVG clause.

AVG clause

Returns the average value of a field.

```
AVG field [NO.NULLS] [GRAND.TOTAL "label"]
AVERAGE field [NO.NULLS] [GRAND.TOTAL "label"]
```

Description

The **AVG** and **AVERAGE** keywords are synonyms.

The **AVG** clause calculates the average value for a numeric field. If a WITH clause (or some other conditional clause) is specified, **AVG** returns the average of the values of that field that pass the condition test.

By default, fields containing nonnumeric values or NULL are counted in determining the average value. NULL and nonnumeric fields are treated as having the numeric value of 0. The optional NO.NULLS keyword removes items without a value (NULL) from the count of items used to calculate the average.

AVG calculates an average to nine decimal places. Leading and trailing zeros are suppressed.

Multiple CMQL clauses may be specified in any order. The order of application of CMQL clauses is always the same. The **WITH** clause (or other condition test clause) is applied first. The **SAMPLED** clause (if present) is applied next, then the **SAMPLE** clause (if present), then the **AVG** clause.

GRAND.TOTAL Keyword

GRAND.TOTAL and GRAND-TOTAL are synonyms.

By default, in a horizontal display the summation line is not labeled; it is indicated by the "***" placeholder. You can use the GRAND.TOTAL keyword to assign a label name to the summation line. GRAND.TOTAL has no effect on vertical display format.

The following example shows the default labeling of an average:

```
LIST SALES WITH AMOUNT > "$100.00" AVG AMOUNT (D
```

It returns:

```
SALES..... AMOUNT.........

***             $504.34

24423 Items listed.
```

The following example uses GRAND.TOTAL to label the average:

```
LIST SALES WITH AMOUNT > "$100.00" AVG AMOUNT GRAND.TOTAL "AvgAmt" (D
```

It returns:

```
SALES..... AMOUNT.........

AvgAmt          $504.34

24423 Items listed.
```

Note that the total, maximum, minimum, and average values are all listed as columns in the same summation line. They are listed in the order in which the clauses were specified. A single GRAND.TOTAL keyword specifies the label for the entire summation line. GRAND.TOTAL does not have to directly follow the clause(s) that it affects. If you specify multiple GRAND.TOTAL keywords, the last one specified is used.

Examples

The following example returns both the total value and the average value of the field F5; in this case the VOC contains 478 records, most of which have no value for the F5 field:

```
SUM VOC F5 AVG F5
```

It returns 52 for the sum of F5, and .108786611 (52 / 478) for the average value of F5.

The following example returns both the total values and the average value of the field F5, when NULLs are not counted. As can be seen from the **COUNT** command, the VOC contains 48 records in which F5 has a value that includes at least one character.

```
COUNT VOC F5 WITH F5 LIKE "...0X"
SUM VOC F5 AVG F5 NO.NULLS
```

The **SUM** command returns 52 for the sum of F5, and 1.083333333 (52 / 48) for the average value of F5.

See Also

- MAX clause
- MIN clause
- TOTAL clause

BETWEEN clause

Returns items with a range of values.

```
[[WITH] field] BETWEEN "start" "end"
WHEN field BETWEEN "start" "end"
```

Description

The **BETWEEN** clause returns those values of *field* between *start* and *end*, inclusive of both *start* and *end*. This range is specified as an ascending collation sequence. Neither *start* nor *end* have to be existing *field* values. Comparisons are case-sensitive.

If *start* and *end* are the same value, that value (if present) is returned. If *start* is higher in the collation sequence than *end*, no values are returned and a [401] “No items present” message is displayed.

BETWEEN cannot use "" (null) as a *start* or *end* value. If *start* or *end* is "" (null) no values are returned and a [401] “No items present” message is displayed. This handling of null differs from other conditional operations such as **AFTER**, >, and >= conditionals, which use "" (null) to indicate the beginning of the collation sequence, or = "" , which returns those fields that have no value (null).

BETWEEN can be used in a WITH clause or a WHEN clause. If you omit the WITH keyword, the **BETWEEN** test defaults to an implicit WITH @ID clause, as shown in the following examples.

The following example tests for a range of F4 values, returning the @ID of the selected records.

```
LIST VOC WITH F4 BETWEEN "A" "M"
```

The following example tests for a range of @ID values, returning the @ID and F4 values of the selected records:

```
LIST VOC F4 BETWEEN "A" "M"
```

Examples

The following example lists the values between “A” and “AM” (inclusive):

```
LIST VOC WITH @ID BETWEEN "A" "AM"
```

It returns A ABORT AFTER ALL ALL.MATCH.

The following example uses “ ” (blank) to approximate the beginning of the collation sequence:

```
LIST VOC WITH @ID BETWEEN " " "AM"
```

It returns # &COMO& &PH& &SAVEDLISTS& ; @CMQLOPTS A ABORT AFTER ALL ALL.MATCH.

BREAK.ON clause

Indicates changes of values with indicator string.

```
BREAK.ON field ["text'code'"]
```

Description

BREAK.ON and **BREAK-ON** are synonyms.

The **BREAK.ON** clause displays the values of *field*, and inserts a break indicator in the display at each point where the *field* value changes. By default, this break indicator consists of a new line with *** indicating a change in *field* values. You can supply a string as the break indicator.

You can specify multiple **BREAK.ON** clauses to insert breaks for multiple levels of sorting.

The **BREAK.ON** clause differs from the BREAK.SUP clause in two ways:

- **BREAK.ON** always displays the break *field* values. **BREAK.SUP** does not display the break *field* values unless you specify this *field* as a display column.
- **BREAK.ON** indicates a *field* value change either with *** (the default) or a user-specified break string. **BREAK.SUP** indicates a *field* value change with a blank line.

Multiple CMQL clauses may be specified in any order. The order of application of CMQL clauses is always the same. The **BREAK.ON** clause is applied after the **WITH** clause, the **SAMPLED** clause (if present), and the **SAMPLE** clause (if present).

The following example displays the a break indicator each time the F1 field value changes:

```
LIST VOC BREAK.ON F1 WHERE FILENAME >= "A"
```

It lists output such as the following:

```
VOC..... F1.....

A        K

         ***

ABORT     V

         ***

AFTER      K
ALL        K
ALL.MATCH  K
AN         K
AND        K
ANY        K
ARE        K
AS         K
ASC        K
ASD        K

         ***

ASSIGN    V
```

The optional `"text'code'"` parameter allows you to specify how **BREAK.ON** values are displayed. Like the HEADING codes, the outer quotes specify a literal and the inner nested quotes specify letter code characters. Both are optional.

- The `"text"` value, if specified, is substituted for the *** default value change marker.
- The `'B'` letter code supplies the current field value to the HEADING clause.

- The 'D' letter code suppresses the value change marker when a break occurs after a unique field value. The value change marker is displayed when there are more than one fields with the same value. If you also specify DET-SUPP, the 'D' letter code is overridden and performs no operation.
- The 'L' letter code suppresses a blank line before the value change marker.
- The 'P' letter code specifies that each change in the field value should force a page break. 'N' is a synonym for 'P'.
- The 'S' letter code suppresses the value change marker. Changes in value are separated by a blank line.
- The 'V' letter code supplies the current field value as the value change marker string.

The following example shows the use of the "text'code'" parameters:

```
LIST VOC BY F1 BREAK.ON F1 "New Value'PB'" HEADING "F1='B'"
```

In this case, the VOC is sorted by F1 values. The literal New Value replaces *** as the F1 field value change indicator. Each time the F1 field value changes, the 'P' letter code issues a page break. The 'B' letter code supplies the current F1 field value to the page header.

You can use a FMT clause to format the break on field display column. The integer specifies the width of the display column. The letters L or R specify left justification or right justification. If neither is specified, the default is left justification.

The following example shows the use of the FMT clause:

```
LIST VOC WITH F4 BY F4 BREAK.ON F4 "Next Value'P'" FMT "5R"
```

In this case, those records with F4 field values are sorted and displayed, with a break each time the F4 value changes. The FMT clause specifies a width of 5 characters, right justified. This formatting affects both the display of the F4 values and the display of the “Next Value” change marker.

BREAK.ON can be used with AVG, ENUM, PERCENT, or TOTAL to return a calculated value for each distinct field value.

The following example uses **BREAK.ON** with the **ENUM** clause to return counts of the number of occurrences of each distinct non-null value for F4, and the final count of all F4 values. The initial F4 count of 0 records indicated that there are one or more records which have no value (null) for F4:

```
SORT VOC BY F4 BREAK.ON F4 ENUM F4 DET-SUPP
```

The following example is almost identical to the previous one. The **WITH** clause eliminates F4 count of 0 records displayed by the previous example:

```
SORT VOC BY F4 WITH F4 BREAK.ON F4 ENUM F4 DET-SUPP
```

The following example uses two **BREAK.ON** clauses. The first inserts a break at each change in the F1 value, the second inserts a break at each change in the F4 value within the same F1 values:

```
LIST VOC BY F1 BY F4 BREAK.ON F1 BREAK.ON F4
```

Emulation

UniData emulation **BREAK.ON** display differs in several details. At a New Value break, UniData provides as a break indicator an asterisk underline, then displays the breaking value underneath. When providing a total, UniData displays a “======” underline, then displays the total value underneath. UniData labels the total value with the word “TOTAL” by default.

BREAK.SUP clause

Indicates changes of values with line breaks.

```
BREAK.SUP field ["'code'"]
```

Description

The **BREAK.SUP** clause inserts a blank line break indicator in the display at each point where the *field* value changes. Optionally, you can specify a *code* of `"'B'"` to provide the current *field* value to the header, or a *code* of `"'P'"` to issue a page break at each point where the *field* value changes. `"'N'"` is a synonym for `"'P'"`.

You can specify multiple **BREAK.SUP** clauses to insert breaks for multiple levels of sorting.

If the DET.SUP keyword is specified, CMQL detects breaks where the break *field* has **BREAK.SUP**, but only outputs a break line for it if it is the lowest level break. If both the DBL.SPC and DET.SUP keywords are specified, the DBL.SPC overrides the DET.SUP break line suppression.

The **BREAK.SUP** clause differs from the BREAK.ON clause in two ways:

- **BREAK.SUP** does not display the break *field* values unless you specify this *field* as a display column. **BREAK.ON** always displays the break *field* values.
- **BREAK.SUP** indicates a *field* value change with a blank line. **BREAK.ON** indicates a *field* value change either with *** (the default) or a user-specified break string.

Multiple CMQL clauses may be specified in any order. The order of application of CMQL clauses is always the same. The **BREAK.SUP** clause is applied after the **WITH** clause, the **SAMPLED** clause (if present), and the **SAMPLE** clause (if present).

