VAX Rdb/VMS

RDML Reference Manual

December 1990

This manual describes the components of the Relational Data Manipulation Language (RDML).

Revision/Update Information:	This manual is a revision and supersedes previous versions.
Operating System:	VMS VAXELN
Software Version:	VAX Rdb/VMS Version 4.0

The Reader's Comments forms at the end of this document request your critical evaluation to assist in preparing future documentation.

This document is available in printed and online versions.

This document was prepared using VAX DOCUMENT, Version 1.2.

Contents

1 Introduction

2 RDML Value Expressions

3 RDML Conditional Expressions

4 RDML Record Selection Expressions

5 RDML Statistical Functions

6 RDML Clauses and Statements

A RDML-Generated Data Types

B VAX C Language Functions for I/O Operations

Index

Figures

Tables

Preface

The Relational Data Manipulation Language (RDML) comprises clauses, expressions, and statements that can be embedded in C and Pascal programs. These programs can be processed by the RDML preprocessor, which converts the RDML statements into a series of equivalent calls to the database. Following successful preprocessing, the programmer can submit the resulting source code to the host language compiler.

Purpose of This Manual

This manual describes the syntax and semantics of all the Relational Data Manipulation Language (RDML) statements and language elements.

Intended Audience

This manual is intended for programmers who will embed RDML statements in C or Pascal programs. To get the most out of this manual, you should be proficient in either C or Pascal. You should also be familiar with data processing procedures and basic database management concepts and terminology.

Operating System Information

Information about the versions of the operating system and related software that are compatible with this version of Rdb/VMS is included with the Rdb/VMS media kit, in the *VAX Rdb/VMS Installation Guide*.

For information on the compatibility of other software products with this version of Rdb/VMS, refer to the System Support Addendum (SSA) that comes with the Software Product Description (SPD). You can use the SPD/SSA to verify which versions of your operating system are compatible with this version of Rdb/VMS.

Structure

This manual contains six chapters and two appendixes:

Chapter 1	Provides an introduction to the RDML language and the RDML preprocessor.
Chapter 2	Describes the syntax and rules of RDML value expressions.
Chapter 3	Describes the syntax and rules of RDML conditional expressions.
Chapter 4	Describes the syntax and rules of RDML record selection expressions.
Chapter 5	Describes the syntax and rules of RDML statistical functions.
Chapter 6	Describes the syntax and rules of RDML clauses and statements.
Appendix A	Contains tables listing the VAX C, VAX Pascal, and VAXELN Pascal data types that RDML generates for each data type permitted in an Rdb database.
Appendix B	Describes the sample C functions used in this manual to handle I/O tasks. This appendix also contains the source code for these functions.

Examples are provided for each statement, clause, and function described in each chapter. These examples are complete programs that you can copy and run against the PERSONNEL database.

Related Manuals

- *VAX Rdb/VMS Introduction and Master Index*

  Introduces Rdb/VMS and explains major terms and concepts. Includes a glossary, a directory of Rdb/VMS documentation, and a master index that combines entries from all the Rdb/VMS manuals.

- *VAX Rdb/VMS Guide to Using RDO, RDBPRE, and RDML*

  Describes how to use the features of Rdb/VMS to retrieve, store, change, and erase data. Shows how to write programs that use Rdb/VMS as a data access method; contains information on writing programs in high-level languages that are supported by Rdb/VMS preprocessors, including Relational Data Manipulation Language (RDML); and describes Callable RDO, an interactive utility for languages without preprocessors.

- *VAX Rdb/VMS RDO and RMU Reference Manual*

  Provides reference material and a complete description of the statements and syntax of the Rdb/VMS Relational Database Operator (RDO) interface and the commands of the Rdb/VMS Management Utility (RMU).

- *VAX Rdb/ELN Technical Overview*

  Contains an introduction to VAX Rdb/ELN concepts and components. It also has a glossary of the terms used throughout the VAX Rdb/ELN documentation set.

- *VAX Rdb/ELN Guide to Application Development*

  Describes VAX Rdb/ELN application design and development. It also describes how to define, back up, restore, and journal your VAX Rdb/ELN database.

- *VAX Rdb/VMS Guide to Distributed Transactions*

  Describes the two-phase commit protocol and distributed transactions, explains how to start and complete distributed transactions using SQL, RDBPRE, and RDML, and how to recover from unresolved transactions using RMU commands.

- *Guide to VAX C*

  Describes VAX C constructs in context with both the history of the C programming language and that of the VMS environment. It contains information on VAX C program development in the VMS environment, the VAX C programming language, and cross-system portability concerns.

- *VAX C Run-Time Library Reference Manual*

  Describes the functions and macros in the VAX C Run-Time Library.

- *VAX Pascal User's Guide*

  Describes how to interact with the VMS operating system using VAX Pascal. It contains information dealing with input and output with the Record Management System (RMS), optimizations, program section use, calling conventions, and error processing. This document is intended for programmers who have full working knowledge of Pascal.

- *Programming in VAX Pascal*

  Presents two sections: Section I introduces the Digital Command Language (DCL) and the VMS text editor (EDT), and explains how to compile, link, execute, and debug programs; Section II describes the elements of the Pascal language supported by VAX Pascal.

Syntax Diagrams

This manual presents the syntax of RDML statements with syntax diagrams. Syntax diagrams graphically portray required, repeating, and optional characteristics of any RDML statement.

To read a syntax diagram, start from the left and follow the arrows until you exit from the diagram at the right. When you come to a branch in the path, choose the branch that contains the option you want. If you want to omit an option, choose the path with no language elements. If a diagram occupies more than one horizontal line, the arrow returns to the left margin. Syntax diagrams can contain:

Names of syntax diagrams	If a diagram is named, the name is in lowercase type followed by an equal sign and appears above and to the left of the diagram. Syntax diagrams can refer to each other by name. The equal sign (=) indicates that the name is equivalent to the diagram and that the diagram can be substituted wherever the name appears.
	If the diagram contains the name of a second diagram, substitute the second diagram where its name appears. Such a substitution is similar to putting the name of a field where "field-name" appears. Most named syntax diagrams appear as subdiagrams following the main diagram.
Keywords	Keywords appear in uppercase type. If a keyword is underlined, you must include it in the statement. A keyword without underlining is optional; however, it makes the statement more readable. Omitting an optional keyword does not change the meaning of a statement.
Punctuation marks	Punctuation marks are included in the diagram when required by the syntax of the command or statement. All punctuation marks shown are required.
User-supplied elements	User-supplied elements appear in lowercase type. These elements can include names, expressions, and literals. They usually follow the diagram.

You can learn the syntax of a command or statement by reading that statement's syntax diagram.

ERASE ⟶ context-var
on-error

on-error =

ON ERROR ⟶ statement ⟶ END_ERROR
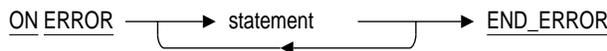
ERASE	Is in uppercase type and underlined on the main line of the diagram. Therefore, you must supply the keyword (which can usually be abbreviated).
context-var	Is in lowercase type on the main line of the diagram. Therefore, you must supply a substitute for context-var. The commentary following the diagram describes the possible values and the function for the user-supplied element, in this case context-var.
on-error	Is in lowercase type on a branch. Because it parallels an empty branch, the on-error clause is optional. The subdiagram expands the definition of on-error.
statement	Is in lowercase type on a main branch. The on-error clause is optional, but if you include it, you must have ON ERROR, at least one statement, and END_ERROR. The optional reverse loop under the *statement* indicates that more than one statement can appear within the ON ERROR . . . END_ERROR block.

All lowercase words are explained in the argument list that follows the diagram. Some explanations refer you to other diagrams that appear elsewhere in this manual.

Conventions

In examples, an implied carriage return occurs at the end of each line, unless otherwise noted. You must press the RETURN key at the end of a line of input.

Often in examples the prompts are not shown. Generally, they are shown where it is important to depict an interactive sequence exactly; otherwise, they are omitted, to focus full attention on the statements or commands themselves.

The section explains the conventions used in this manual:

CTRL/x	This symbol in examples tells you to press the CTRL (control) key and hold it down while pressing the specified letter key.
RETURN	This symbol in examples indicates the RETURN key.
TAB	This symbol in examples indicates the TAB key.
. . .	A vertical ellipsis in an example means that information not directly related to the example has been omitted.
. . .	A horizontal ellipsis in statements or commands means that parts of the statement or command not directly related to the example have been omitted.

e, f, t	Index entries in the printed manual may have a lowercase e, f, or t following the page number; the e, f, or t is a reference to the example, figure, or table, respectively, on that page.
< >	Angle brackets enclose user-supplied names.
[ ]	Brackets enclose optional clauses from which you can choose one or none.
$	The dollar sign represents the DIGITAL Command Language prompt. This symbol indicates that the DCL interpreter is ready for input.
UPPERCASE	Statements appearing in uppercase type in programming examples are RDML statements.
lowercase	Statements appearing in lowercase type in programming examples are host language statements (C or Pascal).

References to Products

This document often refers to the following products by their abbreviated names:

- VAX C software is referred to as C.

- VAX Pascal and VAXELN Pascal software are referred to as Pascal. When the use of a language statement is not the same in both the VAXELN and VMS environments, that language is specified as VAXELN Pascal or VAX Pascal.

- VAX Rdb/VMS and VAX Rdb/ELN relational database systems are referred to as Rdb. When the use of an RDML statement is different for one database system, that product is specified as Rdb/VMS or Rdb/ELN.

- VAX CDD/Plus software is referred to as the data dictionary or the dictionary.

- VIDA software is referred to as VIDA.

Technical Changes and New Features

This section presents a list of some of the new Rdb/VMS Version 4.0 features and technical changes that are described in this manual. See the Version 4.0 *VAX Rdb/VMS Release Notes* for more information about all the new Version 4.0 features and technical changes, and for reports of current limitations or restrictions.

The Version 4.0 major new features and technical changes described in this manual include:

- Support for the two-phase commit protocol
- Support for internationalization in the following areas:
  - DEC Multinational Character Set (MCS) characters in database object names

    Database object names that can be referred to from RDML include context variables, relations, fields, and constraint names.
  - Date and time literals appearing in RDML queries

    These date and time literals are converted at compile time using the VMS V5.0 RTL routine LIB$CONVERT_DATE_STRING.

    If the logical name SYS$LANGUAGE is set to the appropriate language (SPANISH, for example) and an appropriate LIB$DT_INPUT_FORMAT is defined, then RDML will compile the query.

    If the language or input format is set incorrectly, RDML generates an appropriate date conversion error.
  - Date and time support for headers in RDML programs

    If the logical name SYS$LANGUAGE is set to the appropriate language and the logical name LIB$DT_FORMAT is defined, the date and time appearing in RDML listing files will be formatted using the formats specified in LIB$DT_FORMAT.

- Technical changes have been minimal in this manual and have been made only where necessary in order to provide technical clarifications, to fix errors of omission, and to make corrections.

1

Introduction

This chapter provides a brief overview of the Relational Data Manipulation
Language (RDML) and the RDML preprocessor.

1.1 RDML Language

RDML, the language, is a set of data manipulation statements, clauses,
expressions, and functions that can be embedded in VAX C and VAX Pascal
programs to access an Rdb/VMS or Rdb/ELN database.

1.1.1 RDML Language Elements

The RDML language elements fall into five broad categories:

- Value expressions

- Conditional expressions

- Record selection expressions

- Statistical functions

- Clauses and statements

1.1.1.1 Value Expressions A value expression is a symbol or string of
symbols used to calculate a value. Value expressions allow you to perform
arithmetic calculations on database values, so that, for example, you could
double each employee's salary by using one expression, rather than modifying
the value of each employee's salary one by one. Host language variables also
fall into the category of value expressions. By using host language variables
in your application you allow the end user to decide which value Rdb/VMS
will retrieve from the database. For a complete list and information on value
expressions, see Chapter 2.

1.1.1.2 Conditional Expressions A conditional expression, sometimes called a Boolean expression, represents the relationship between two value expressions. Conditional expressions can be used to retrieve a subset of records from a relation on the basis of requirements you specify. For example, you can specify that you want Rdb to return only those records in the EMPLOYEES relation in which an employee's last name begins with S. For a complete list and information on conditional expressions, see Chapter 3.

1.1.1.3 Record Selection Expressions A record selection expression (RSE) is an expression that defines specific conditions individual records must meet before Rdb includes them in a record stream. A record stream is a temporary group of related records that satisfies the conditions you specify in the record selection expression. With a record selection expression, you can specify that you want Rdb to retrieve only those records in the EMPLOYEES relation that have a corresponding record in the COLLEGES relation. For a complete list and information on record selection expressions, see Chapter 4.

1.1.1.4 Statistical Functions Statistical functions calculate values based on a value expression for every record in a record stream. Expressions that use statistical functions are sometimes called aggregate expressions, because they calculate a single value for a collection of records. For example, you could use a statistical function to find the total number of employees in the database, or the total number of employees in a department. For a complete list and information on statistical functions, see Chapter 5.

1.1.1.5 Clauses and Statements RDML clauses and statements are the basic elements of the RDML language; they allow you to start and end a transaction, step through a record stream, add new records, modify existing records, or delete records. They are also the elements that can make programming easier by providing standardized ways to define host language variables and host language functions to hold database values. For a complete list and information on RDML clauses and statements, see Chapter 6.

1.1.2 RDML in the Rdb/VMS and Rdb/ELN Environments

All RDML language elements can be used in both Rdb/VMS and Rdb/ELN environments. However, two RDML language elements have meaning only within the Rdb/ELN environment. They are:

- The PREPARE statement
- The CONCURRENCY option of the START_TRANSACTION statement

Both of these RDML language elements may be used in programs that access an Rdb/VMS database; however, they will have no effect in that environment.

1.1.3 Data Definition and RDML

RDML does not include data definition statements. In order to perform data definition tasks you must use:

- The SQL interactive environment, an SQL program, the Relational Database Operator (RDO), or the Callable RDO program interface in the Rdb/VMS environment. RDO and the SQL interactive environment are interactive interfaces available to Rdb/VMS users. Callable RDO lets your RDML program communicate with Rdb/VMS using a callable procedure, RDB$INTERPRET. Calls to RDB$INTERPRET may be embedded in your RDML program to perform data definition tasks. For more information on using SQL, see the *VAX Rdb/VMS Guide to Using SQL*. For more information on RDO, see the *VAX Rdb/VMS Guide to Using RDO, RDBPRE, and RDML*. For more information on Callable RDO, see the *VAX Rdb/VMS Guide to Using RDO, RDBPRE, and RDML*.

- ERDL, the Rdb/ELN data definition language (DDL) compiler in the Rdb/ELN environment. By creating an Rdb/ELN DDL file on the Rdb/ELN development system and processing it with ERDL, you can perform data definition tasks. For more information on ERDL, see the *VAX Rdb/ELN Guide to Application Development*.

1.1.4 RDML Keywords and Naming Conventions

When you create a name for a context variable, database handle, or stream, make sure you do not choose RDML keywords for these names. RDML keywords are listed in Table 1–1. Also, do not use context variables or database handle names that are the same as the name of a relation in your database. You may, however, use field names that are the same as RDML keywords or relation names.

Table 1–1   RDML Keywords

ALPHABETIZED	ERASE	ON
AND	ERROR	ON_ERROR
ANY	EVALUATING	OR
AS	EXCLUSIVE	OVER
ASC	EXTERN	PATHNAME
ASCENDING	EXTERNAL	PREPARE
AT	FETCH	PROTECTED
AVERAGE	FILENAME	RDB$LENGTH
BASED	FINISH	RDB$MISSING
BATCH_UPDATE	FIRST	RDB$VALUE
BETWEEN	FOR	READ
BY	FROM	READ_ONLY
COMMIT	GE	READ_WRITE
COMMIT_TIME	GET	READY
COMPILETIME	GLOBAL	REDUCED
CONCURRENCY	GREATER_EQUAL	REQUEST_HANDLE
CONSISTENCY	GREATER_THAN	RESERVING
CONTAINING	GT	ROLLBACK
COUNT	IN	RUNTIME
CROSS	INVOKE	SAME
DATABASE	IS	SCOPE
DBKEY	LE	SHARED
DECLARE_STREAM	LENGTH	SORTED
DECLARE_VARIABLE	LESS_EQUAL	STARTING
DEFAULT	LESS_THAN	START_STREAM
DEFAULTS	LOCAL	START_TRANS
DESC	LT	STORE
DESCENDING	MATCHING	TO
DIV	MAX	TOTAL
END	MIN	TRANSACTION_HANDLE
END_ERROR	MISSING	UNIQUE
END_FETCH	MODIFY	USING
END_FOR	NE	VALUE
END_GET	NOT	VERB_TIME
END_MODIFY	NOT_EQUAL	WAIT
END_STORE	NOWAIT	WITH
END_STREAM	OF	WRITE
EQ		

1.1.5 Multinational Characters in Rdb/VMS Metadata Names

Rdb/VMS supports the use of the DEC Multinational Character Set (MCS) in database object names such as relations, fields, indexes, and constraints. Context variables can also contain MCS characters.

Required support in the language processors includes: fields, relations and context variables.

Note the following restrictions:

- Database handles are declared as host language variables and used as PSECT names, therefore, these cannot contain MCS characters due to current VMS and host language restrictions.

- Transaction and request handles are assumed to be host language variables, therefore, these cannot contain MCS characters due to current host language restrictions.

- The included CDD/Plus record may not be acceptable to the host language if the fields contain MCS characters.

  The CDO DEFINE FIELD syntax allows the NAME FOR {language} clause to be used for some VMS languages to override the default field name. Being able to override the default field name permits records that are based on Rdb/VMS relations to be included and to succeed in executing even if MCS characters are used.

- The GET *, STORE *, and MODIFY * statements assume that the record field names (from CDD/Plus or declared manually) match the relation field names precisely.

  This mean that fields with MCS characters will generate illegal host language variable names.

  The workaround is to name each field specifically and avoid the wildcard field syntax.

1.1.6 Conversion of Date and Time Literals in RDML Queries

RDML converts date and time literals that appear in queries at compile time to international date and time formats. RDML converts the date and time literals by using the VMS V5.0 (or higher) Run-Time Library (RTL) routine LIB$CONVERT_DATE_STRING. The input formats for this RTL routine are described in the *VMS RTL Library (LIB$) Manual*.

If the logical name SYS$LANGUAGE is set to the appropriate language and the appropriate LIB$DT_INPUT_FORMAT logical name is defined, then RDML will compile the query.

If the language or input format is set incorrectly, RDML generates an appropriate date conversion error.

The following example assumes that the SYS$LANGUAGE logical name has been set to SPANISH:

```
FOR E IN EMPLOYEES
    WITH E.BIRTHDAY = '1 abril, 1990'

END_FOR
```

1.1.7 Copying Date and Time Literals Within RDML Host Language Programs

This section describes how to copy date and time literals within host language programs.

For C programs you can use either of the following:

- The memcpy function

  When the value of J.JOB_START has been stored in the database, you can use the *memcpy* function for copying the date from the field, J.JOB_START, to the field, J.JOB_END, as shown in the following example:

  ```
  memcpy (J.JOB_END,J.JOB_START,8)
  ```

  The third argument, 8, indicates the length in bytes of the field that is copied.

Note *Do not use* strcpy *function to copy date literals from field to field.*

- An assignment statement that copies the date to a field

  In the following example, you assign J.JOB_END to a literal date string:

  ```
  J.JOB_END = '23-APR-1990'
  ```

For Pascal programs, you can use the := assignment statement for copying the date to or from a field. In the following example, the field, J.JOB_START, is copied to the field, J.JOB_END:

```
J.JOB_START := '1-JAN-1989'
J.JOB_START := J.JOB_END
```

For more information see the examples in Section 6.16.

1.1.8 Date and Time Support for Headers in RDML Program Listings

RDML provides support for international date and time formats in the headers of program listings.

If the logical name SYS$LANGUAGE is set to the appropriate language and the logical name that converts program headers, LIB$DT_FORMAT, is defined, then the date and time appearing in RDML listing files will be formatted using the formats specified in the LIB$DT_FORMAT logical name.

1.2 RDML Preprocessor

The RDML preprocessor converts RDML statements embedded in a VAX C or
VAX Pascal program into a series of equivalent RDB$ . . . calls to Rdb/VMS.
Following successful preprocessing, you can submit your program to the host
language compiler.

Note *RDML/C programs are case sensitive. In addition to following the VAX C
conventions about the use of upper case and lower case, you must use upper case
for* all *RDML language elements in RDML/C programs. RDML/Pascal is not
case sensitive.*

For information on preprocessing, linking, and running an RDML program, see
the *VAX Rdb/VMS Guide to Using RDO, RDBPRE, and RDML* for Rdb/VMS
applications or the *VAX Rdb/ELN Guide to Application Development* for
Rdb/ELN applications.

2

RDML Value Expressions

This chapter describes the Relational Data Manipulation Language (RDML) value expressions that can be used with embedded RDML statements in C and Pascal programs.

The C and Pascal programs in this chapter access the sample personnel database provided with Rdb/VMS and Rdb/ELN.

A **value expression** is a symbol or string of symbols used to calculate a value. When you use a value expression in a statement, Rdb calculates the value associated with the expression and uses it when executing the statement.

Format

value-expr =

```
                ──────►  host-variable  ──────────────────────────►
                ──────►  db-field     ────────────────
                ──────►  quoted-string ───────────────
                ──────►  numeric-literal ─────────────
                ──────►  statistical-expr ────────────
                ──────►  arithmetic-expr ─────────────
                ──────►  missing-value ──────────────
                ──────►  first-expr   ────────────────
                ──────►  concat-expr  ────────────────
                ──────►  dbkey        ────────────────
                ──────►  ( ──►  value-expr  ──► ) ────
                ──────►  -  ──►  value-expr  ─────────
                ──────►  +  ──►  value-expr  ─────────
```

Table 2–1 summarizes the function of each value expression.

Table 2–1   Value Expressions

Value Expression	Function
Arithmetic	Combines arithmetic operators with numeric values, numeric host language variables, and/or numeric database fields.
Concatenated	Consists of the concatenate operator (	) and two value expressions. Joins the second value expression to the first value expression.
Database field	Consists of a context variable and a field name. Use a context variable as a temporary name for a relation. You define a context variable in a record selection expression.
FIRST FROM	Returns the first value from the record stream, formed by a record selection expression. Use to find the first record that contains a value that you specify.
Host language variable	Holds data to be passed between your calling program and your database system. A host language variable is a program variable in your host language.