The following example displays the a break indicator each time the F1 field value changes:

```
LIST VOC BREAK.SUP F1 WHERE FILENAME >= "A"
```

It lists output such as the following, where blank line breaks indicate changes in the F1 value:

```
VOC..... .....

A


ABORT


AFTER
ALL
ALL.MATCH
AN
AND
ANY
ARE
AS
ASC
ASD


ASSIGN
```

If you wish to display the actual *field* values, you must specify the field, as shown in the following:

```
LIST VOC F1 BREAK.SUP F1 WHERE FILENAME >= "A"
```

The following example shows the use of the *code* parameter to supply the current F1 value to each page header and issue a page break:

```
LIST VOC BY F1 BREAK.SUP F1 "'BP'" HEADING "Type='B'"
```

In this case, the VOC is sorted by F1 values. Each time the F1 field value changes, the `'P'` letter code issues a page break. The `'B'` letter code supplies the current F1 field value to the page header. Note that the F1 values are not displayed; however, the F1 value for the current page is shown in the page header.

BY clause

Sorts query output using a field value.

```
BY field
BY.DSND field
BY.EXP field
BY.EXP.DSND field
```

Description

The **BY** clause sorts the output returned by the query in ascending order by the values of the specified *field*. You can specify multiple **BY** clauses for nested sorting operations. Sorts are performed in the order that the **BY** clauses are specified. For example, `BY F1 BY F3` sorts first by F1 values, then by F3 values within identical F1 values. NULL values always sort to the beginning of the sort order.

BY sorts in ascending collation sequence order. **BY.DSND** sorts in descending collation sequence order. **BY.DSND** and **BY-DSND** are synonyms.

BY sorts a single-value field. Use **BY.EXP** when sorting a multivalue dynamic array field. **BY.EXP** explodes each dynamic array into its own data row for the purposes of this sort operation. **BY.EXP** sorts in ascending collation sequence order; **BY.EXP.DSND** sorts in descending collation sequence order. **BY.EXP** and **BY-EXP** are synonyms. **BY.EXP.DSND** and **BY-EXP-DSND** are synonyms.

When multiple CMQL clauses are specified the clauses may be specified in any order. The order of application of CMQL clauses is as follows: the **WITH** clause is always applied first. Its results are supplied to the **BY** clause(s) (if present), which supplies its results to the **SAMPLED** clause (if present), and its results are supplied to the **SAMPLE** clause (if present).

Single-valued and Multivalued Fields

A *field* is defined in its SM dictionary entry as either “S” (single-valued) or “M” (multivalued). It is also possible that a *field* has no SM dictionary entry.

BY assumes that its *field* dictionary entry is a single-valued element. If *field* is a multivalued element, the **BY** clause is logically incorrect. In this case, CMQL treats *field* as single valued for the purpose of sorting; all the multivalues will be sorted as if they were a single string. This can sometimes be used as a 'trick' to sort by the first multivalue only. However, if you are doing this it is suggested that you review your file layout.

BY.EXP assumes that its *field* dictionary entry is a multivalued element. If *field* is a single valued field, the **BY.EXP** clause is logically incorrect. In this case, CMQL treats *field* as if it were a multivalued element.

Specifying the wrong **BY** or **BY.EXP** for the *field* may result in CMQL being unable to optimize the query using indexes.

When performing a sort on a child table, when a row is specified as single-valued, Caché MultiValue generates a reference to the parent table for that column. This means that a single value (the first value) is returned for every row. When displaying single values in an exploding list, single valued attributes should be designated with an “S” in the SM dictionary attribute (field 5):

- A and S type dictionaries: all single valued attributes must be designated with an “S”. Otherwise a **BY.EXP** sort will not recognize these attributes.
- D and I type dictionaries: all single valued attributes should be designated with an “S”. If not designated as either “S” or ”M”, a **BY.EXP** sort assumes an “S” dictionary attribute value.

Examples

The following example searches the FILENAME field of the VOC for all values that begin with the letter J, and sorts them by the values of the F1 field:

```
LIST VOC WITH FILENAME LIKE J... BY F1
```

It lists items in the following sequence: JOIN JOBS JED. (JOIN F1=K, JOBS F1=PA, JED F1=V).

The following examples sort the VOC by F1 and F3 values. The first example sorts by F1 and within that by F3 field values. The second example sorts by F3 and within that by F1 field values:

```
LIST VOC WITH F1 > K BY F1 BY F3
```

It lists items in the following sequence (partial list): @CMQLOPTS JOBS LIST.JOB LISTME LISTU COL-HDR-SUPP COL.HDR.SUPP LISTF LISTPA LISTPH LISTS ; ABORT ASSIGN AUTOLOGOUT BASIC ...

```
LIST VOC WITH F1 > K BY F3 BY F1
```

It lists items in the following sequence (partial list): @CMQLOPTS JOBS LIST.JOB LISTME LISTU COL-HDR-SUPP COL.HDR.SUPP LISTF LISTPA LISTPH LISTS RELLEVEL ; ABORT ASSIGN AUTOLOGOUT BASIC ... Notice that RELLEVEL has sorted differently than in the first example.

Multivalued Examples

The following example uses **BY.EXP** to sort the multivalued field AMOUNT in the SALES file:

```
LIST SALES AMOUNT BY.EXP AMOUNT WITH @ID < 5
```

It returns the following data. Note that each @ID item is exploded into as many rows as there are AMOUNT values for that item. The number of items listed is the exploded number of rows (8), not the number of records (4).

```
SALES..... AMOUNT.....

1          $170.03
4          $196.13
2          $361.95
1          $707.59
4          $788.58
3          $807.70
1          $848.47
3          $968.34

8 Items listed.
```

The following example uses **BY** to sort the multivalued field AMOUNT in the SALES file:

```
LIST SALES AMOUNT BY AMOUNT WITH @ID < 5
```

The **BY** sort treats the multivalue string as a single value. This results in a sort by the first element in the multivalue; it returns the following data. Because the multivalue data is not exploded, the number of items listed is the number of records (4):

```
SALES..... AMOUNT.....

4          $196.13
           $788.58
2          $361.95
3          $807.70
           $968.34
1          $848.47
           $170.03
           $707.59

4 Items listed.
```

The following example first uses **BY** to sort the single-value @ID field, then uses **BY.EXP** to sort the multivalue AMOUNT field:

```
LIST SALES AMOUNT BY @ID BY.EXP AMOUNT WITH @ID < 5
```

It returns the following data:

```
SALES..... AMOUNT.....

1          $170.03
1          $707.59
1          $848.47
2          $361.95
3          $807.70
3          $968.34
4          $196.13
4          $788.58

8 Items listed.
```

Emulation

When exploding, UniVerse and UniData explode subvalues, but D3, Reality, and jBASE only explode values. Reality provides a BY.EXP.SUB keyword to explode by subvalues which gives the same result as a simple BY.EXP on UniVerse.

CALC clause

Returns the calculated value for a field.

```
CALC field
```

Description

The **CALC** clause calculates the value for an I-type numeric field. **CALC** calculates to two decimal places. Leading and trailing zeros are suppressed.

Multiple CMQL clauses may be specified in any order. The order of application of CMQL clauses is always the same. The **CALC** clause is applied after the **WITH** clause, the **SAMPLED** clause (if present), and the **SAMPLE** clause (if present).

Examples

The following MultiValue Basic example creates an I-type field IPCT the value of which is calculated from the values of the fields PRICE and DSCPRICE:

```
0001 EXECUTE 'CREATE-FILE MYTESTFILE 1 1' CAPTURING STUFF
0005 OPEN 'DICT','MYTESTFILE' TO DICT.FP ELSE ABORT
0006 OPEN 'MYTESTFILE' TO FP ELSE ABORT
0007 WRITE '100':@AM:'50' ON FP,3
0008 WRITE '197':@AM:'175' ON FP,2
0009 WRITE '203':@AM:'180' ON FP,1
0010
0011 WRITE 'D':@AM:1:@AM:@AM:'PRICE':@AM:'10R' ON DICT.FP,'PRICE'
0012 WRITE 'D':@AM:2:@AM:@AM:'DSCPRICE':@AM:'10R' ON DICT.FP,'DSCPRICE'
0013 WRITE 'I':@AM:\(TOTAL(PRICE-DSCPRICE))*100/TOTAL(PRICE)\:@AM:'MR20':@AM:'IPCT':@AM:'10R' ON
DICT.FP,'IPCT'


LIST MYTESTFILE TOTAL PRICE TOTAL DSCPRICE CALC IPCT
```

COL.HDG clause

Synonym for DISPLAY.NAME.

Description

The **COL.HDG** clause is a synonym for the DISPLAY.NAME clause.

COL.SPACES clause

Changes the column display spacing.

```
COL.SPACES n
```

Description

The **COL.SPACES** and **COL.SPCS** keywords are synonyms.

The **COL.SPACES** clause enlarges (or shrinks) the spacing between display columns by *n* spaces. **COL.SPACES** adjusts the width of all display columns equally. The *n* argument can be a positive or negative integer.

The default column widths are specified in the dictionary file listing FORMAT column, as displayed, for example, using `LIST DICT VOC`.

COL.SPACES only affects column spacing in horizontal orientation. It has no effect on data displayed in vertical orientation. Vertical orientation can either be caused by specifying too many items for horizontal display, or by specifying the VERT keyword. When displaying multiple items, you can use **COL.SPACES** to shift display orientation from vertical to horizontal (negative *n*) or horizontal to vertical (positive *n*).

Note that decreasing the spacing between columns may result in displayed data appearing truncated if the column width is narrower than the number of characters in the data.

COL.SPACES uniformly changes the width of all of the display columns. Use FMT to change the width of a single display column.

Examples

The following example increases the spacing between display columns by 5 spaces:

```
LIST VOC F1 F2 F3 COL.SPACES 5
```

The following example decreases the spacing between display columns by 2 spaces:

```
LIST VOC F1 F2 F3 F4 F5 COL.SPACES -2
```

In this example, this is enough to shift display orientation from vertical to horizontal.

CONV clause

Converts the value for a field using conversion codes.

```
field CONV "code"
```

Description

The **CONV** and **CONVERSION** keywords are synonyms.

The **CONV** clause converts the value(s) for the preceding field based on the value of *code*. This *code* is character code, specified as a quoted string, that specifies the type of conversion to perform. Conversion is from internal format to external format.

The conversion *code* values are the same as those used in the MVBasic OCONV function, where they are described in detail. For a complete list of conversion codes, refer to the Conversion Codes table in the *MultiValue Basic Quick Reference*.