(continued on next page)

Table 2–1 (Cont.)   Value Expressions

Value Expression	Function
RDB$DB_KEY	Returns a logical key to a specific record by using an internal system pointer. Use to retrieve a specific record from the database.
RDB$MISSING	Returns the constant that is the missing value. If you use this value to store or modify a field, it will be marked as empty. No data will be stored in the field.
Statistical	Uses functions, such as AVERAGE or MAX. Use to calculate values based on a value expression for every record in a record stream. Statistical expressions are described in Chapter 5.

2.1 Arithmetic Value Expression

Use an arithmetic value expression to combine arithmetic operators with numeric values, numeric host language variables, and database fields.

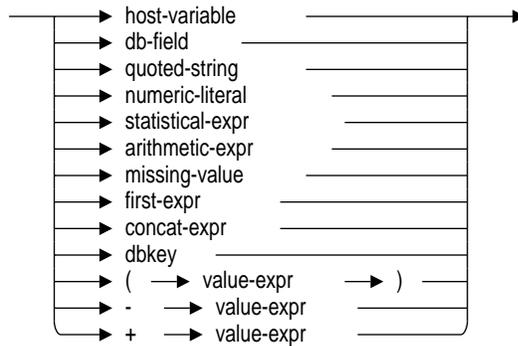When you use an arithmetic value expression in a statement, Rdb calculates the value associated with the expression and uses that value when executing the statement. Therefore, an arithmetic expression must result in a value. If either operand of an arithmetic expression is a missing value, the resultant value also is missing.

Format

arith-expr =

```
      ┌─► numeric-value          ┌─► +  ┌─► numeric-value
──────┼─► numeric-host-var    ───┼─► -  ┼─► numeric-host-var ───►
      └─► numeric-db-field        ├─► *  └─► numeric-db-field
                                  └─► /
```
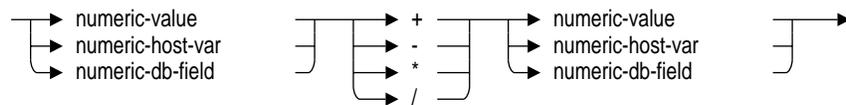
Arguments

numeric-value
A numeric literal.

numeric-host-var
A host language variable that holds a numeric value.

numeric-db-field
A database field (qualified with a context variable) that holds a numeric value.

+ – * /
Arithmetic operators. Table 2–2 lists the arithmetic operators and their functions.

Arithmetic Value Expression

|

The concatenation operator. A concatenated expression is a value expression
that combines two other value expressions by joining the second to the end of
the first.

Table 2–2   Arithmetic Operators and Functions

Operator	Function
+	Add
–	Subtract
*	Multiply
/	Divide

Usage Notes

- The minus sign (–) is also used as the unary operator for negation.

- You do not have to use spaces to separate arithmetic operators from value
  expressions.

- You can combine value expressions of any kind—including numeric
  expressions, string expressions, and literals—with the concatenation
  operator.

- You can use parentheses to control the order in which Rdb performs
  arithmetic operations. Rdb evaluates arithmetic expressions in the
  following order:

  1 Value expressions in parentheses

  2 Unary negation

  3 Multiplication and division, from left to right

  4 Addition and subtraction, from left to right

  5 Concatenation, from left to right

Arithmetic Value Expression

Examples

Example 1

The following programs demonstrate the use of the multiplication (*) arithmetic operator and the MODIFY statement. These programs select the record of an employee in the SALARY_HISTORY relation with the specified employee ID and with no value for the SALARY_END field. The purpose of specifying the MISSING option for the SALARY_END field is to ensure that the only salary amount affected is the employee's present salary. Next, the employee's salary is multiplied by 1.1 to produce a 10% salary increase. The MODIFY statement replaces the old value in this employee's SALARY_ AMOUNT field with the new value.

C Program

```
#include <stdio.h>
DATABASE PERS = FILENAME "PERSONNEL";

main()
{
READY PERS;
START_TRANSACTION READ_WRITE;

FOR SH IN SALARY_HISTORY
   WITH SH.EMPLOYEE_ID = "00164"
   AND SH.SALARY_END MISSING
      MODIFY SH USING
         SH.SALARY_AMOUNT = SH.SALARY_AMOUNT * 1.1;
      END_MODIFY;
END_FOR;

ROLLBACK;
FINISH;
}
```

Pascal Program

```
program multiply (input,output);
DATABASE PERS = FILENAME 'PERSONNEL';

begin
READY PERS;
START_TRANSACTION READ_WRITE;
```

```
FOR SH IN SALARY_HISTORY
   WITH SH.EMPLOYEE_ID = '00164'
   AND SH.SALARY_END MISSING
      MODIFY SH USING
          SH.SALARY_AMOUNT := SH.SALARY_AMOUNT * 1.1;
      END_MODIFY;
END_FOR;

ROLLBACK;
FINISH;
end.
```

Example 2

The following programs demonstrate the use of the subtraction (–) arithmetic operator, the CROSS clause, and the MODIFY statement. These programs decrease a selected employee's salary by an amount you enter from the keyboard while the program runs. To achieve this interactive processing, these programs declare the host language variable, *deduction*, with the DECLARE_ VARIABLE clause. For more information on the DECLARE_VARIABLE clause, see Chapter 6.

Additionally, the C program declares and uses a function named read_float. This function (described in Appendix B) causes the program to prompt for, and store, a value for *deduction*. The Pascal readln and writeln statements perform a similar function.

After you enter a value for *deduction*, the programs join records from the EMPLOYEES and SALARY_HISTORY relations over the common field, EMPLOYEE_ID. This creates a record stream consisting of the records specified by E.EMPLOYEE_ID that have no value stored in the SALARY_END field. By specifying SALARY_END as MISSING, these programs will select only the current SALARY_HISTORY record for the employee. The value of *deduction* is subtracted from the selected employee's salary amount. The MODIFY statement stores a value of 1 in the SALARY_AMOUNT field for that employee.

C Program

```
#include <stdio.h>
DATABASE PERS = FILENAME "PERSONNEL";

extern float read_float();
static DECLARE_VARIABLE deduction SAME AS SALARY_HISTORY.SALARY_AMOUNT;

main()
{
deduction =  read_float("Amount to be deducted for malfeasance:");
```

Arithmetic Value Expression

```
READY PERS;
START_TRANSACTION READ_WRITE;

FOR E IN EMPLOYEES CROSS SH IN SALARY_HISTORY
   OVER EMPLOYEE_ID WITH E.EMPLOYEE_ID = "00164"
   AND SH.SALARY_END MISSING

   MODIFY SH USING
      SH.SALARY_AMOUNT = SH.SALARY_AMOUNT - deduction;
   END_MODIFY;

END_FOR;

ROLLBACK;
FINISH;
}
```

Pascal Program

```
program subtract (input,output);
DATABASE PERS = FILENAME 'PERSONNEL';

var
     DECLARE_VARIABLE deduction SAME AS SALARY_HISTORY.SALARY_AMOUNT;

begin

write  ('Amount to be deducted for malfeasance:');
readln (deduction);

READY PERS;
START_TRANSACTION READ_WRITE;

FOR E IN EMPLOYEES CROSS SH IN SALARY_HISTORY
   OVER EMPLOYEE_ID WITH E.EMPLOYEE_ID = '00164'
   AND SH.SALARY_END MISSING

   MODIFY SH USING
      SH.SALARY_AMOUNT := SH.SALARY_AMOUNT - deduction;
   END_MODIFY;

END_FOR;

ROLLBACK;
FINISH;
end.
```

2.2 Database Field Value Expression

Use the database field value expression to refer to specific database fields in
record selection expressions and in other value expressions.

Format

db-field-expr

⟶ context-var ⟶ . ⟶ field-name ⟶

Arguments

context-var
A context variable. A temporary name that you associate with a relation.
You define a context variable in a relation clause. See Chapter 4 for more
information.

field-name
The name of a field in a relation.

Usage Notes

- If you access several record streams at once, the context variable lets you
  distinguish among fields from different records, even if different fields have
  the same name.

- If you access several record streams at once that consist of the same
  relation and fields within that relation, context variables let you
  distinguish among the record streams.

- The context established by the context variable is valid during the
  execution of the statement or clause in which the context variable is
  declared. See Chapter 4 for more information on context variables.

Database Field Value Expression

Examples

Example 1

The following programs demonstrate the use of the database field value expression. These programs use the clause, FOR J IN JOBS, to declare the context variable J. This allows the programs to use the database field value expression, J.JOB_CODE, to mean the JOB_CODE field from the JOBS relation. These programs search the JOB_CODE field for the string "APGM". Any record that contains the specified string becomes part of the record stream. These programs then use the context variable J to qualify the fields in the host language print statements. The job title, minimum salary, and the maximum salary for each record in the record stream are printed.

C Program

```
#include <stdio.h>
DATABASE PERS = FILENAME "PERSONNEL";

main()
{
READY PERS;
START_TRANSACTION READ_ONLY;

FOR J IN JOBS WITH J.JOB_CODE = "APGM"
   printf ("%s", J.JOB_TITLE);
   printf ("  $%f", J.MINIMUM_SALARY);
   printf ("  $%f\n", J.MAXIMUM_SALARY);
END_FOR;

COMMIT;
FINISH;
}
```

Pascal Program

```
program fld_value (input,output);
DATABASE PERS = FILENAME 'PERSONNEL';

begin
READY PERS;
START_TRANSACTION READ_ONLY;

FOR J IN JOBS WITH J.JOB_CODE = 'APGM'
   writeln (J.JOB_TITLE,
            ' $', J.MINIMUM_SALARY: 10 : 2,
            '  $', J.MAXIMUM_SALARY: 10 : 2);
END_FOR;
```

Database Field Value Expression

```
COMMIT;
FINISH;
end.
```

Example 2

The following programs demonstrate the use of a database field value expression to qualify fields in a CROSS clause, a SORTED BY clause, and a REDUCED TO clause of a record selection expression. These programs:

- Declare the context variables E for EMPLOYEES and SH for SALARY_ HISTORY

- Using a CROSS clause, join these two relations on the basis of their common field, EMPLOYEE_ID (that is, E.EMPLOYEE_ID and SH.EMPLOYEE_ID)

- Reduce the record stream so that the only values returned are unique combinations of the values in SH.SALARY_AMOUNT, E.LAST_NAME, and E.EMPLOYEE_ID

- Sort the record stream on the basis of the database fields, E.LAST_NAME, SH.SALARY_AMOUNT, and E.EMPLOYEE_ID

- Display fields from the two relations

C Program

```
#include <stdio.h>
DATABASE PERS = FILENAME "PERSONNEL";

main()
{
READY PERS;
START_TRANSACTION READ_ONLY;

FOR E IN EMPLOYEES CROSS SH IN SALARY_HISTORY OVER EMPLOYEE_ID
   REDUCED TO E.LAST_NAME, SH.SALARY_AMOUNT, E.EMPLOYEE_ID
   SORTED BY E.LAST_NAME, SH.SALARY_AMOUNT, E.EMPLOYEE_ID
      printf ("%s  ", E.EMPLOYEE_ID);
      printf ("%s  ", E.LAST_NAME);
      printf ("%f\n", SH.SALARY_AMOUNT);
END_FOR;

COMMIT;
FINISH;
}
```

Database Field Value Expression

Pascal Program

```
program two_rel (input,output);
DATABASE PERS = FILENAME 'PERSONNEL';

begin
READY PERS;
START_TRANSACTION READ_ONLY;

FOR E IN EMPLOYEES CROSS SH IN SALARY_HISTORY OVER EMPLOYEE_ID
    REDUCED TO E.LAST_NAME, SH.SALARY_AMOUNT, E.EMPLOYEE_ID
    SORTED BY E.LAST_NAME, SH.SALARY_AMOUNT, E.EMPLOYEE_ID
      writeln (E.EMPLOYEE_ID, ' ', E.LAST_NAME, ' ', SH.SALARY_AMOUNT:10:2);
END_FOR;

COMMIT;
FINISH;
end.
```

2.3 FIRST FROM Value Expression

The FIRST FROM value expression causes Rdb to return the first record in
the record stream that matches the record selection expression specified in
the FIRST FROM value expression. If there are no matches, you receive a
run-time error.

Format

first-from-expr =
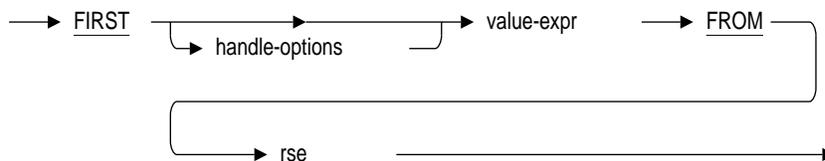
handle-options =
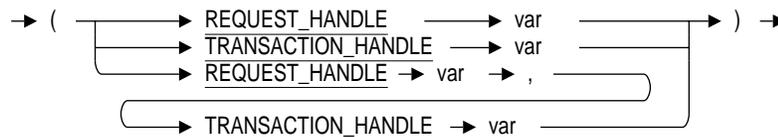
Arguments

handle-options
A transaction handle, a request handle, or both.

REQUEST_HANDLE var
A REQUEST_HANDLE keyword followed by a host language variable. A
request handle identifies a compiled Rdb/VMS request. If you do not supply
a request handle explicitly, RDML generates a unique request handle for the
compiled request. See Section 6.20 for more information on request handles.

FIRST FROM Value Expression

TRANSACTION_HANDLE var
A TRANSACTION_HANDLE keyword followed by a host language variable. A transaction handle identifies a transaction. If you do not supply a transaction handle explicitly, RDML uses the default transaction handle.

value-expr
A value expression. A symbol or a string of symbols used to calculate a value. When you use a value expression in a statement, Rdb calculates the value associated with the expression and uses that value when executing the statement.

rse
A record selection expression. A phrase that defines specific conditions that individual records must meet before Rdb includes them in a record stream. See Chapter 4 for more information.

Usage Notes

- The following statements (using Pascal) produce the same answer if there is exactly one employee with the specified ID number:

  - GET statement with FIRST FROM value expression:

    ```
    GET
      id =  FIRST E.STATE FROM E IN EMPLOYEES
            WITH E.EMPLOYEE_ID = '00176';
    END_GET;
    ```

  - FOR statement with restrictive record selection expression:

    ```
    FOR FIRST 1 E IN EMPLOYEES WITH E.EMPLOYEE_ID = '00176'
       writeln (E.STATE);
    END_FOR;
    ```

  - writeln statement with a FIRST FROM expression with a host language statement:

    ```
    writeln (FIRST E.STATE FROM E IN EMPLOYEES
             WITH E.EMPLOYEE_ID = '00176');
    ```

- However, Digital recommends that you use the GET statement instead of the host language display statement. The GET statement supports the ON ERROR clause and thereby allows you to identify errors that might occur during the GET operation.

Furthermore, when you use the GET statement, RDML generates its own code to retrieve the database value; when you use a host language display statement, RDML calls a function to retrieve the database value and thereby increases the overhead associated with the query.

Examples

Example 1

The following programs demonstrate the use of the FIRST FROM value expression. These programs find and print the first occurrence of a supervisor ID that is the same as the specified employee ID from the CURRENT_JOB relation.

C Program

```
#include <stdio.h>
DATABASE PERS = FILENAME "PERSONNEL";

DECLARE_VARIABLE id SAME AS PERS.CURRENT_JOB.EMPLOYEE_ID;

main()
{
READY PERS;
START_TRANSACTION READ_ONLY;

GET
  id = FIRST CJ.SUPERVISOR_ID FROM CJ IN CURRENT_JOB
       WITH CJ.EMPLOYEE_ID = "00200"
       SORTED BY CJ.EMPLOYEE_ID;
END_GET;

printf ("Id is  %s", id);

COMMIT;
FINISH;
}
```

Pascal Program

```
program first_value (input,output);
DATABASE PERS = FILENAME 'PERSONNEL';

DECLARE_VARIABLE id SAME AS PERS.CURRENT_JOB.EMPLOYEE_ID;

begin
READY PERS;
START_TRANSACTION READ_ONLY;
```

FIRST FROM Value Expression

```
GET
 id =   FIRST CJ.SUPERVISOR_ID FROM CJ IN CURRENT_JOB
        WITH CJ.EMPLOYEE_ID = '00200'
        SORTED BY CJ.EMPLOYEE_ID;
END_GET;

writeln (id);

COMMIT;
FINISH;
end.
```

Example 2

The following programs demonstrate the use of the FIRST FROM value
expression. The programs find the first record in the JOBS relation with
the value "Company President" in the JOB_TITLE field. Using this record's
value for JOB_CODE, these programs create a record stream that contains
the records in the CURRENT_JOB relation that have this same job code. The
programs print the value that the first record from this record stream holds in
the LAST_NAME field.

C Program

```
#include <stdio.h>
DATABASE PERS = FILENAME "PERSONNEL";

DECLARE_VARIABLE name SAME AS PERS.CURRENT_JOB.LAST_NAME;
main()
{
READY PERS;
START_TRANSACTION READ_ONLY;

GET
 name =   FIRST CJ.LAST_NAME FROM CJ IN CURRENT_JOB
          WITH CJ.JOB_CODE = FIRST J.JOB_CODE FROM J IN JOBS
          WITH J.JOB_TITLE = "Company President"
          SORTED BY CJ.JOB_CODE;
END_GET;

printf ("Last name is %s", name);

COMMIT;
FINISH;
}
```

FIRST FROM Value Expression

Pascal Program

```
program first_val (input,output);
DATABASE PERS = FILENAME 'PERSONNEL';

DECLARE_VARIABLE name SAME AS PERS.CURRENT_JOB.LAST_NAME;

begin
READY PERS;
START_TRANSACTION READ_ONLY;

GET
  name =  FIRST CJ.LAST_NAME FROM CJ IN CURRENT_JOB
          WITH CJ.JOB_CODE = FIRST J.JOB_CODE FROM J IN JOBS
          WITH J.JOB_TITLE = 'Company President'
          SORTED BY CJ.JOB_CODE;
END_GET;

writeln ('Last name is: ', name);

COMMIT;
FINISH;
end.
```

Example 3

The following programs demonstrate the use of the FIRST FROM value expression and the SORTED BY clause in a record selection expression. The programs sort (in alphabetical order) the records in the CURRENT_JOB view, based on the sort key DEPARTMENT_CODE. JOB_CODE is the second sort key, so RDML arranges (alphabetically) those records with different values for the JOB_CODE field that have the same value stored in the DEPARTMENT_CODE field. EMPLOYEE_ID is the third sort key, so RDML arranges (in ascending numerical order) those records with different values for the EMPLOYEE_ID field that have the same value stored in the JOB_CODE field.

The first print statement displays the EMPLOYEE_ID and the LAST_NAME fields of the sorted records. A GET statement retrieves records from a record stream created by joining the DEPARTMENTS relation with the CURRENT_ JOB view over the DEPARTMENT_CODE field. The FIRST statement selects the first record from the record stream in which the department code in the DEPARTMENTS relation is the same as the department code for a record in the sorted CURRENT_JOB view. The print statement displays the department name of this selected record.

FIRST FROM Value Expression

A third record stream is created by joining the JOBS relation with the CURRENT_JOB view over the JOB_CODE field. The FIRST FROM statement selects the first record from the JOBS relation in which the job code in the JOBS relation is the same as the job code for a record in the sorted CURRENT_ JOB view. The print statement displays the job title of this selected record.

C Program

```
#include <stdio.h>
DATABASE PERS = FILENAME "PERSONNEL";

DECLARE_VARIABLE name SAME AS PERS.DEPARTMENTS.DEPARTMENT_NAME;
DECLARE_VARIABLE title SAME AS PERS.JOBS.JOB_TITLE;

main()
{
READY PERS;
START_TRANSACTION READ_ONLY;

FOR CJ IN CURRENT_JOB
    SORTED BY CJ.DEPARTMENT_CODE, CJ.JOB_CODE, CJ.EMPLOYEE_ID
        printf ("%s  %s\n",CJ.EMPLOYEE_ID, CJ.LAST_NAME);

        GET
            name = FIRST D.DEPARTMENT_NAME FROM D IN DEPARTMENTS
                     WITH D.DEPARTMENT_CODE = CJ.DEPARTMENT_CODE;

            title = FIRST J.JOB_TITLE FROM J IN JOBS
                     WITH J.JOB_CODE = CJ.JOB_CODE;

        END_GET;

        printf ("Department name is: %s\n", name);
        printf ("Title is: %s\n\n", title);

END_FOR;

COMMIT;
FINISH;
}
```

Pascal Program

```
program first_comp (input,output);
DATABASE PERS = FILENAME 'PERSONNEL';

DECLARE_VARIABLE name SAME AS PERS.DEPARTMENTS.DEPARTMENT_NAME;
DECLARE_VARIABLE title SAME AS PERS.JOBS.JOB_TITLE;

begin
READY PERS;
START_TRANSACTION READ_ONLY;
```

FIRST FROM Value Expression

```
FOR CJ IN CURRENT_JOB
   SORTED BY CJ.DEPARTMENT_CODE, CJ.JOB_CODE, CJ.EMPLOYEE_ID
      writeln (CJ.EMPLOYEE_ID, ' ', CJ.LAST_NAME);

      GET
         name = FIRST D.DEPARTMENT_NAME FROM D IN DEPARTMENTS
                WITH D.DEPARTMENT_CODE = CJ.DEPARTMENT_CODE;

         title = FIRST J.JOB_TITLE FROM J IN JOBS
                 WITH J.JOB_CODE = CJ.JOB_CODE;
      END_GET;

      writeln ('Department name is: ', name);
      writeln ('Title is: ', title);
      writeln;

END_FOR;

COMMIT;
FINISH;
end.
```

2.4 Host Language Variable Value Expression

Use a host language variable value expression to pass data between a calling
program and Rdb.