Examples

The following example returns two columns: item names as stored in the VOC, and these item names converted to lowercase letters:

```
LIST VOC @ID CONV "MCL"
```

The following MultiValue Basic example creates an I-type field IPCT the value of which is calculated from the values of the fields PRICE and DSCPRICE:

```
0001 EXECUTE 'CREATE-FILE MYTESTFILE 1 1' CAPTURING STUFF
0005 OPEN 'DICT','MYTESTFILE' TO DICT.FP ELSE ABORT
0006 OPEN 'MYTESTFILE' TO FP ELSE ABORT
0007 WRITE '100':@AM:'50' ON FP,3
0008 WRITE '197':@AM:'175' ON FP,2
0009 WRITE '203':@AM:'180' ON FP,1
0010
0011 WRITE 'D':@AM:1:@AM:@AM:'PRICE':@AM:'10R' ON DICT.FP,'PRICE'
0012 WRITE 'D':@AM:2:@AM:@AM:'DSCPRICE':@AM:'10R' ON DICT.FP,'DSCPRICE'
0013 WRITE 'I':@AM:\(TOTAL(PRICE-DSCPRICE))*100/TOTAL(PRICE)\:@AM:'MR20':@AM:'IPCT':@AM:'10R' ON
DICT.FP,'IPCT'
```

Using the above data, the following example converts the display of PRICE and DSCPRICE to dollar values with two fractional digits:

```
LIST MYTESTFILE TOTAL PRICE CONV "MR2$" TOTAL DSCPRICE CONV "MR2$" CALC IPCT
```

CONVERSION clause

Synonym for CONV.

Description

The **CONVERSION** clause is a synonym for the CONV clause.

DISPLAY.LIKE clause

Displays DICT entry as column heading.

```
dict DISPLAY.LIKE dict
```

Description

The **DISPLAY.LIKE** and **DISPLAYLIKE** keywords are synonyms.

The **DISPLAY.LIKE** clause replaces the default column heading with a synonym defined as the corresponding DICT element in the DICT file. The specified DICT elements are case-sensitive, and are not enclosed in quote characters.

DISPLAY.LIKE displays a defined DICT element as the column heading. **DISPLAY.NAME** displays a user-specified name as the column heading.

Examples

The following example first shows how the F1 field is returned without **DISPLAY.LIKE** and then with a **DISPLAY.LIKE** clause:

```
LIST VOC F1 WITH @ID LIKE Q...
LIST VOC F1 DISPLAY.LIKE TYPE WITH @ID LIKE Q...
```

```
VOC..... F1.....

Q       V
QSELECT    V
QUIT       V

3 Items listed.

VOC..... Typ

Q       V
QSELECT    V
QUIT       V

3 Items listed.
```

DISPLAY.NAME clause

Displays specified string as column heading.

```
dict DISPLAY.NAME name
```

Description

DISPLAY.NAME and **COL.HDG** are synonyms.

The **DISPLAY.NAME** clause replaces the default column heading with user-supplied *name*. This name may be supplied with quote delimiters (`"my text"`), which permits the column heading to contain blank spaces or other special characters. To include quote characters in a column heading, use the backslash character as the string delimiter (`\Survey "yes" results\`). If no special characters are included, *name* delimiters are not required. Column headings are case-sensitive.

To specify a blank column heading, use `"\"` as the *name* value. This removes the default column heading without supplying a replacement value.

DISPLAY.NAME displays a user-specified name as the column heading. DISPLAY.LIKE displays a defined DICT element as the column heading.

The EVAL clause can specify a column heading name using the AS keyword. If you wish to display an **EVAL** column with no column heading, you can use `DISPLAY.NAME "\"`, instead of the AS keyword.

Examples

The following example first shows how the F1 field is returned without **DISPLAY.NAME** and then with a **DISPLAY.NAME** clause:

```
LIST VOC F1 WITH @ID LIKE Q...
LIST VOC F1 DISPLAY.NAME "Type Code" WITH @ID LIKE Q...
```

returns:

```
VOC..... F1.....

Q       V
QSELECT   V
QUIT      V

3 Items listed.

VOC..... Type Code.....

Q       V
QSELECT   V
QUIT      V

3 Items listed.
```

The following example shows **DISPLAY.NAME** used with the **TOTAL** clause to assign a name to a returned total (the (D letter code suppresses display of all details except the total):

```
LIST VOC TOTAL F5 DISPLAY.NAME "Total F5" (D
```

returns:

```
VOC..... Total F5.....

***       52

478 Items listed.
```

ENUM clause

Returns the number of occurrences of a field.

```
ENUM field [NO.NULLS]
```

Description

The **ENUM** clause counts the number of occurrences of a field. It counts every occurrence of the specified field, including nulls (items that do not contain a value). You can use the **NO.NULLS** keyword to limit the **ENUM** count to non-null values.

The following example uses **ENUM** to count all of the occurrences of the F5 field, and **TOTAL** to return the total of these values:

```
LIST VOC ENUM F5 TOTAL F5 DET-SUPP
```

returns:

```
VOC..... F5..... F5.....

***              478 52

478 Items listed.
```

In this example, the DET-SUPP keyword suppresses the listing of individual items.

The following example uses the **NO.NULLS** keyword to limit the **ENUM** count to all non-null occurrences of the F5 field:

```
LIST VOC ENUM F5 NO.NULLS TOTAL F5 DET-SUPP
```

returns:

```
VOC..... F5..... F5.....

***              48 52

478 Items listed.
```

To just return the count of occurrences of a field, you can use the **COUNT** command. To count all occurrences:

```
COUNT VOC F5
```

To count all non-null occurrences:

```
COUNT VOC F5 WITH F5
```

ENUM with BREAK.ON

You can use the BREAK.ON clause with **ENUM** to return subcounts for each distinct value, as well as an overall count.

The following example counts the number of occurrences of each distinct value for F4, and the final count of all F4 values. These counts include a count of F4 fields with no value (null):

```
LIST VOC BY F4 BREAK.ON F4 ENUM F4 DET-SUPP
```

The following example counts the number of occurrences of each distinct non-null value for F4, and the final count of all non-null F4 values:

```
LIST VOC BY F4 BREAK.ON F4 ENUM F4 NO.NULLS DET-SUPP
```

Note that this example displays a line for the F4 nulls, with an **ENUM** subcount of 0. This indicates that there are one or more nulls, but they are not being added to the final count.

The following example also counts the number of occurrences of each distinct non-null value for F4, and the final count of all non-null F4 values:

```
LIST VOC WITH F4 BY F4 BREAK.ON F4 ENUM F4 DET-SUPP
```

The `WITH F4` conditional clause is applied before **ENUM**; it restricts the result set to non-null values. Therefore **ENUM** does not indicate the presence (or absence) of nulls.

EVAL clause

Returns an evaluated expression value as an item.

```
EVAL "expression" [AS colname]
```

Description

The **EVAL** clause evaluates *expression* and returns the result as a separate item. An *expression* can be an expression involving numeric and string literals, variables, MVBasic functions, or field values. An *expression* must be enclosed in double quotes. If *expression* contains the name of a column, that column name is truncated to the first 50 characters.

An **EVAL** *expression* that contains a field returns the contents of the field's itype attribute. Note that in Caché MultiValue if the value of *field* is a number, it is treated as a literal number and not an attribute name. For example, the itype expression `"@DATE-1"` evaluates 1 as a numeric literal, and thus returns yesterday's date.

You can specify a string literal as *expression* by enclosing it in single quotes, for example `EVAL "'OK'"`. You can concatenate a string literal to an *expression*, for example `EVAL "@DATE-1:' Yesterday'"`.

This evaluation of quoted numerics is consistent with most MultiValue implementation, with the exception of jBASE. jBASE evaluates a quoted number as an itype attribute name, and attempts to return the attribute contents.

Postfix Operators

An **EVAL** *expression* can contain postfix operators, such as the substring extraction operator [n,m]. The following example extracts a substring consisting of the first 3 characters of the specified @ID values:

```
LIST VOC "BASIC" "ASSOC" EVAL "@ID[1,3]"
```

It returns BAS and ASS.

You can specify multiple postfix operators. They are evaluated in the order specified. The following example uses two postfix operators. The first ([1,3]) extracts a substring consisting of the first 3 characters of the specified @ID values; the second ([2,99]) extracts a substring from that substring consisting of the second through 99th characters:

```
LIST VOC "BASIC" "ASSOC" EVAL "@ID[1,3][2,99]"
```

It returns AS and SS.

The number of sequential postfix operators is unlimited.

AS clause

The optional AS clause lets you assign a column name to the resulting column. This *colname* may be supplied with quote delimiters (`"my text"`), which permits the column heading to contain blank spaces or other special characters. If no special characters are included, *colname* delimiters are not required. Column headings are case-sensitive.

If you do not specify an AS clause, the *expression* is used as the column header. The AS clause cannot specify a blank column heading. To display an **EVAL** column with no column heading, use `DISPLAY.NAME "\"`, instead of the AS keyword. To display a column heading that contains literal quote characters, use a DISPLAY.NAME clause such as the following (`DISPLAY.NAME \survey "yes" answers\`), instead of the AS keyword.

Examples

The following example evaluates an arithmetic operation on a variable value. **EVAL** returns a date one week from the current date. It uses the AS clause to assign a name to this calculated date column:

```
LIST VOC WITH @ID LIKE Q... EVAL "@DATE+7" AS "Next Week"
```

returns:

```
VOC..... Next Week.

Q         15513
QSELECT     15513
QUIT        15513

3 Items listed.
```

The following example uses **EVAL** to specify a dictionary entry by position (rather than name) using the @RECORD variable. The following two statements are equivalent:

```
LIST VOC EVAL "@RECORD<4>" AS "Field4" WITH EVAL "@RECORD<4>"
LIST VOC F4 DISPLAY.NAME "Field4" WITH F4
```

Note the use of the AS keyword and the DISPLAY.NAME clause to assign a column heading to the specified field for each statement.

The following example uses **EVAL** to return the length of each field's @ID name, using the MVBasic SUBR function to call the LENS (length) system function:

```
LIST VOC EVAL "SUBR('-LENS',@ID)" AS "NameLength"
```

FIRST clause

Synonym for SAMPLE.

Description

The **FIRST** clause is a synonym for the SAMPLE clause.

FMT clause

Sets column width and justification.

```
field FMT "n[L | R | U]"
```

Description

The **FMT** clause allows you to format the column width and justification for an individual field display column. The integer *n* specifies the width of the display column. The letters L or R specify left justification or right justification. These letter codes are not case-sensitive. If neither letter is specified, the default is left justification.

If the field data is wider than the display column width, the display behavior depends on the justification setting and the current emulation:

- L (left justification) causes data wider than the display column width to wrap in Caché MultiValue and all emulations.
- R (right justification) causes data wider than the display column width to wrap in Caché MultiValue and in the following emulations: INFORMATION, PIOpen, Prime, UniData, and UniVerse. In all other emulations the data column automatically expands leftward, potentially overwriting the data in the column(s) to its left.

The U letter code suppresses column width formatting, allowing the column to expand rightward without line wrapping. The *n* value can be omitted, or it can be specified and is ignored.