Format

C-host-variable =

vax-name

```
        * 
              . ─────────► field-identifier
              [ ──► expression ──► ]
                         ,
              "->" ─────► field-identifier
```

Pascal-host-variable =

vax-name

```
              . ─────────► field-identifier
              [ ──► expression ──► ]
                         ,
              ^
                 field-identifier
```
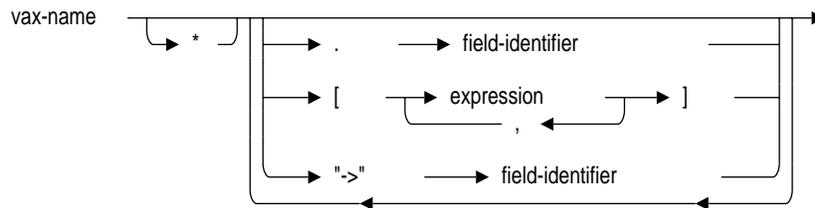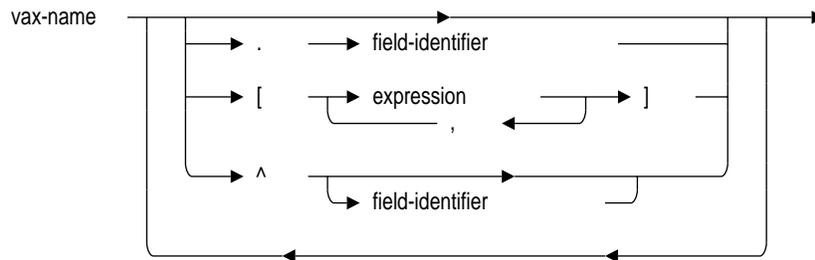
Arguments

vax-name
A valid VAX name.

field-identifier
A valid host language field identifier.

Host Language Variable Value Expression

expression
An expression that resolves to a valid host language array element in C or
Pascal. May include an RDML arithmetic operator. However, host language
operators, such as ++ and -- in C or DIV in Pascal are not supported.

"→"
The C pointer symbol. It is shown in quotes to distinguish it from the arrows
that show the logical flow of the syntax. Do not use quotes around the pointer
symbol in your program.

Usage Notes

- Host language variables can be:

  - Simple names, such as HEIGHT and NAME

  - Record fields, such as P1.TERMINAL

  - Pointers, such as PT^ and TREE^.NODENAME in Pascal, or
    TREE→NODENAME in C

  - Array elements, such as A[1] and B [I1, (I2-1)*2] in Pascal,
    B[I1][(I2-1)*2] in C

- You can use host language variables in record selection expressions.

- You can use host language variables as names to represent databases
  and database elements. These names are called handles. See Section 6.4,
  Section 6.20, and Section 6.27 for more information.

- You can declare a host language variable by referring to a database field
  with a DECLARE_VARIABLE clause. See Section 6.6 for details.

- When using C:

  - Be certain that text string variables are the same length as the text
    field in which you are storing them. Pad strings that are shorter than
    the text field with blank spaces; truncate strings that are longer than
    the text field.

  - Because the DECLARE_VARIABLE clause provides an extra character
    for null termination of character string variables, you may terminate
    text string variables with the null character in C programs. For
    example, if the field is defined as "DATATYPE IS TEXT SIZE IS 10",
    then the first ten characters of the text string variable must be valid
    data, and the eleventh may be the null character.

Host Language Variable Value Expression

    – General host language array elements such as [(int)(etype)] cannot be used in RSEs.

Examples

Example 1

The following programs demonstrate the use of a host language variable value expression. These programs declare a host language variable, *badge*, to hold the value of an employee ID. You enter the value of *badge* from the keyboard as the program runs. These programs declare *badge* using the DECLARE_VARIABLE clause. See Chapter 6 for more information on the DECLARE_VARIABLE clause.

Additionally, the C program declares and uses a function named read_string. This function causes the program to prompt for, and store, a value for *badge*. See Appendix B for the source code and more information on read_string. The Pascal readln and writeln statements perform a similar function.

The programs find the employee in the EMPLOYEES relation with an ID that is the same as the value of the host language variable. The MODIFY statement stores a new value for that employee's status code.

C Program

```
#include <stdio.h>
DATABASE PERS = FILENAME "PERSONNEL";

extern void read_string();
static DECLARE_VARIABLE badge SAME AS EMPLOYEES.EMPLOYEE_ID;

main()
{
read_string ("Employee ID: ", badge, sizeof(badge));

READY PERS;
START_TRANSACTION READ_WRITE;

FOR E IN EMPLOYEES WITH E.EMPLOYEE_ID = badge
   MODIFY E USING
      strcpy(E.STATUS_CODE,"1");
   END_MODIFY;
END_FOR;

ROLLBACK;
FINISH;
}
```

Host Language Variable Value Expression

Pascal Program

```
program modify_with_host (input,output);
DATABASE PERS = FILENAME 'PERSONNEL';

var
   DECLARE_VARIABLE badge SAME AS EMPLOYEES.EMPLOYEE_ID;

begin
write  ('Employee ID: ');
readln (badge);

READY PERS;
START_TRANSACTION READ_WRITE;

FOR E IN EMPLOYEES WITH E.EMPLOYEE_ID = badge
    MODIFY E USING
        E.STATUS_CODE := '1';
    END_MODIFY;
END_FOR;

ROLLBACK;
FINISH;
end.
```

Example 2

The following programs demonstrate the use of the host language variable value expression. As in Example 1, the programs declare host language variables with the DECLARE_VARIABLE clause and prompt for user input at run time.

The programs create a record stream that contains all the employee records in the EMPLOYEES relation with a status code equal to the value stored in the host language variable, *stat_code*. The programs print the employee ID, first name, and last name of these employees.

C Program

```
#include <stdio.h>
DATABASE PERS = FILENAME "PERSONNEL";

extern void read_string();
static DECLARE_VARIABLE stat_code SAME AS EMPLOYEES.STATUS_CODE;

main()
{
read_string("Status Code: ",stat_code,sizeof(stat_code));

READY PERS;
START_TRANSACTION READ_ONLY;
```

Host Language Variable Value Expression

```
FOR E IN EMPLOYEES WITH E.STATUS_CODE = stat_code
   printf ("%s %s %s\n\n",
             E.EMPLOYEE_ID,
             E.FIRST_NAME,
             E.LAST_NAME);
END_FOR;

COMMIT;
FINISH;
}
```

Pascal Program

```
program host_var (input,output);
DATABASE PERS = FILENAME 'PERSONNEL';

var
   DECLARE_VARIABLE stat_code SAME AS EMPLOYEES.STATUS_CODE;

begin

write  ('Status Code: ');
readln (stat_code);

READY PERS;
START_TRANSACTION READ_ONLY;

FOR E IN EMPLOYEES WITH E.STATUS_CODE = stat_code
   writeln (E.EMPLOYEE_ID, ' ', E.FIRST_NAME, ' ', E.LAST_NAME);
END_FOR;

COMMIT;
FINISH;
end.
```

Example 3

The following programs demonstrate the use of a host language variable value
expression as a transaction handle. See Section 6.27 for more information
on transaction handles. These programs declare the host language variable,
EMP_UPDATE. The programs use EMP_UPDATE to qualify the transaction
in the START_TRANSACTION statement, the record selection expression,
and the COMMIT statement. The record selection expression modifies the
record with the specified ID number in the EMPLOYEES relation. The
COMMIT statement, also qualified with the transaction handle, ensures that
the modified record is stored in the database.

The C program uses the function pad_string to append trailing blanks to
the LAST_NAME field. This ensures that the last name matches the length
defined for the field. For more information and the source code for pad_string,
see Appendix B.

Host Language Variable Value Expression

C Program

```
#include <stdio.h>
DATABASE PERS = FILENAME "PERSONNEL";

extern void pad_string();

main()
{
int EMP_UPDATE = 0;

READY PERS;
START_TRANSACTION (TRANSACTION_HANDLE EMP_UPDATE) READ_WRITE;

FOR (TRANSACTION_HANDLE EMP_UPDATE) E IN EMPLOYEES
   WITH E.EMPLOYEE_ID = "00178"
      MODIFY E USING
          pad_string("Brannon", E.LAST_NAME, sizeof(E.LAST_NAME));
      END_MODIFY;
END_FOR;

COMMIT(TRANSACTION_HANDLE EMP_UPDATE);
FINISH;
}
```

Pascal Program

```
program trhand (input,output);
DATABASE PERS = FILENAME 'PERSONNEL';

var EMP_UPDATE : [volatile] integer := 0;

begin

READY PERS;
START_TRANSACTION (TRANSACTION_HANDLE EMP_UPDATE) READ_WRITE;

FOR (TRANSACTION_HANDLE EMP_UPDATE) E IN EMPLOYEES
   WITH E.EMPLOYEE_ID = '00178'
      MODIFY E USING
          E.LAST_NAME := 'Brannon';
      END_MODIFY;
END_FOR;

COMMIT (TRANSACTION_HANDLE EMP_UPDATE);
FINISH;
end.
```

2.5 RDB$DB_KEY Value Expression

The RDB$DB_KEY (database key or dbkey) value expression lets you retrieve
a specific record from the database using an internal system pointer. The
database key is a logical pointer that indicates a specific record in the database.
You can retrieve this key as though it were a field in the record. Once you have
retrieved the database key, you can use it to retrieve its associated record
directly, as part of a record selection expression. The database key gives you
the ability to keep track of a subset of records in the database and retrieve
them directly.

Format

db-key =

$\longrightarrow$ context-var $\longrightarrow$. $\longrightarrow$ RDB$DB_KEY $\longrightarrow$

Argument

context-var
A context variable. A temporary name that you associate with a relation. You
define a context variable in a relation clause.

Usage Notes

- The database key reference must be within the scope of the context variable
  in the source code. RDML determines which relation the RDB$DB_KEY
  refers to from the context variable that you use.

- The scope of the database key can be either the COMMIT or FINISH
  statement. When the scope is COMMIT, the database key is valid for as
  long as the transaction in which it is retrieved is active. When the scope is
  FINISH, the database key is valid for the duration of the database attach
  in which it is retrieved. By default, the scope is COMMIT.

- You should use the RDB$DB_KEY value expression only if you have to
  repeatedly access the same records. For example, you may sort employees
  by seniority and use the database key for each employee as a way of
  moving back and forth within the list of sorted employees.

- In conjunction with a GET statement, you can retrieve the database key of a record being stored by using this expression as part of a STORE statement.

Examples

Example 1

The following programs demonstrate the use of the RDB$DB_KEY value expression in a record selection expression. The programs sort the EMPLOYEES relation in ascending order of employee ID. Then, using the first 100 records from the EMPLOYEES relation, the programs build two arrays: rdb_key_array and rdb_name_array. In building these arrays within a FOR statement, these programs create a one-to-one correspondence between the elements in the rdb_key_array and the rdb_name_array. Each time a new element is added to each of these arrays, the next EMPLOYEES record from the sorted stream is printed.

This one-to-one correspondence allows the programs to step through the EMPLOYEES records directly. This is demonstrated in the second FOR statement. The second FOR statement loops through the rdb_key_array in reverse order; each time the address of an array element in rdb_key_array is incremented, an EMPLOYEES record is accessed and printed, also in reverse order.