FMT changes the width of a single display column. Use COL.SPACES to uniformly change the width of all of the display columns.

Examples

The following example shows the use of the **FMT** clause:

```
LIST VOC WITH F4 F4 FMT "5R"
```

In this case, the F4 column is displayed with a width of 5 characters, right justified.

The following example shows the use of the **FMT** clause with the BREAK-ON clause:

```
LIST VOC WITH F4 BY F4 BREAK-ON F4 "Next Value'P'" FMT "5R"
```

In this case, those records with F4 field values are sorted and displayed, with a break each time the F4 value changes. The **FMT** clause specifies a width of 5 characters, right justified. This formatting affects both the display of the F4 values and the display of the “Next Value” change marker.

FOOTER clause

Synonym for FOOTING.

Description

The **FOOTER** clause is a synonym for the FOOTING clause.

FOOTING clause

Inserts a footer on each page of displayed output.

```
FOOTING "text['code']"
```

Description

The **FOOTING** and **FOOTER** keywords are synonyms.

The **FOOTING** clause causes the specified footer to be displayed (or printed) at the bottom of each page of CMQL output, with the exception of the final page. This footer can consist of any combination of literal *text* and *code* characters. Note that footer literal text is in outer quotes and code characters are in nested quotes.

The available outer quotes are `"text"` or `\text\`. Inner (nested) quotes must be single quotes. Therefore, to display a text and a page number footer you must specify a footer such as `"This is page 'P' of my report"` or `\This is page 'P' of my report\`.

Footer Code Characters

The following footer *code* characters can be included in a specified footer:

'B'	The BREAK-ON field value. Refer to the BREAK-ON clause for details.
'C'	Center the footer. The default is left justified.
'D'	Date. Inserts the current system date in DD MMM YYYY format. For example, `25 May 2010`.
'G'	Right justify the following *text*. The default is left justified.
'L'	Line break inserted in the footer at this point.
'N'	No page break prompts. Displays the header at the top of each page, displays all of the pages without waiting for user prompting for the next page.
'P'	Page numbers incrementing from 1 with leading indent. For example, `"This is page'P' of my report"`.
'Q'	Treat \,], and ^ as literals, not code characters for rest of header.
'S'	Page numbers incrementing from 1 with no indent. For example, `"This is page 'S' of my report"`.
'T'	Time and date. Inserts the current system timestamp in 12–hour format: hh:mm:ss DD MMM YYYY. For example, `9:35:22am 25 May 2010`. Note that the time increments for each page.
^	Page number. Same as 'P'

\	Time and date. Inserts the current system timestamp in 12–hour format: hh:mm:ss DD MMM YYYY format. For example, `"Report printed \ for your review"`. Same as ‘T’.
]	Line break. Same as ‘L’
\“\	A literal double quote character. For example, `\Report is "mostly" accurate\`
\‘\ or “’“	A literal single quote character, specified as two single quote characters. For example, `"Don''t review"`

You can specify more than one letter code within nested quotes. For example, `"Report printed 'CTLC' For your review"` displays a two-line footer with both lines centered, and the with first line containing the current system timestamp.

For further details on using footer codes, refer to the MVBasic FOOTING command, which uses the same codes.

FROM clause

Uses the contents of a select list as query input.

```
FROM slist
```

Description

The **FROM** clause uses the items in the specified active select list as the input set for the query. You can further limit the query results using the WITH clause and other query clauses. You can use the **FROM** clause with any of the various LIST or SELECT commands; these are listed in the CMQL Commands list.

FROM specifies a numbered select list. You must explicitly specify **FROM 0** to specify the default select list (Select List 0). However, if a numbered select list specified in the **FROM** clause is not active, **FROM** uses the default select list (Select List 0). If neither the numbered select list specified in **FROM** nor the default select list is active, the **FROM** clause is ignored.

Valid *slist* values are 0 through 10 (inclusive). Specifying a *slist* value outside of this range generates a [819] error.

Note that an active select list can only be used once.

FROM and REQUIRE.SELECT

The **FROM** clause and the REQUIRE.SELECT clause perform the same operation. They differ when there is no active select list. If there is no active select list, the **FROM** clause is ignored and CMQL executes the other clauses of the query. If there is no active select list, **REQUIRE.SELECT** causes the query to fail with a [7013] error message.

Examples

The following example uses **SELECT** to populate Select List 3, then uses **LIST** with a **FROM** clause to query using the input set in Select List 3:

```
SELECT VOC WITH @ID > "T" TO 3
LIST VOC WITH F1="K" FROM 3
```

The **SELECT** stores 36 items in Select List 3. The **LIST** lists 19 of those items that pass the WITH clause restriction.

The following example uses **SELECT** to populate Select List 4 with all items that begin with the letter S, then uses another **SELECT** with a **FROM** clause to populate Select List 5 with the subset of those items that only contain alphabetic characters. It then uses **LIST** to list those items in uppercase and lowercase:

```
SELECT VOC WHERE @ID LIKE S... TO 4
SELECT VOC WHERE @ID LIKE "0A" FROM 4 TO 5
LIST VOC @ID CONV "MCL" FROM 5
```

See Also

- SELECT command
- REQUIRE.SELECT clause

HEADER clause

Synonym for HEADING.

Description

The **HEADER** clause is a synonym for the HEADING clause.

HEADING clause

Inserts a heading on each page of displayed output.

```
HEADING "text['code']"
```

Description

The **HEADING** and **HEADER** keywords are synonyms.

The **HEADING** clause causes the specified heading to be displayed (or printed) at the top of each page of CMQL output. This heading can consist of any combination of literal *text* and *code* characters. Note that heading literal text is in outer quotes and code characters are in nested quotes.

The available outer quotes are `"text"` or `\text\`. Inner (nested) quotes must be single quotes. Therefore, to display a text and a page number heading you must specify a heading such as `"This is page 'P' of my report"` or `\This is page 'P' of my report\`.

A heading specified in the **HEADING** clause replaces the default page heading provided with many commands. This default page heading can be suppressed using the `(H` or `(C` letter code options or the HDR-SUPP or COL-HDR-SUPP keyword options. These letter code and keyword options do not suppress a page heading specified using the **HEADING** clause.

The **HEADING** clause ignores the setting of **SP-CONDUCT** bit 1024. **HEADING** always behaves as if bit 1024 is not set.

Heading Code Characters

The following heading *code* characters can be included in a specified heading:

‘B’	The BREAK-ON field value. Refer to the BREAK-ON clause for details.
‘C’	Center the heading. The default is left justified.
‘D’	Date. Inserts the current system date in DD MMM YYYY format. For example, `25 May 2010`.
‘G’	Right justify the following *text*. The default is left justified.
‘L’	Line break inserted in the heading at this point.
‘N’	No page break prompts. Displays the header at the top of each page, displays all of the pages without waiting for user prompting for the next page.
‘P’	Page numbers incrementing from 1 with leading indent. For example, `"This is page'P' of my report"`.
‘Q’	Treat \,], and ^ as literals, not code characters for rest of header.
‘S’	Page numbers incrementing from 1 with no indent. For example, `"This is page 'S' of my report"`.

'T'	Time and date. Inserts the current system timestamp in 12–hour format: hh:mm:ss DD MMM YYYY. For example, `9:35:22am 25 May 2010`. Note that the time increments for each page.
^	Page number. Same as 'P'
\	Time and date. Inserts the current system timestamp in 12–hour format: hh:mm:ss DD MMM YYYY format. For example, `"Report printed \ for your review"`. Same as 'T'.
]	Line break. Same as 'L'
\"\	A literal double quote character. For example, `\Report is "mostly" accurate\`
\"\ or "'"	A literal single quote character, specified as two single quote characters. For example, `"Don''t review"`

You can specify more than one letter code within nested quotes. For example, `"Report printed 'CTLC' For your review"` displays a two-line heading with both lines centered, and the with first line containing the current system timestamp.

For further details on using heading codes, refer to the MVBasic HEADING command, which uses the same codes.

IF clause

Synonym for WITH.

Description

The **IF** clause is a synonym for the WITH clause.

LPTR clause

Directs query output to a printer.

```
LPTR [n]
```

Description

The **LPTR** clause directs all output from a query statement to printer channel 0. The same operation can be performed using the (P letter code option. **LPTR n** directs all output from a query statement to the printer channel specified in *n*. For further information on printer channels, refer to the "Spooler Commands" chapter of *The Caché MultiValue Spooler*.

In Caché MultiValue (and all emulations except Reality, jBase, and UniData) issuing a CMQL statement with an **LPTR** clause closes the spooler job when the CMQL program terminates. In UniData emulation, a CMQL statement gets its own unique spooler job. This default behavior can be changed using SP-CONDUCT.

MAX clause

Returns the maximum value of the values of a field.

```
MAX field [GRAND.TOTAL "label"]
```

Description

The **MAX** clause returns the maximum numeric value for a field. If a WITH (or other conditional clause) is specified, **MAX** returns the largest value of the values of that field that pass the condition test.

The following example lists all of the values of the AMOUNT field, then returns the largest of these values:

```
LIST SALES AMOUNT MAX AMOUNT
```

The following example lists all of the values of the @ID field that pass the condition test, then returns the largest of these values:

```
LIST SALES MAX @ID WITH @ID > 24990
```

It lists a result set such as the following:

```
SALES..... SALES.....

24991     24991
24992     24992
24993     24993
24994     24994
24995     24995
24996     24996
24997     24997
24998     24998
24999     24999
25000     25000

***       25000

10 Items listed.
```

The **MAX** value is listed after the detail listing, using the same format as a **TOTAL** value.

You can use the DET-SUPP keyword, or the (D letter code, to suppress listing individual values, as follows:

```
LIST SALES MAX @ID WITH @ID > 24990 DET-SUPP
```

Which returns:

```
SALES..... SALES.....

***       25000

10 Items listed.
```

The following example lists the total, maximum, and minimum values:

```
LIST SALES TOTAL @ID MAX @ID MIN @ID WITH @ID > 24990
```

It lists a result set such as the following:

```
SALES..... SALES..... SALES..... SALES.....

24991    24991    24991    24991
24992    24992    24992    24992
24993    24993    24993    24993
24994    24994    24994    24994
24995    24995    24995    24995
24996    24996    24996    24996
24997    24997    24997    24997
24998    24998    24998    24998
24999    24999    24999    24999
25000    25000    25000    25000

***      249955   25000    24991

10 Items listed.
```

Note that the maximum, minimum, and total values are all listed in the same summation line in the order in which the clauses were specified. You can clarify what these values are using the GRAND.TOTAL keyword.

GRAND.TOTAL Keyword

GRAND.TOTAL and GRAND-TOTAL are synonyms.

By default, in a horizontal display the summation line is not labeled; it is indicated by the “***” placeholder. You can use the GRAND.TOTAL keyword to assign a label name to this placeholder. GRAND.TOTAL has no effect on vertical display format.