C Program

```
#include <stdio.h>
DATABASE PERS = FILENAME "PERSONNEL";

main()
{
DECLARE_VARIABLE rdb_key_array[100] SAME AS EMPLOYEES.RDB$DB_KEY;
DECLARE_VARIABLE rdb_name_array[100] SAME AS EMPLOYEES.LAST_NAME;

int cnt = 0;

READY PERS;

START_TRANSACTION READ_ONLY;

FOR FIRST 100 E IN EMPLOYEES SORTED BY E.EMPLOYEE_ID
    rdb_key_array[cnt] = E.RDB$DB_KEY;
    strcpy (rdb_name_array[cnt], E.LAST_NAME);
    printf("%s - 1st pass\n", E.LAST_NAME);
    ++cnt;
END_FOR;
```

RDB$DB_KEY Value Expression

```
for ( cnt = --cnt; cnt >= 0; --cnt)
    FOR E IN EMPLOYEES
    WITH E.RDB$DB_KEY = rdb_key_array[cnt]
        if ( strcmp( E.LAST_NAME, rdb_name_array[cnt]) != 0 )
            printf("%s DOES NOT MATCH %s\n",
                        E.LAST_NAME,  rdb_name_array[cnt]);
        else printf("%s - 2nd pass\n", E.LAST_NAME);
    END_FOR;

COMMIT;
FINISH;
}
```

Pascal Program

```
program db_key (input,output);
DATABASE PERS = FILENAME 'PERSONNEL';

type
    Rdb_Key_Type = BASED ON EMPLOYEES.RDB$DB_KEY;
    Rdb_Name_Type =  BASED ON EMPLOYEES.LAST_NAME;
var
    Rdb_Key_Array  : ARRAY [1..101] OF Rdb_Key_Type;
    Rdb_Name_Array : ARRAY [1..101] OF Rdb_Name_Type;
    Cnt : INTEGER := 1;

begin
READY PERS;
START_TRANSACTION READ_ONLY;

FOR FIRST 100 E IN EMPLOYEES SORTED BY E.EMPLOYEE_ID
    Rdb_Key_Array[Cnt] := E.RDB$DB_KEY;
    Rdb_Name_Array[Cnt] := E.LAST_NAME;
    WRITELN(E.LAST_NAME, ' - 1st pass');
    Cnt := Cnt + 1;
END_FOR;

for Cnt := Cnt - 1 downto 1 do
    FOR E IN EMPLOYEES
    WITH E.RDB$DB_KEY = Rdb_Key_array[Cnt]
        if  E.LAST_NAME <> Rdb_Name_Array[Cnt]
        then
            writeln (E.LAST_NAME, 'DOES NOT MATCH',
                Rdb_Name_Array[Cnt])
        else
     writeln (E.LAST_NAME, ' - 2nd pass');
    END_FOR;

COMMIT;
FINISH;
end.
```

2.6 RDB$MISSING Value Expression

The RDB$MISSING value substitutes the missing value (if one was defined) for a specified database field.

To use RDB$MISSING, you must have previously defined a missing value for the field when you defined the database. If a field is left blank, or you use RDB$MISSING without having defined a missing value for that field in its field definition, RDML issues an error.

For information on how to define a missing value for a field, see the documentation for your database system. If you are using Rdb/VMS, see the *VAX Rdb/VMS Guide to Database Design and Definition*. If you are using Rdb/ELN, see the Define Field section in the *VAX Rdb/ELN Reference Manual*.

Format

missing-value =

```
        ──────▶  RDB$MISSING  ──────────────────────▶──────────────────────╮
                                                                            │
        ╰──────▶  (  ──┬──▶  context-var . field-name  ──────────────┬──▶ )  ──▶
                       ├──▶  relation-name . field-name  ────────────┤
                       ╰──▶  db-handle . relation-name . field-name ─╯
```

Arguments

context-var
A context variable. A temporary name that you associate with a relation. You define a context variable in a relation clause.

field-name
The name of a field in a relation.

relation-name
The name of a relation in a database.

db-handle
A database handle. A host language variable that identifies a database.

RDB$MISSING Value Expression

Usage Notes

- There is no default missing value.

- Use the RDB$MISSING value expression as though it is a constant in the host language.

- Do not use the RDB$MISSING expression to test for the presence of values. Rather, you should use the MISSING conditional expression.

- During a STORE operation, instead of using RDB$MISSING to mark a field as empty, you can simply exclude this field from the STORE statement. When you retrieve the record that contains this field, the missing value associated with the field will be returned. However, you cannot use this method, nor RDB$MISSING, if the field has the validation clause "VALID IF NOT MISSING".

- The value of RDB$MISSING is set at preprocessing time. If you redefine the missing value for a field and do not preprocess the program with the RDB$MISSING value expression, your program actually stores the old value rather than marking the field as empty. Note that the MISSING conditional expression checks the missing value for a field at run time.

Examples

Example 1

The following programs demonstrate the use of the RDB$MISSING value expression with the STORE statement. The programs store the specified values for the fields in the DEGREES relation. In these programs, a value for DEGREE_FIELD is not specified; instead, the RDB$MISSING value expression is specified. This does not actually assign a value to the degree field; RDML marks the DEGREE_FIELD as empty and stores nothing in this field.

The C program uses the function pad_string to append trailing blanks to the strings before they are stored. This ensures that the strings match the length defined for the fields. For more information and the source code for pad_string, see Appendix B.

C Program

```
#include <stdio.h>
DATABASE PERS = FILENAME "PERSONNEL";

extern void pad_string();

main()
{
READY PERS;
START_TRANSACTION READ_WRITE;

STORE D IN DEGREES USING
    pad_string ("76156", D.EMPLOYEE_ID, sizeof(D.EMPLOYEE_ID));
    pad_string ("HVDU" , D.COLLEGE_CODE, sizeof(D.COLLEGE_CODE));
    D.YEAR_GIVEN = 1978;
    pad_string ("BA", D.DEGREE, sizeof(D.DEGREE));
    pad_string (RDB$MISSING(D.DEGREE_FIELD),D.DEGREE_FIELD,
                sizeof(D.DEGREE_FIELD));
END_STORE;

ROLLBACK;
FINISH;
}
```

Pascal Program

```
program store_missing (input,output);
DATABASE PERS = FILENAME 'PERSONNEL';

begin
READY PERS;
START_TRANSACTION READ_WRITE;

STORE D IN DEGREES USING
    D.EMPLOYEE_ID := '76156';
    D.COLLEGE_CODE := 'HVDU';
    D.YEAR_GIVEN := 1978;
    D.DEGREE := 'BA';
    D.DEGREE_FIELD := RDB$MISSING(D.DEGREE_FIELD);
END_STORE;

ROLLBACK;
FINISH;
end.
```

Example 2

The following programs demonstrate the use of the RDB$MISSING value expression with the MODIFY statement and the COUNT statistical expression. The programs print an introductory statement before attaching to the database.

RDB$MISSING Value Expression

The record selection expression crosses the SALARY_HISTORY and EMPLOYEES relations over the common EMPLOYEE_ID field. The COUNT function limits the record stream to those records in the EMPLOYEES relation with five or more corresponding records in the SALARY_HISTORY relation. The programs print the last name of the employees in this record stream.

Using the MODIFY statement, the programs mark the STATUS_CODE field as empty for the employees in the record stream (no value is stored in the field). However, the ROLLBACK statement undoes all changes to the database, and all the fields remain as they were before the program began.

C Program

```
#include <stdio.h>
DATABASE PERS = FILENAME "PERSONNEL";

main()
{
READY PERS;
START_TRANSACTION READ_WRITE;

printf ("Impose early retirement on all employees with\n");
printf ("5 or more salary history records\n");

FOR E IN EMPLOYEES
   WITH (COUNT OF SH IN SALARY_HISTORY
   WITH SH.EMPLOYEE_ID = E.EMPLOYEE_ID >= 5)
      printf ("%s is being forced to retire early\n", E.LAST_NAME);
         MODIFY E USING
            strncpy( E.STATUS_CODE, RDB$MISSING (E.STATUS_CODE), 1);
         END_MODIFY;
END_FOR;

printf ("Only fooling...Let's rollback and forget it.\n");

ROLLBACK;
FINISH;
}
```

Pascal Program

```
program missing_with_modify (input,output);
DATABASE PERS = FILENAME 'PERSONNEL';

begin

writeln ('Impose early retirement on all employees with ');
writeln ('5 or more salary history records');

READY PERS;
START_TRANSACTION READ_WRITE;
```

RDB$MISSING Value Expression

```
FOR E IN EMPLOYEES
   WITH (COUNT OF SH IN SALARY_HISTORY
   WITH SH.EMPLOYEE_ID = E.EMPLOYEE_ID >= 5)
      writeln (E.LAST_NAME, ' is being forced to retire early');
         MODIFY E USING
            E.STATUS_CODE  := RDB$MISSING (E.STATUS_CODE);
         END_MODIFY;
END_FOR;

writeln ('Only fooling...Let''s rollback and forget it.');

ROLLBACK;
FINISH;
end.
```

3

RDML Conditional Expressions

This chapter describes the Relational Data Manipulation Language (RDML) conditional expressions that can be used with embedded RDML statements in C and Pascal programs.

The C and Pascal programs in this chapter access the sample personnel database available with Rdb/VMS.

A **conditional expression**, sometimes called a Boolean expression, represents the relationship between two value expressions. Conditional expressions are used in the WITH clause of the record selection expression.

The value of a conditional expression is true, false, or missing. If there is no value stored in a field of a record, then the relationship of that field to others is unknown. Therefore, the results of comparisons that use that field are considered missing.

A missing value for a field in a relation has no value associated with it. The missing value is an attribute of a field rather than a value stored in a field.

The three types of conditional expressions are:

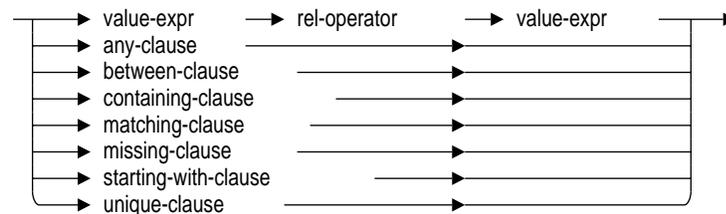- Those that express a relationship between two value expressions, using a relational operator

  For example, the expression SH.SALARY_AMOUNT > 50000 is true if the value in the SALARY_AMOUNT field of the SALARY_HISTORY record is greater than 50,000. When Rdb evaluates this expression, it examines the relationship between the two value expressions, SH.SALARY_AMOUNT and 50,000. If the value in the SALARY_AMOUNT field of a record is MISSING, then that record is not included in the record stream.

- Those that express a characteristic of a single value expression

  For example, E.STATE MISSING is true if there is no value in the STATE field of an EMPLOYEES record.

- Those that express a relationship among three value expressions

  For example, E.MIDDLE_INITIAL BETWEEN "A" AND "N".

Format

conditional-expr =

rel-operator =

```
          ┌──────→  EQ  ──────┐
          ├──────→  =   ──────┤
          ├──────→  NE  ──────┤
          ├──────→  <>  ──────┤
──────────┼──────→  GT  ──────┼──────────→
          ├──────→  >   ──────┤
          ├──────→  GE  ──────┤
          ├──────→  >=  ──────┤
          ├──────→  LT  ──────┤
          ├──────→  <   ──────┤
          ├──────→  LE  ──────┤
          └──────→  <=  ──────┘
```
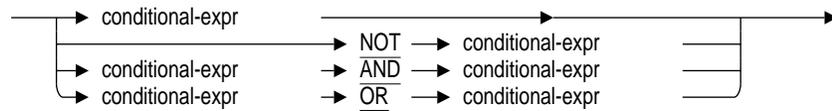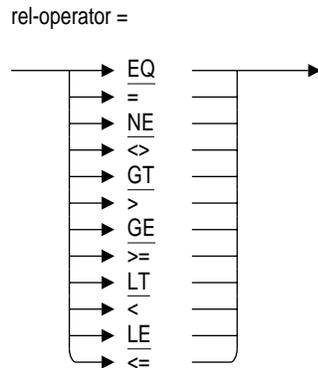
Arguments

NOT
AND
OR
Logical operators that combine conditional expressions. The result of such a combination is also a conditional expression.

value-expr
A value expression. A symbol or a string of symbols used to calculate a value. When you use a value expression in a statement, Rdb calculates the value associated with the expression and uses that value when executing the statement. See Chapter 2 for more information.

rel-operator
A relational operator. Controls the comparison of value expressions. In all cases, if either operand in a relational expression is missing, the value of the condition is missing.