The following example uses GRAND.TOTAL to label the summation line. Note that GRAND.TOTAL does not have to directly follow the clause(s) that it affects.

```
LIST SALES TOTAL @ID MAX @ID MIN @ID WITH @ID > 24990 GRAND.TOTAL "Sum/Max/Min"
```

It returns:

```
SALES..... SALES..... SALES..... SALES.....

24991    24991    24991    24991
24992    24992    24992    24992
24993    24993    24993    24993
24994    24994    24994    24994
24995    24995    24995    24995
24996    24996    24996    24996
24997    24997    24997    24997
24998    24998    24998    24998
24999    24999    24999    24999
25000    25000    25000    25000

Sum/Max/Min 249955   25000    24991

10 Items listed.
```

See Also

- AVG clause
- MIN clause
- TOTAL clause

MIN clause

Returns the minimum value of the values of a field.

```
MIN field [GRAND.TOTAL "label"]
```

Description

The **MIN** clause returns the minimum numeric value for a field. If a WITH clause (or some other conditional clause) is specified, **MIN** returns the smallest value of the values of that field that pass the condition test.

The following example lists all of the values of the AMOUNT field, then returns the smallest of these values:

```
LIST SALES AMOUNT MIN AMOUNT
```

The following example lists all of the values of the @ID field that pass the condition test, then returns the smallest of these values:

```
LIST SALES MIN @ID WITH @ID > 24990
```

It lists a result set such as the following:

```
SALES..... SALES.....

24991     24991
24992     24992
24993     24993
24994     24994
24995     24995
24996     24996
24997     24997
24998     24998
24999     24999
25000     25000

***       24991

10 Items listed.
```

The **MIN** value is listed after the detail listing, using the same format as a **TOTAL** value.

You can use the DET-SUPP keyword, or the (D letter code, to suppress listing individual values, as follows:

```
LIST SALES MIN @ID WITH @ID > 24990 DET-SUPP
```

Which returns:

```
SALES..... SALES.....

***       24991

10 Items listed.
```

The following example lists the total, maximum, and minimum values:

```
LIST SALES TOTAL @ID MAX @ID MIN @ID WITH @ID > 24990
```

It lists a result set such as the following:

```
SALES..... SALES..... SALES..... SALES.....

24991    24991    24991    24991
24992    24992    24992    24992
24993    24993    24993    24993
24994    24994    24994    24994
24995    24995    24995    24995
24996    24996    24996    24996
24997    24997    24997    24997
24998    24998    24998    24998
24999    24999    24999    24999
25000    25000    25000    25000

***      249955   25000    24991

10 Items listed.
```

Note that the maximum, minimum, and total values are all listed in the same summation line in the order in which the clauses were specified. You can clarify what these values are using the GRAND.TOTAL keyword.

GRAND.TOTAL Keyword

GRAND.TOTAL and GRAND-TOTAL are synonyms.

By default, in a horizontal display the summation line is not labeled; it is indicated by the “***” placeholder. You can use the GRAND.TOTAL keyword to assign a label name to this placeholder. GRAND.TOTAL has no effect on vertical display format.

The following example uses GRAND.TOTAL to label the summation line. Note that GRAND.TOTAL does not have to directly follow the clause(s) that it affects.

```
LIST SALES TOTAL @ID MAX @ID MIN @ID WITH @ID > 24990 GRAND.TOTAL "Sum/Max/Min"
```

It returns:

```
SALES..... SALES..... SALES..... SALES.....

24991    24991    24991    24991
24992    24992    24992    24992
24993    24993    24993    24993
24994    24994    24994    24994
24995    24995    24995    24995
24996    24996    24996    24996
24997    24997    24997    24997
24998    24998    24998    24998
24999    24999    24999    24999
25000    25000    25000    25000

Sum/Max/Min 249955   25000    24991

10 Items listed.
```

See Also

- AVG clause
- MAX clause
- TOTAL clause

PCT clause

Returns the percent of the total numeric values of a field.

```
PCT field
```

Description

The **PCT** and **PERCENT** keywords are synonyms.

The **PCT** clause calculates the percentage of the overall total for a numeric field. Fields containing nonnumeric values or NULL comprise 0 percent of the total.

Multiple CMQL clauses may be specified in any order. The order of application of CMQL clauses is always the same. The **PCT** clause is applied after the **WITH** clause, the **SAMPLED** clause (if present), and the **SAMPLE** clause (if present).

The following example returns the percent of total for each distinct F5 value. Note that non-numeric values are returned as 0%. The **WITH** clause eliminates records which have no value (null) for F5:

```
LIST VOC WITH F5 BY F5 BREAK.ON F5 PCT F5 DET-SUPP
```

PERCENT clause

Synonym for PCT.

Description

The **PERCENT** clause is a synonym for the PCT clause.

PREFETCH clause

Improves query performance.

```
PREFETCH
```

Description

The **PREFETCH** keyword can be used to improve performance of CMQL queries by using Caché prefetch demons to read in data from disk prior to the data being used by the query.

PREFETCH is an optional keyword that can be specified anywhere within a CMQL query.

Performance gains vary with different queries. In some cases the gains may be negligible. Other cases may get a five-fold improvement or more. Of course, if the data already exists in shared memory (used as a data cache) there is no disk I/O, and therefore no prefetch performance improvement.

Note that in all performance tuning exercises there could be a penalty to pay. For example, if by using **PREFETCH** your disks get heavier usage, this may have (or may not have) an adverse effect on concurrent interactive users.

To activate disk I/O prefetch, you must do the following:

- Add the **PREFETCH** keyword anywhere within your CMQL statement. For example:

  ```
  SELECT ACCOUNTS WITH CUSTOMER LIKE ...COOPER... PREFETCH
  ```

- Start one or more prefetch demons. The following ObjectScript example starts three prefetch demons:

  ObjectScript

  ```
  Start(njobs)
      FOR i=1:1:njobs {JOB PreFetch}
      QUIT
  PreFetch ;
      DO $ZU(180,1)
      QUIT
  ```

  From the Terminal prompt, issue:

  ```
  %SYS>DO Start^ROUTINE(3)
  ```

  Where ROUTINE is the name of the ObjectScript routine. This command starts up three background jobs starting at the PreFetch label. The call to $ZU(180,1) means the background job becomes a prefetch background demon.

Examples

This is an example of a simple CMQL statement run with and without **PREFETCH**. There is around 48 Mb of data on an installation using 64 Mb of shared memory.

```
USER:TIME COUNT Bigfile
   627706 Items counted.
 [256] Execution time 126.915667 Seconds.


USER:TIME COUNT Bigfile PREFETCH
   627706 Items counted.
 [256] Execution time 20.412388 Seconds.
```

Queries vary in their prefetch performance improvement.

REQUIRE.SELECT clause

Uses the contents of a select list as query input.

```
REQUIRE.SELECT [FROM slist]
```

Description

The **REQUIRE.SELECT** and **SELECT.ONLY** keywords are synonyms.

The **REQUIRE.SELECT** clause uses the items in an active select list as the input set for the query. You can further limit the query results using the WITH clause and other query clauses. You can use the **REQUIRE.SELECT** clause with any of the various LIST or SELECT commands; these are listed in the CMQL Commands list.

You can use the FROM keyword to specify a numbered select list. If you omit the FROM keyword, **REQUIRE.SELECT** uses Select List 0. If the select list specified in the FROM subclause is not active, **REQUIRE.SELECT** uses the default select list (Select List 0). If neither the select list specified in the (optional) FROM subclause nor the default select list is active, **REQUIRE.SELECT** generates a [7013] error.

Valid *slist* values are 0 through 10 (inclusive). Specifying a *slist* value outside of this range generates a [819] error.

Note that an active select list can only be used once.

REQUIRE.SELECT and FROM

The FROM clause and the **REQUIRE.SELECT** clause perform the same operation. They differ when there is no active select list. If there is no active select list, the **FROM** clause is ignored; CMQL executes the other clauses of the query. If there is no active select list, **REQUIRE.SELECT** causes the query to fail with a [7013] error message.

Examples

The following query lists the items selected into the default select list (Select List 0):

```
SELECT VOC WITH @ID LIKE G...
LIST.ITEM VOC REQUIRE.SELECT
```

The following query lists the items selected into Select List 7 that meet the requirements of the WITH clause:

```
SELECT VOC WITH @ID LIKE G... TO 7
LIST.ITEM VOC REQUIRE.SELECT FROM 7 WITH F1="K"
```

See Also

- SELECT command
- FROM clause

SAMPLE clause

Limits number of items returned.

```
SAMPLE n
```

Arguments

n	An integer specifying the number of items to return.

Description

The **SAMPLE** clause limits the number of items returned by the query to the first *n* items. It thus gives a sample of the items that fulfill the query conditions. If *n* is 0 or a negative integer, all items are returned. If *n* is larger than the total number of items, all items are returned.

When multiple CMQL clauses are specified the clauses may be specified in any order. The order of application of CMQL clauses is as follows: the **WITH** clause is always applied first. Its results are supplied to the **SAMPLED** clause (if present), and the results are supplied to the **SAMPLE** clause (if present).

Examples

The following example lists the first five items in the VOC:

```
LIST VOC SAMPLE 5
```

It lists #, &COMO&, &PH&, &SAVEDLISTS&, and ;.

The following example lists the first five items in the VOC that fulfill the SAMPLED clause:

```
LIST VOC SAMPLED 50 SAMPLE 5
```

It lists #, BY.EXP, CREATE.FILE, ENUMERATE, and IF.

The following example lists the first three items in the VOC that are specified in the fields list:

```
LIST VOC 'ASSOC' 'BY.EXP' 'DUMMY' 'ASC' 'ASD' SAMPLE 3
```

It lists the three fields in the order listed: ASSOC, BY.EXP, and ASC. The DUMMY field is not listed because it is not found in the VOC.

See Also

- SAMPLED clause

SAMPLED clause

Limits the items returned by sampling every *n*th item.

```
SAMPLED n
```

Arguments

n	An integer specifying the *n*th item count used for sampling.

Description

The **SAMPLED** clause limits the number of items returned by the query to a sampled subset taken every *n*th item. It thus gives a sample of the items that fulfill the query conditions. **SAMPLED** returns the first item, then the *n*+1 item, then the (*n**2)+1 item, and so forth. If *n* is 0 or a negative integer, all items are returned. If *n* is larger than the total number of items, one item is returned: the first item.

When multiple CMQL clauses are specified the clauses may be specified in any order. The order of application of CMQL clauses is as follows: the **WITH** clause is always applied first. Its results are supplied to the **SAMPLED** clause (if present), and the results are supplied to the **SAMPLE** clause (if present).

Examples

The following example samples every 100th item in the VOC:

```
LIST VOC SAMPLED 100
```

The VOC in this example contains 479 items. This program lists # (item 1), CREATE.FILE (item 101), IF (item 201), ORDER (item 301), and SP.RESUME (item 401).

The following example lists the first five items in the VOC that fulfill the SAMPLED clause:

```
LIST VOC SAMPLED 100 SAMPLE 3
```

This program lists # (item 1), CREATE.FILE (item 101), and IF (item 201).

The following example lists the every third item in the VOC that is specified in the fields list:

```
LIST VOC 'ASSOC' 'BY.EXP' 'DUMMY' 'ASC' 'ASD' 'ORDER' 'IF' 'SP.LOOK' 'SP.RESUME' SAMPLED 3
```

It lists three fields sampled in the order listed: ASSOC, ASD, and SP.LOOK. Note that the DUMMY field, which is not found in the VOC, is not used for the sampling count.

See Also

- SAMPLE clause

SAMPLING clause

Synonym for SAMPLE.

Description

The **SAMPLING** clause is a synonym for the SAMPLE clause.

SAVING clause

Saves DICT entry values as @ID values.

```
SAVING [UNIQUE] dict [NO.NULLS] [dict2 [NO.NULLS]] [...]
```

Arguments

dict	An existing DICT entry or multiple DICT entries separated by blank spaces.

Description

The **SAVING** clause saves the specified *dict* value as the @ID value. If you specify multiple *dict* arguments it saves each one as an @ID value. The *dict* argument must be an existing DICT entry in the specified file.

If you specify the UNIQUE keyword, only the unique *dict* values are saved as the @ID values. The UNIQUE keyword applies to all the *dict* arguments.

If you specify the NO.NULLS keyword, only the *dict* values that are not null are saved as the @ID values. The NO.NULLS keyword follows a *dict* argument and applies to only the *dict* argument that immediately precedes it.

Examples

The following example saves the F2 dictionary entry values from the VOC as @ID values in select list #8:

```
SELECT VOC SAVING F2 TO 8
COUNT VOC FROM 8
```

The VOC in this example contains 478 items, all of which have an F2 value, so the SELECT saves 478 items to select list #8. When the LIST command inputs these items from select list #8 and matches them with the VOC @ID values, it finds matches for 327 of them (VOC @ID value = VOC F2 value), and lists the remaining 151 items as “not found”.

The following example saves the F1 and F2 dictionary entry values from the VOC as @ID values in select list #7:

```
SELECT VOC SAVING F1 F2 TO 7
```

It saves 956 items (478 x 2).

The following example saves the unique F2 dictionary entry values from the VOC as @ID values in select list #6:

```
SELECT VOC SAVING UNIQUE F2 TO 6
```

It saves 394 unique items.

The following example saves unique F1 and F2 dictionary entry values from the VOC as @ID values in select list #5:

```
SELECT VOC SAVING F1 F2 TO 5
```

It saves 402 items (F2 = 394 unique items + F1 = 8 unique items).

The following example saves unique and non-null F1 dictionary entry values and unique F2 dictionary entry values from the VOC as @ID values in select list #4:

```
SELECT VOC SAVING UNIQUE F1 F2 NO.NULLS TO 4
```

It saves 401 items (F1 = 8 unique items + F2 = 393 unique non-null items).

SELECT.ONLY clause

Synonym for REQUIRE.SELECT.

```
SELECT.ONLY [FROM n]
```

Description

The **SELECT.ONLY** clause is a synonym for the REQUIRE.SELECT clause.

TOTAL clause

Returns the total of the values of a field.

```
TOTAL field [GRAND.TOTAL "label"]
```

Description

The **TOTAL** clause calculates the total of the numeric values of a field. A nonnumeric field has the numeric value of 0. If a WITH clause (or some other conditional clause) is specified, **TOTAL** returns the total of the values of that field that pass the condition test.

The following example lists all of the values of the F5 field, then returns the total of these values:

```
LIST VOC WHERE F5 TOTAL F5
```

It returns 52 for the total of the F5 values. This example both lists field values and returns a total of these values.

To just return the total of value for a field, you can use the **SUM** command:

```
SUM VOC F5
```

You can use the BREAK.ON clause with **TOTAL** to return subtotals. The following example returns subtotals for each distinct F5 value and a total for all F5 values. The **WITH** clause eliminates records which have no value (null) for F5:

```
LIST VOC WITH F5 NE "" BY F5 BREAK-ON F5 TOTAL F5
```

The following example uses the DET-SUPP keyword to suppress the details shown by the previous example. Only the subtotals for each F5 value and the final F5 total are listed:

```
LIST VOC WITH F5 NE "" BY F5 BREAK-ON F5 TOTAL F5 DET-SUPP
```

GRAND.TOTAL Keyword

GRAND.TOTAL and GRAND-TOTAL are synonyms.

By default, in a horizontal display the summation line is not labeled; it is indicated by the “***” placeholder. You can use the GRAND.TOTAL keyword to assign a label name to the summation line. GRAND.TOTAL has no effect on vertical display format.

The following example shows the default labeling of a total:

```
LIST VOC WHERE F5 TOTAL F5 (D
```

It returns:

```
VOC..... F5.....

***      52

48 Items listed.
```

The following example uses GRAND.TOTAL to label the total:

```
LIST VOC WHERE F5 TOTAL F5 GRAND.TOTAL "Final Tally" (D
```

It returns:

```
VOC..... F5.....

Final Tally  52

48 Items listed.
```

Note that the total, maximum, minimum, and average values are all listed as columns in the same summation line. They are listed in the order in which the clauses were specified. A single GRAND.TOTAL keyword specifies the label for the entire summation line. GRAND.TOTAL does not have to directly follow the clause(s) that it affects. If you specify multiple GRAND.TOTAL keywords, the last one specified is used.

See Also

- AVG clause
- MAX clause
- MIN clause

WHEN clause

Specifies a condition for query results.

Description

The **WHEN** clause is in most cases a synonym for the WITH clause. Refer to the **WITH** clause for details about condition expressions.

The **WHEN** clause differs from the **WITH** clause in the following respects:

- **WHEN** has no complement (inverse) keyword. **WITH** has the complement keywords **WITH NO** and **WITHOUT**.
- **WHEN** can be used to specify a conditional expression for **LISTF**, **LISTPA**, **LISTPH**, and **LISTS**. **WITH** cannot be used with these commands.
- **WHEN** requires single quote delimiters for test values. **WITH** supports single quote, double quote, and backslash delimiters. **WHEN** requires delimiters for a LIKE clause (`WHEN @ID LIKE 'A...'`); these delimiters are optional in a **WITH** clause (`WITH @ID LIKE A...`).
- Multiple **WITH** clauses must be associated with explicit OR or AND logical operators. **WHEN** clauses are specified without explicit OR or AND logical operators; the default is implicit AND. When specifying multiple **WHEN** clauses, or a **WITH** clause with one or more **WHEN** clauses, implicit AND logic is applied. You can associate two **WHEN** clauses with an explicit OR, but not a **WITH** clause and a **WHEN** clause. An explicit OR between a **WITH** clause and a **WHEN** clause is ignored; the two clauses are associated with an implicit AND. The following are valid syntactical forms that return the same results:

  WHEN <condition1> WHEN <condition2>
  WHEN <condition1> AND WHEN <condition2>
  WITH <condition1> WHEN <condition2>
  WITH <condition1> AND WHEN <condition2>
  WITH <condition1> OR WHEN <condition2> (avoid using this)

- **WHEN** can be used with the ASSOCIATED keyword, associating multiple conditional clauses using exclusive OR logic. A WHEN ASSOCIATED clause can have either of the following syntax:

  ```
  WHEN ASSOCIATED (condition) OR WHEN condition
  ```

  ```
  WHEN ASSOCIATED (condition OR condition)
  ```

  Note that the parentheses are mandatory. These parentheses may enclose a single conditional expression or multiple conditional expressions. ASD is a synonym for ASSOCIATED. The ASSOCIATED keyword cannot be use with the WITH keyword.

WHEN ASSOCIATED

The WHEN ASSOCIATED clause permits a more sophisticated use of compound conditions, using XOR (exclusive OR) logic. The CMQL default is inclusive OR logic.

The following example uses inclusive OR logic:

```
LIST VOC F1 F4 WITH (F1="V" AND F4 # "") OR WITH @ID > "W"
```

returns “25 Items listed.” and displays a list of 25 items:

```
VOC..... F1..... F4.....

BSELECT     V          L
COUNT.VERB  V          L
LIST        V          DA
LIST.ITEM   V          I
```

```
LIST.LABEL    V           D
PRINT.CATALOG V
REFORMAT      V           DA
SELECT        V           L
SORT          V           DAS
SORT.ITEM     V           IS
SORT.LABEL    V           DS
SREFORMAT     V           DAS
SSELECT       V           LS
STAT          V           J
SUM.VERB      V           T
WHEN          K
WHERE         K
WHERE.VERB    V
WHO           V
WITH          K
WITHIN        K
WITHOUT       K
Z             V
ZH            V
[             V

25 Items listed.
```

This example creates a result set by including or excluding items that meet each conditional expression.

The following WHEN ASSOCIATED statement uses exclusive OR (XOR) logic.

```
LIST VOC F1 F4 WHEN ASD (F1="V" AND F4 # "") OR WHEN @ID > "W"
```

The result set is selected using exclusive OR logic, but the count of items listed uses inclusive OR logic. Therefore, it returns “25 Items listed.” and displays a list of 20 items:

```
VOC.......  F1.............. F4..............

BSELECT       V           L
COUNT.VERB    V           L
LIST          V           DA
LIST.ITEM     V           I
LIST.LABEL    V           D
PRINT.CATALOG V
REFORMAT      V           DA
SELECT        V           L
SORT          V           DAS
SORT.ITEM     V           IS
SORT.LABEL    V           DS
SREFORMAT     V           DAS
SSELECT       V           LS
STAT          V           J
SUM.VERB      V           T
WHERE.VERB    V
WHO           V
Z             V
ZH            V
[             V

25 Items listed.
```

This example creates a result set of all records with F1="V", then removes items that don’t meet the other conditional expression criteria from this result set.

WHERE clause

Synonym for WITH.

Description

The **WHERE** clause is a synonym for the WITH clause.

WITH clause

Specifies a condition for query results.

```
WITH condition
```

Arguments

condition	A boolean condition expression, or a series of condition expressions associated with AND or OR logical operators.

Description

The **WITH** clause limits the items returned by the query to the ones whose value passes the condition test.

A **WITH** clause *condition* test consists of a boolean comparison statement, such as `field = value`, where *field* is a dictionary entry field, and *value* is a literal value. A variety of comparison operators can be used.

Multiple *condition* tests can be associated using the AND or OR logical operators: `field1 = value1 AND field2 = value2`. Further details are provided in the "Multiple Condition Tests" section below.