Usage Notes

- Rdb compares character string literals according to the ASCII collating sequence. Rdb considers lowercase letters to have a greater value than uppercase letters and the letters near the beginning of the alphabet to have a lesser value than those near the end.

  "a" > "A"

"b" > "Z"

"a" < "z"

"A" < "Z"

- The RDML preprocessor evaluates conditional expressions in the following order:

    NOT
    AND
    OR

    You can use parentheses to alter this default order of evaluation.

- Table 3–1 is a truth table for complex conditional expressions that use logical operators. For example, if conditional expression A is true and B is missing, then "A AND B" is evaluated as missing.

Table 3–1   Conditional Expression Truth Table

Values of A and B		NOT Condition	AND Condition	OR Condition
A	B	NOT A	A AND B	A OR B
True	True	False	True	True
True	False	False	False	True
True	Missing	False	Missing	True
False	True	True	False	True
False	False	True	False	False
False	Missing	True	False	Missing
Missing	Missing	Missing	Missing	Missing

Table 3–2 describes the function of each type of conditional expression.

Table 3–2   Values Returned by Conditional Expressions

Conditional Expression	Values
ANY	True if the record stream specified by the record selection expression (RSE) includes at least one record.
BETWEEN	True if the first value expression is equal to or between the second and third value expressions.
CONTAINING	True if the string specified by the second string expression is found within the string specified by the first. Case insensitive.
MATCHING	True if the second expression matches a substring of the first value expression. MATCHING allows you to use the asterisk (*) to specify a string of any characters, and the percent character (%) to specify a single character. Case insensitive.
MISSING	True if the specified value expression is missing.
Relational operator	True if the first and second value expressions are found in the relationship specified by the relational operator.
STARTING WITH	True if the characters of the first string expression match the second string expression. Case sensitive.
UNIQUE	True if the record stream specified by the record selection expression (RSE) consists of exactly one record.

Examples

Example 1

The following programs demonstrate the use of a FOR loop with a conditional expression. The conditional expression limits the records contained in the record stream, and compares the SALARY_AMOUNT field name to the host language variable (limit).

Figure 3–1   Conditional Expression Component of an RSE

FOR SH IN SALARY_HISTORY WITH SH.SALARY_AMOUNT GT LIMIT

conditional expression

record selection expression
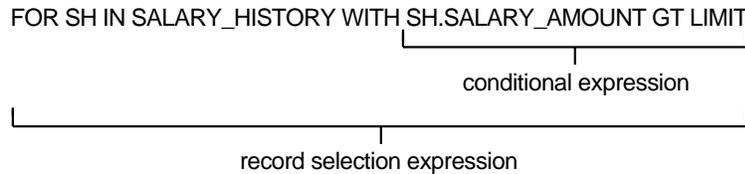
ZK–7549–GE

The record stream consists of all records in which the result of the comparison is true. Figure 3–1 shows the relationship of the conditional expression to the record selection expression.

Notice that the host language variable in these programs receives its value at run time through interactive processing. The C program uses the function, read_float, to receive and store the value for the host language variable. See Appendix B for the source code and details on using this function. The Pascal program uses the writeln and readln statements to produce similar results.

C Program

```
#include <stdio.h>
DATABASE PERS = FILENAME "PERSONNEL";

extern float read_float();
static DECLARE_VARIABLE limit SAME AS SALARY_HISTORY.SALARY_AMOUNT;

main()
{
limit = read_float("Salary limit: ");

READY PERS;
START_TRANSACTION READ_ONLY;

FOR SH IN SALARY_HISTORY WITH SH.SALARY_AMOUNT GT limit
   printf ("$%f\n", SH.SALARY_AMOUNT);
END_FOR;

COMMIT;
FINISH;
}
```

Pascal Program

```
program cond_exp (input,output);
DATABASE PERS = FILENAME 'PERSONNEL';

var
   DECLARE_VARIABLE limit SAME AS SALARY_HISTORY.SALARY_AMOUNT;

begin

write  ('Salary limit: ');
readln (limit);

READY PERS;
START_TRANSACTION READ_ONLY;

FOR SH IN SALARY_HISTORY WITH SH.SALARY_AMOUNT GT limit
   writeln ('$', SH.SALARY_AMOUNT:10:2);
END_FOR;

COMMIT;
FINISH;
end.
```

Example 2

The following programs combine several conditional expressions using the
AND, NOT, and CONTAINING expressions. If, for a given record, the first,
second, and third conditions are all true, that record becomes part of the record
stream defined by the FOR statement. The programs print the names of the
colleges that meet the specified conditions.

C Program

```
#include <stdio.h>
DATABASE PERS = FILENAME "PERSONNEL";

main()
{
READY PERS;
START_TRANSACTION READ_ONLY;

FOR C IN COLLEGES
   WITH C.COLLEGE_NAME NOT CONTAINING "UNIV"
   AND C.COLLEGE_NAME NOT CONTAINING "COLLEGE"
   AND C.COLLEGE_NAME NOT CONTAINING "ACADEMY"
      printf ("%s\n", C.COLLEGE_NAME);
END_FOR;

COMMIT;
FINISH;
}
```

Pascal Program

```
program cond_and_bool (input,output);
DATABASE PERS = FILENAME 'PERSONNEL';

begin
READY PERS;
START_TRANSACTION READ_ONLY;

FOR C IN COLLEGES
   WITH C.COLLEGE_NAME NOT CONTAINING 'UNIV'
   AND C.COLLEGE_NAME NOT CONTAINING 'COLLEGE'
   AND C.COLLEGE_NAME NOT CONTAINING 'ACADEMY'
      writeln (C.COLLEGE_NAME);
END_FOR;

COMMIT;
FINISH;
end.
```

3.1 ANY Conditional Expression

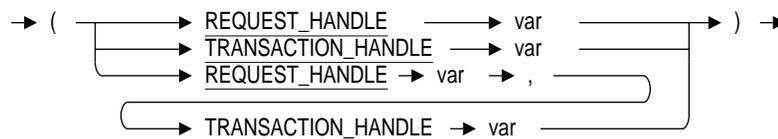The ANY conditional expression tests for the presence of any record in a record stream.

An ANY conditional expression is true if the record stream specified by the record selection expression includes at least one record. If you precede the ANY expression with the optional NOT qualifier, the condition is true if no records are in the record stream.

Format

any-clause =

NOT ANY handle-options

rse

handle-options =

(REQUEST_HANDLE var)
TRANSACTION_HANDLE var
REQUEST_HANDLE var ,
TRANSACTION_HANDLE var
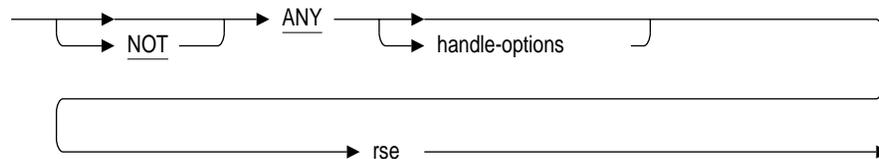
Arguments

handle-options
A request handle, transaction handle, or both.

REQUEST_HANDLE var
A REQUEST_HANDLE keyword followed by a host language variable. A request handle identifies a compiled Rdb/VMS request. If you do not supply a request handle explicitly, RDML generates a unique request handle for the compiled request. See Section 6.20 for more information on request handles.

ANY Conditional Expression

TRANSACTION_HANDLE var
A TRANSACTION_HANDLE keyword followed by a host language variable. A transaction handle identifies a transaction. If you do not supply a transaction handle explicitly, RDML uses the default transaction handle.

rse
A record selection expression. A clause that defines specific conditions that individual records must meet before Rdb includes them in a record stream. See Chapter 4 for more information.

Examples

Example 1

The following programs demonstrate the use of the NOT ANY conditional expression. The programs join the EMPLOYEES and DEGREES relations over their common EMPLOYEE_ID field. The NOT ANY expression finds those employees who do not have an employee ID stored in a DEGREES record (and therefore, either do not have a degree or this information has not been added to the database). Then the programs print the last names of those employees.

C Program

```
#include <stdio.h>
DATABASE PERS = FILENAME "PERSONNEL";

main()
{
READY PERS;
START_TRANSACTION READ_ONLY;

FOR E IN EMPLOYEES
   WITH NOT ANY D IN DEGREES WITH D.EMPLOYEE_ID = E.EMPLOYEE_ID
      printf ("%s \n",E.LAST_NAME);
END_FOR;

COMMIT;
FINISH;
}
```

Pascal Program

```
program any_with_not (input,output);
DATABASE PERS = FILENAME 'PERSONNEL';

begin
READY PERS;
START_TRANSACTION READ_ONLY;
```

```
FOR E IN EMPLOYEES
   WITH NOT ANY D IN DEGREES WITH D.EMPLOYEE_ID = E.EMPLOYEE_ID
      writeln (E.LAST_NAME);
END_FOR;

COMMIT;
FINISH;
end.
```

Example 2

The following programs demonstrate the use of the ANY conditional expression. The programs create a record stream that contains all the records from the SALARY_HISTORY relation that hold a value greater than 50,000 in the SALARY_AMOUNT field. The informational message "Someone is not underpaid" is printed if one or more records are found that meet the previously stated condition. Note that the print statements in these examples do not have access to the context variable created in the GET statement.

C Program

```
#include <stdio.h>

DATABASE PERS = FILENAME "PERSONNEL";

int who;

main()
{
READY PERS;
START_TRANSACTION READ_ONLY;

GET
 who = ANY SH IN SALARY_HISTORY WITH SH.SALARY_AMOUNT > 50000.00;
END_GET;

COMMIT;

if (who)
    printf ("Someone is not underpaid \n");

FINISH;
}
```

ANY Conditional Expression

Pascal Program

```
program anycond (input,output);
DATABASE PERS = FILENAME 'PERSONNEL';

who : boolean;

begin
READY PERS;
START_TRANSACTION READ_WRITE;

GET
 who = ANY SH IN SALARY_HISTORY WITH SH.SALARY_AMOUNT > 50000.00
END_GET;

COMMIT;

if (who) then
   writeln ('Someone is not underpaid.');

FINISH;
end.
```

3.2 BETWEEN Conditional Expression

The BETWEEN conditional expression creates a record stream that contains records with values that fall within a range you specify.

This expression is true if the first value expression is equal to or between the second and third value expressions. If you precede the BETWEEN expression with the optional NOT qualifier, the condition is true if no records are within the range you specify in the second and third value expressions.

Format

between-clause =
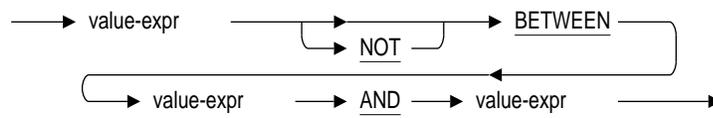
Argument

value-expr
A value expression. A symbol or a string of symbols used to calculate a value. When you use a value expression in a statement, Rdb calculates the value associated with the expression and uses that value when executing the statement. See Chapter 2 for more information.