When multiple CMQL clauses are specified the clauses may be specified in any order. The order of application of CMQL clauses is as follows: the **WITH** clause is always applied first. Its results are supplied to the **SAMPLED** clause (if present), and the results are supplied to the **SAMPLE** clause (if present).

Synonym and Complement Keywords

The **WITH** keyword has the following synonyms: **IF**, **WHERE**, and **WHEN**. (WHEN is not an exact synonym. It differs from **WITH** in a few circumstances, and supports features not supported by **WITH**.)

The complement (inverse) of the **WITH** keyword is **WITHOUT**. The **WITHOUT** keyword has the following synonyms: **WITH NO**, **WITH NOT**; **IF NO**, **IF NOT**; **WHERE NO**, **WHERE NOT**. (**WHEN NO** and **WHEN NOT** are not valid keyword phrases.)

For example, the following are all equivalent statements:

```
LIST VOC F1 WITHOUT F1="K" AND WITHOUT F1="V"
LIST VOC F1 WITHOUT F1="K" AND WITH NO F1="V"
LIST VOC F1 WITHOUT F1="K" AND NO F1="V"
LIST VOC F1 WITH NO F1="K" AND NO F1="V"
LIST VOC F1 IF NO F1="K" AND NO F1="V"
LIST VOC F1 WHERE NO F1="K" AND NO F1="V"
```

Note that the complement keyword affects only the condition expression that immediately follows it. Subsequent conditions default to **WITH**. Thus the following example returns all records where F1 is not K:

```
LIST VOC F1 WITHOUT F1="K" OR F1="V"
```

You can use parentheses to extend the scope of a complement keyword to multiple conditions. Thus the following example returns all records where F1 is neither K nor V:

```
LIST VOC F1 WITHOUT (F1="K" OR F1="V")
```

Value Test

Specifying **WITH** *field* limits the items returned by the query to the records that have a value for the specified dictionary entry *field*. This excludes null fields.

In the following example, the **WITH** clause limits the result set to those records that have value for field F4:

```
LIST VOC F4 WITH F4
```

It lists the following 15 items: BSELECT COUNT.VERB LIST LIST.ITEM LIST.LABEL PRINT.CATALOG REFORMAT SELECT SORT SORT.ITEM SORT.LABEL SREFORMAT SSELECT STAT SUM.VERB. All of these items have an F4 field value. Note that PRINT.CATALOG is listed, but [is not, though both have an F4 field that apparently display no value. The difference is that PRINT.CATALOG has an F4 field with a value of " " (one blank space) and thus has a value; [has an F4 field that is null and thus has no value.

Null Test

There are several ways to test whether an item is null (has no assigned value). You can use a value test, an equality operator with null symbolized using `""`, or a keyword.

The following statements return non-null values and exclude nulls:

- `WITH field`
- `WITH field IS.NOT.NULL`
- `WITH field # ""`

The following statements return nulls and exclude non-null values:

- `WITH NO field`
- `WITH field IS.NULL`
- `WITH field = ""`

All values are greater than `""`, no values are less than `""`. The MultiValue equality operators can return null values. The BETWEEN clause cannot return null values. Therefore, in the following example the first two statements are functionally identical, but the second two statements are not:

```
LIST VOC WITH F4 >= "DA" AND <= "J"
LIST VOC WITH F4 BETWEEN "DA" "J"

LIST VOC WITH F4 >= "" AND <= "J"
LIST VOC WITH F4 BETWEEN "" "J"
```

Equality Operators

You can perform an equality comparison using any of the following operators:

= EQ EQUAL	Equal to.
# <> NE NOT	Not equal to.
< LT LESS BEFORE	Less than.
<= LE	Less than or equal to.
> GT GREATER AFTER	Greater than.
>= GE	Greater than or equal to.

The following example limits the query result set to those records with a field F4 value of “L”:

```
LIST VOC WITH F4="L"
```

It lists the following 3 items: BSELECT COUNT.VERB SELECT.

The following three statements are functionally identical:

```
LIST VOC WITH @ID EQ "ASSOC"
LIST VOC WITH @ID = "ASSOC"
LIST VOC WITH @ID="ASSOC"
```

Spaces before and after symbolic operators are not required, but spaces are required for alphabetic code operators.

Values should be enclosed with quote characters to avoid ambiguity, but quotes are not required in all cases, as shown in the following:

```
LIST VOC WITH F1 = "PH"
LIST VOC WITH F1 = PH
```

In Caché MultiValue, the equal sign is optional; a condition expression defaults to an equality test, as shown in the following example:

```
LIST VOC WITH F1 "PH"
```

This equality test using double quotes is supported for Caché MultiValue and all emulations *except* INFORMATION, PIOpen, Prime, UniData, and UniVerse. For further details on these emulations and the use of single quotes, refer to the Emulation section below.

The less than and greater than operators select according to ascending collation sequence. The specified limiting value does not need to exist in the file.

The following example searches the @ID dictionary entry field of the VOC for all values less than AM. The **LIST** command lists those items that pass this condition test:

```
LIST VOC WITH @ID LT "AM"
```

It lists items such as: # &COMO& &PH& &SAVEDLISTS& ; @CMQLOPTS A ABORT AFTER ALL ALL.MATCH.

The following example searches the @ID dictionary entry field of the VOC for all values greater than W. The **LIST** command lists those items that pass this condition test:

```
LIST VOC WITH @ID > "W"
```

It lists items such as: WHEN WHERE WHERE.VERB WHO WITH WITHIN WITHOUT Z ZH [.

LIKE Pattern Match Operator

The LIKE operator matches the value of a field to a pattern and returns a boolean value. A pattern may consist of either a pattern match code string, or some combination of pattern match codes and literal substrings. Pattern match codes are used to specify an expected pattern of character types to match with the field value, and/or a location in the field value to search for a literal substring. The substring may be a single character or several characters. Note that string comparisons are case-sensitive.

Synonym and Complement Keywords

The LIKE keyword operator selects values that match the pattern code. The MATCHES keyword operator, and the MATCHING keyword operator are synonyms.

The UNLIKE keyword operator reverses the sense of the pattern string. Thus `LIKE "3A"` or `MATCHES "3A"` returns three-letter words; `UNLIKE "3A"` returns all values *except* three-letter words. The NOT.MATCHING keyword operator is a synonym.

Pattern Match Codes

The following are the available pattern match codes:

Pattern Match Code	Meaning
SS... “’SS’...”	Trailing ellipsis — A literal substring (here the string SS) followed by any number of characters of any type. The ellipsis must appear at the end of a pattern match string.
...SS “...’SS’”	Leading ellipsis — Any number of characters of any type followed by a literal substring (here the string SS). The ellipsis must appear at the beginning of a pattern match string.
“nA”	A — A specified integer number (n) of alphabetic characters. You can use “0A” to specify any number of alphabetic characters. The nA pattern code may appear anywhere in a pattern match string.
“nN”	N — A specified integer number (n) of numeric characters. You can use “0N” to specify any number of numeric characters. The nN pattern code may appear anywhere in a pattern match string.
“nX”	X — A specified integer number (n) of characters of any type. You can use “0X” to specify any number of characters of any type. The nX pattern code may appear anywhere in a pattern match string.

Any of the following combinations of nested quote marks can be used: `"patcode'literal'"`, `'patcode"literal"'`, `\patcode'literal'\`, `\patcode"literal"\`. When specifying a literal with an ellipsis, quote marks are optional.

A pattern may consist of only pattern match codes, with no literals. For example, the pattern `"2A"` matches all two-letter words.

Note: Caché MultiValue does not support the use of the “}” (right curly brace) character to delimit multiple match patterns. Code from other MultiValue implementations (such as Pick and UniVerse) that uses this syntax must be changed to replace the right curly brace with the OR logical operator.

Pattern Match Examples

The following example searches the @ID dictionary entry field of the VOC for all values that begin with the letter A. The **LIST** command lists those items that pass this condition test:

```
LIST VOC WITH @ID LIKE A...
```

It lists items such as: A ABORT AFTER ALL ALL.MATCH AN AND ANY ARE AS ASC ASD ASSIGN ASSOC ASSOC.WITH ASSOCIATED ASSOCIATION AT ATTACH.ACCOUNTS AUTOLOGOUT AVERAGE AVG. Note that item A passes this test, with the ... representing no trailing characters.

The following example searches the @ID dictionary entry field of the VOC for all values that end with the letter A. The **LIST** command lists those items that pass this condition test:

```
LIST VOC WITH @ID LIKE ...A
```

It lists items such as: A CLEARDATA DATA LISTPA. Note that item A passes this test, with the ... representing no leading characters.

The following example searches the @ID dictionary entry field of the VOC for all values that contain the substring SS, including at the beginning and at the end of the @ID value. The **LIST** command lists those items that pass this condition test:

```
LIST VOC WITH @ID LIKE ...SS...
```

It lists items such as: ASSIGN ASSOC ASSOC.WITH ASSOCIATED ASSOCIATION CROSS LESS MESSAGE PAGE.MESSAGE SP-ASSIGN SP.ASSIGN SSELECT SUPPRESS UNASSIGN.

The following example matches two alphabetic characters, followed by the letter T:

```
LIST VOC WITH @ID LIKE "2A'T'"
```

It lists items such as: CRT FMT NOT PCT SET.

The following example matches two alphabetic characters, followed by the letter T, followed by two alphabetic characters:

```
LIST VOC WITH @ID LIKE "2A'T'2A"
```

It lists items such as: AFTER OUTER TOTAL.

The following example matches any value consisting of two alphabetic characters:

```
LIST VOC WITH @ID LIKE "2A"
```

It lists items such as: AN AS AT BY CS CT ED EQ GE GT IF IN IS LE LT MD ME NE NO OF ON OR SH TO ZH.

The following example matches two alphabetic characters, followed by a hyphen, followed by any number of characters of any type:

```
LIST VOC WITH @ID LIKE "2A'-'..."
```

It lists items such as: BY-DSND BY-EXP BY-EXP-DSND ID-SUP ID-SUPP NI-SUPP SP-ASSIGN SP-COPIES SP-CREATE SP-DELETE SP-DEVICE SP-EDIT SP-FORM SP-GLOBAL SP-KILL SP-OPTS SP-RESUME SP-START SP-STOP SP-SUSPEND.

SAID and SPOKEN Pattern Match Operators

The SAID pattern match operator returns data values that sound like the specified value. The first character of the SAID value must match exactly with the first letter of the data value. The other characters follow Soundex conventions. SAID pattern matching is not case-sensitive.

The following example returns “BASIC”:

```
LIST VOC WITH @ID SAID BSIK
```

These two values match because both are Soundex B220. Note that neither BSC nor BASI return “BASIC” because both of these are Soundex B200.

SAID ignores all non-alphabetic characters. For example:

```
LIST VOC F5 WITH F5 SAID CPN
```

Returns the F5 values CPUN, 2CPUN, and 2CPM.

You can specify SAID * to return all values that contain no alphabetic characters.

SPOKEN is a synonym for SAID.

For further details on Soundex analysis, refer to the MVBasic SOUNDEX function in the *Caché MultiValue Basic Reference*.

Alternative Syntax Omitting the WITH Keyword

Equality Tests without WITH

You can perform an equality condition test without the WITH keyword (or any of its synonyms). The results are the same, but the processing is different, as shown in the following examples:

```
LIST VOC F1 WITH F1="PH"
```

returns the following:

```
VOC..... F1.....

COL-HDR-SUPP PH
COL.HDR.SUPP PH

2 Items listed.
```

```
LIST VOC F1="PH"
```

returns the following:

```
VOC..... F1.....

COL-HDR-SUPP PH
COL.HDR.SUPP PH

478 Items listed.
```

Note the difference in the count of items listed. This behavior is supported for Caché MultiValue and all emulations *except* INFORMATION, PIOpen, Prime, UniData, and UniVerse.

Condition Tests on @ID without WITH

When performing an equality or LIKE condition test on the @ID dictionary entry, you can omit the WITH keyword. The @ID entry is assumed. The following pairs of commands are equivalent:

```
LIST VOC WITH @ID > "W"
LIST VOC > "W"

LIST VOC WITH @ID LIKE ...A
LIST VOC LIKE ...A

LIST VOC WITH @ID LIKE "2A'-'..." AND @ID > "SP"
LIST VOC LIKE "2A'-'..." AND > "SP"
```

Note that in these cases, there is no difference in the count of items listed. This behavior is supported in all emulations.

You can use the FOR keyword to clarify such condition tests. FOR provides no additional functionality, but can be useful in clarifying code, as shown in the following two examples:

```
LIST VOC F1 F4 LIKE ...A
```

In this example, the condition test is performed on the @ID dictionary entry, *not* on F4. Inserting a FOR keyword can help to clarify this logic, as shown in the following functionally identical example:

```
LIST VOC F1 F4 FOR LIKE ...A
```

Information on specifying an @ID value using double quotes or single quotes in different emulations, refer to the Emulation section below.

Multiple Condition Tests

A **WITH** clause can contain more than one condition test. These condition tests can be applied to the same field or to different fields. They can be associated with an explicit logical operator or with an implicit logical operator.

By default, multiple condition tests are applied in left-to-right order.

In the following example the F4 field is being tested for both having a value greater than “F” and having a value ending with the letter “S”:

```
LIST VOC F4 WITH F4>"F" AND WITH F4 LIKE ...S
```

Explicit Logical Operators

The **WITH** clause supports the AND and OR explicit logical operators.

- The AND keyword is a synonym for the & symbol. The EVERY logical operator is also a synonym for AND, as described below.
- The OR keyword specifies a logical inclusive OR. To specify a logical exclusive OR (an XOR) you must use the WHEN clause with the ASSOCIATED keyword.

When using explicit logical operators, multiple WITH keywords are optional. WITH can be specified for each condition test, or specified only for the first condition test. The following statements are equivalent:

```
LIST VOC F4 WITH F4>"F" AND WITH F4 LIKE ...S
LIST VOC F4 WITH F4>"F" AND F4 LIKE ...S
```

The following examples shows multiple condition tests with explicit logic:

```
LIST VOC F1 WITH WITH F1="PH" OR F1="F"
LIST VOC F4 WITH F4>"F" AND F4 LIKE ...S
```

The EVERY logical operator is a synonym for AND, with the following difference: EVERY can be specified before the first condition test, as well as between condition tests. The following statements are equivalent:

```
LIST VOC F4 WITH F4>"F" AND WITH F4 LIKE ...S
LIST VOC F4 WITH F4>"F" EVERY F4 LIKE ...S
LIST VOC F4 WITH EVERY F4>"F" EVERY F4 LIKE ...S
```

EACH is a synonym for EVERY.

Equality Tests and Implicit OR Logic

When performing multiple equality tests using explicit logic, you can omit repeating the equality operator for each test. For example, the following statements are equivalent:

```
LIST VOC F1 WITH F1="F" OR F1="PH"
LIST VOC F1 WITH F1="F" OR "PH"
```

When performing multiple equality tests with OR logic, you can also omit the OR keyword. For example, the following statements are equivalent:

```
LIST VOC F1 WITH F1="PA" OR "PH" OR "S"
LIST VOC F1 WITH F1="PA" "PH" "S"
```

Implicit OR logic is only supported for true equality tests (=, <=, >=), not for #, <, or > comparisons. This use of implicit OR logic for equality tests is supported in all emulations.

Conditional Tests and Implicit AND Logic

For all multiple conditional tests, except equality tests, Caché MultiValue uses implicit AND logic. For example, the following statements are equivalent:

```
LIST VOC F1 WITH F1 > "J" AND F1 < "M"
LIST VOC F1 WITH F1 > "J" < "M"
```

The following example uses implicit AND logic for condition tests on the F1 and F2 fields:

```
LIST VOC F1 WITH F1="K" F2 LIKE "...L"
```

This kind of implicit logic is emulation-dependent. Caché MultiValue and some emulations support implicit AND logic, other emulations apply implicit OR logic. These emulations are further described below.

ONLY Connective Keyword

The ONLY keyword specifies that the condition that follows it applies to the @ID field. The ID.ONLY keyword is a synonym for ONLY.

The following example uses explicit AND logic to test two different fields. It tests for the existence of an F4 value, and for a record ID having a value ending with the letter “L”:

```
LIST VOC F4 WITH F4 AND @ID LIKE ...L
```

The same results can be returned using the ONLY connective keyword, which both provides implicit AND logic and an implicit *field* of @ID:

```
LIST VOC F4 WITH F4 ONLY LIKE ...L
```

Without the ONLY connective keyword, the above example would perform a single condition test, returning F4 values that end with the letter “L”.

The ONLY keyword suppresses line wrapping of @ID values when the @ID is the only field displayed. Compare `LIST VOC WITH F2 AND @ID LIKE A...` which wraps long @ID values, and `LIST VOC WITH F2 ONLY LIKE A...` which does not wrap long @ID values.

The ONLY keyword always overrides the ID-SUPP keyword.

Order of Logical Condition Testing

In Caché MultiValue, logical conditions are tested in left-to-right order; OR and AND have equal precedence. Logical conditions can be grouped using parentheses to establish a different order of evaluation.

The following example uses left-to-right order. First all items with F1="PH" are selected, then all items with F1="F" are selected, and then the @ID value of these selected items is tested using a pattern match test:

```
LIST VOC F1 WITH F1="PH" OR F1="F" AND @ID LIKE "3A'-'0X"
```

This results in the following:

```
VOC..... F1.....

COL-HDR-SUPP PH
TCL-STACK    F
```

In the following example, all items with F1="PH" are selected, then all items with F1="F" and an @ID value tested using a pattern match test are selected:

```
LIST VOC F1 WITH F1="PH" OR (F1="F" AND @ID LIKE "3A'-'0X")
```

This results in the following:

```
VOC..... F1.....

COL-HDR-SUPP PH
COL.HDR.SUPP PH
TCL-STACK    F
```

Emulation

This section describes differences in the parsing of conditional expressions in different MultiValue emulations.

Omitting the WITH Keyword

In INFORMATION, PIOpen, Prime, UDPICK, UniData, and UniVerse emulations, the **WITH** or **WHEN** keyword is mandatory for a conditional expression clause on a specified field. When the keyword is omitted, the conditional expression always tests @ID, as shown in the following:

```
LIST VOC F1 = "A"
```

returns the @ID and F1 of the one item with @ID = “A”

```
LIST VOC F1 # "A"
```

returns the @ID and F1 for all items, except the one item with @ID = “A”.

In Caché and all other emulations the conditional expression tests the preceding field name, in this case F1.

Logical Precedence

In Caché MultiValue, logical conditions are tested in left-to-right order; OR and AND have equal precedence. In some emulations, including PICK and Reality, AND has higher precedence than OR.

Implicit Logic

Caché MultiValue uses implicit AND logic when two (or more) conditions are specified without explicit AND or OR keywords. Other emulations uses implicit OR logic in the same circumstances.

Implicit AND	Implicit OR
Cache	D3
INFORMATION	IN2
PIOpen	jBASE
Prime	MVBase
UniData	PICK
UniVerse	R83
	POWER95
	Reality
	Ultimate

Double Quotes and Single Quotes

Caché MultiValue and most emulations make a distinction between single quotes and double quotes in implied equality tests.

The following single quote syntax is parsed in Caché MultiValue and all emulations to return items that have an F1 value (F1 is not null) and that have a @ID value of ASSOC:

```
LIST VOC F1 WITH F1 'ASSOC'
```

Thus, all emulations return one item: ASSOC.

The following double quote syntax is parsed differently in Caché MultiValue and some emulations.

```
LIST VOC F1 WITH F1 "PH"
```

In Caché MultiValue and most emulations, this is parsed to return all items with F1="PH".

In INFORMATION, PIOpen, Prime, UniData, and UniVerse emulations, this is parsed to return all items that have an F1 value (F1 is not null) and that have a @ID value of “PH”. Thus, these emulations treat double quotes the same as single quotes in this type of syntax.

Pattern Match Wildcards

Some MultiValue emulations support both the LIKE clause pattern match operators and a set of pattern match wildcards which are not used with the LIKE keyword. The following wildcards are supported by MultiValue emulations:

^	A single character wildcard.
[	A multiple character wildcard. Can be zero characters, one character, or multiple characters.
]	A multiple character wildcard. Can be zero characters, one character, or multiple characters.

These wildcards are supported by D3, IN2, jBASE, MVBase, PICK, R83, POWER95, Reality, and Ultimate emulations. Caché MultiValue does not support these ^, [, or] pattern match wildcards.

In the emulations that support these wildcards, the following statement is parsed to match F5 values to the pattern of the literal character 2 followed by any single character:

```
LIST VOC F5 WITH F5="2^"
```

This is functionally identical to:

```
LIST VOC F5 WITH F5 LIKE "'2'1X"
```

In the emulations that support these wildcards, the following example is parsed to match @ID values to the pattern of the any number of characters, followed by the literal characters SU, followed by any single character, followed by the literal character P, followed by any number of characters:

```
LIST VOC WITH @ID = [SU^P]
```

This is functionally identical to:

```
LIST VOC WITH @ID LIKE "0X'SU'1X'P'0X"
```

Both examples return: COL-HDR-SUPP COL-SUPP COL.HDR.SUPP DET-SUPP HDR-SUPP ID-SUPP NI-SUPP SP-SUSPEND SP.SUSPEND SUPP SUPPRESS.