Usage Notes

- Value expressions that are string literals must be enclosed in quotation marks. Use double quotation marks (" ") in C programs. Use single quotation marks (ʼ ʼ) in Pascal programs.

- Value expressions that are numeric literals must not be enclosed in quotation marks.

- Dates are stored in the database in an encoded binary format. Therefore, when using the BETWEEN conditional expression with dates, your application must first convert the dates to a binary format. See Section 4.1 for an example of a date conversion.

BETWEEN Conditional Expression

Examples

Example 1

The following programs demonstrate the use of the BETWEEN conditional expression with a numeric field. These programs form a record stream that consists of all the records in the CURRENT_SALARY relation where the SALARY_AMOUNT field contains a value greater than or equal to 10,000 and less than or equal to 20,000. These programs print the last name and salary from each of the records in the record stream.

C Program

```
#include <stdio.h>
DATABASE PERS = FILENAME "PERSONNEL";

main()
{
READY PERS;
START_TRANSACTION READ_WRITE;

FOR CS IN CURRENT_SALARY
   WITH CS.SALARY_AMOUNT
   BETWEEN 10000.00 AND 20000.00
      printf ("%s %f\n", CS.LAST_NAME, CS.SALARY_AMOUNT);
END_FOR;

COMMIT;
FINISH;
}
```

Pascal Program

```
program between_numeric (input,output);
DATABASE PERS = FILENAME 'PERSONNEL';

begin
READY PERS;
START_TRANSACTION READ_ONLY;

FOR CS IN CURRENT_SALARY
   WITH CS.SALARY_AMOUNT
   BETWEEN 10000.00 AND 20000.00
      writeln (CS.LAST_NAME, CS.SALARY_AMOUNT :10:2);
END_FOR;

COMMIT;
FINISH;
end.
```

BETWEEN Conditional Expression

Example 2

The following programs demonstrate the use of the BETWEEN conditional expression with a text string field. The programs form a record stream that consists of all the records in the EMPLOYEES relation where the LAST_ NAME field begins with any letter between "A" and "M". Note that any last name that begins with an "M" is not within this range (unless the entire last name is "M"). The programs then print the last name contained in each record in the record stream.

C Program

```
#include <stdio.h>
DATABASE PERS = FILENAME "PERSONNEL";

main()
{
READY PERS;
START_TRANSACTION READ_WRITE;

FOR E IN EMPLOYEES
   WITH E.LAST_NAME BETWEEN "A" AND "M"
      printf ("%s\n", E.LAST_NAME);
END_FOR

COMMIT;
FINISH;
}
```

Pascal Program

```
program between_alphabetic (input,output);
DATABASE PERS = FILENAME 'PERSONNEL';

begin
READY PERS;
START_TRANSACTION READ_ONLY;

FOR E IN EMPLOYEES
   WITH E.LAST_NAME BETWEEN "A" AND "M"
       writeln (E.LAST_NAME);
END_FOR;

COMMIT;
FINISH;
end.
```

3.3 CONTAINING Conditional Expression

The CONTAINING conditional expression tests for the presence of a specified string anywhere inside a string expression.

This expression is true if the string specified by the second (pattern) string expression is found within the string specified by the first (target) string expression. If either of the string expressions in a CONTAINING conditional expression is a missing value, the result is the missing value.

If you precede CONTAINING with the optional NOT qualifier, the condition is true if no records contain the specified string.

Note *The CONTAINING conditional expression is not case sensitive; it considers uppercase and lowercase forms of the same character to be a match.*

Format

containing-clause =

$$\longrightarrow \text{value-expr} \longrightarrow \boxed{\text{NOT}} \longrightarrow \underline{\text{CONTAINING}} \longrightarrow \text{value-expr} \longrightarrow$$
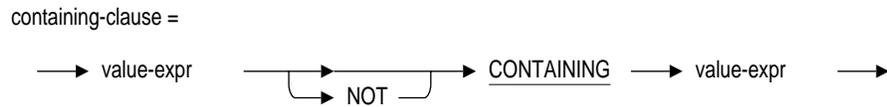
Argument

value-expr
A value expression. A symbol or a string of symbols used to calculate a value. When you use a value expression in a statement, Rdb calculates the value associated with the expression and uses that value when executing the statement. With the CONTAINING conditional expression, Rdb searches the first value expression for the presence of the second value expression. The second value expression is a literal.

CONTAINING Conditional Expression

Usage Notes

- Dates are stored in the database in an encoded binary format. Therefore, when using the CONTAINING conditional expression with dates, your program must first convert the dates to a binary format. See Section 4.1 for an example of a date conversion.

- The CONTAINING conditional expression will not execute properly in RDML/Pascal when you use a host language variable of data type PACKED ARRAY for comparison in this expression. For example, in the following code fragment host-var is the comparison value.

```
FOR E IN EMPLOYEES
   E.LAST_NAME CONTAINING host-var
      .
      .
      .
END_FOR;
```

   Note that a PACKED ARRAY data type is generated by the DECLARE_VARIABLE, DEFINE_TYPE, and BASED_ON clauses for field values of data type TEXT.

   Therefore, when you declare a host language variable in an RDML/Pascal program as the comparison value in a CONTAINING conditional expression, you should declare a variable of data type VARYING STRING. Do not use the DECLARE_VARIABLE, DEFINE_TYPE, or BASED_ON clause to declare this variable.

- The CONTAINING conditional expression is not sensitive to diacritical markings nor is it case sensitive. Thus "a" matches "A", "á", "à", "ä", "Á", "À", "Â", and so on. (Note that in Norwegian, "ä" is treated as if it were "ae".)

   In Spanish, the combinations "ch" and "ll" are treated as if they were individual unique single letters. Thus, CONTAINING "C" will find "C", "c", "ç", and "Ç", but not "CH", "ch", "Ch" and "cH".

CONTAINING Conditional Expression

Examples

Example 1

The following programs demonstrate the use of the CONTAINING conditional expression. The programs create a record stream that contains all the records in the EMPLOYEES relation in which the LAST_NAME field contains the string "IACO" (in upper or lower case letters). The programs print the employee ID and last name from each record contained in the record stream.

C Program

```
#include <stdio.h>
DATABASE PERS = FILENAME "PERSONNEL";

main()
{
READY PERS;
START_TRANSACTION READ_ONLY;

FOR E IN EMPLOYEES
   WITH E.LAST_NAME CONTAINING "IACO"
      printf ("%s  %s\n", E.EMPLOYEE_ID,
                          E.LAST_NAME);
END_FOR;

COMMIT;
FINISH;
}
```

Pascal Program

```
program containing (input,output);
DATABASE PERS = FILENAME 'PERSONNEL';

begin
READY PERS;
START_TRANSACTION READ_ONLY;

FOR E IN EMPLOYEES
   WITH E.LAST_NAME CONTAINING 'IACO'
      writeln (E.EMPLOYEE_ID, ' ', E.LAST_NAME);
END_FOR;

COMMIT;
FINISH;
end.
```

Example 2

The following programs demonstrate the use of the NOT CONTAINING conditional expression. The programs declare two host language variables, name1 and name2, to hold values to use in the CONTAINING conditional expression. The programs then create a record stream that contains all the records in the COLLEGES relation where the COLLEGE_NAME field contains neither the string "univ" nor the string "college" (in uppercase or lowercase). The programs then print the college name from each record contained in the record stream.

C Program

```
#include <stdio.h>
DATABASE PERS = FILENAME "PERSONNEL";

DECLARE_VARIABLE name1, name2 SAME AS COLLEGES.COLLEGE_NAME;

main()
{
strcpy(name1,"univ");
strcpy(name2, "college");

READY PERS;
START_TRANSACTION READ_ONLY;

FOR C IN COLLEGES
   WITH C.COLLEGE_NAME NOT CONTAINING name1
   AND C.COLLEGE_NAME NOT CONTAINING name2
      printf("%s\n",C.COLLEGE_NAME);
END_FOR;

COMMIT;
FINISH;
}
```

Pascal Program

```
program not_contain (input,output);
DATABASE PERS = FILENAME 'PERSONNEL';

var
 name1 : VARYING [10] OF CHAR;
 name2 : VARYING [10] OF CHAR;
begin

name1 := 'univ';
name2 := 'college';

READY PERS;
START_TRANSACTION READ_ONLY;
```

CONTAINING Conditional Expression

```
FOR C IN COLLEGES
   WITH C.COLLEGE_NAME NOT CONTAINING name1
   AND C.COLLEGE_NAME NOT CONTAINING name2
      writeln (C.COLLEGE_NAME);
END_FOR;

COMMIT;
FINISH;
end.
```

3.4 MATCHING Conditional Expression

The MATCHING conditional expression lets you use the asterisk (*) pattern matching character in combination with other characters to test for the presence of a specified string anywhere inside a string expression.

This expression is true if the string specified by the second (pattern) string expression is found within the string specified by the first (target) string expression. If either of the string expressions in a MATCHING conditional expression is missing, the result is missing.

If you precede MATCHING with the optional NOT qualifier, the condition is true if the pattern string is not found within the string specified by the target string.

Note *The MATCHING conditional expression is not case sensitive; it considers uppercase and lowercase forms of the same character to be a match.*

Format

matching-clause =

⟶ value-expr ──────┬──────────┬─→ MATCHING ⟶ match-expr ⟶

        └─→ NOT ─┘
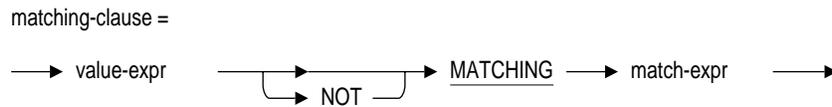
Arguments

value-expr
A value expression. A symbol or a string of symbols used to calculate a value. When you use a value expression in a statement, Rdb calculates the value associated with the expression and uses that value when executing the statement. When you use the MATCHING conditional expression, Rdb searches the first value expression to see if it starts with the characters specified in the second value expression. The second value expression is a string literal. See Chapter 2 for more information on value expressions.

MATCHING Conditional Expression

match-expr

A match expression. An unquoted host language variable or an expression in quotation marks that is used to match a pattern. Use double quotation marks (`" "`) in C programs. Use single quotation marks (`' '`) in Pascal programs. The match expression can include the following special symbols (called wildcards):

- * Matches a string of zero or more characters

- % Matches a single character

Usage Notes

- The MATCHING conditional expression will not execute properly in RDML/Pascal when you use a host language variable of data type PACKED ARRAY for comparison in this expression. For example, in the following code fragment, host-var is the comparison value.

```
FOR E IN EMPLOYEES
    E.LAST_NAME MATCHING host-var
        .
        .
        .
END_FOR;
```

  Note that a PACKED ARRAY data type is generated by the DECLARE_ VARIABLE, DEFINE_TYPE, and BASED_ON clauses for field values of data type TEXT.

  Therefore, when you declare a host language variable in an RDML/Pascal program as the comparison value in a MATCHING conditional expression, you should declare a variable of data type VARYING STRING. Do not use the DECLARE_VARIABLE, DEFINE_TYPE, or BASED_ON clause to declare this variable.

- You can use any combination of wildcards in a matching expression; however, if you choose not to use *any* wildcards in a matching expression; the expression must match the value stored in the database *exactly*. For example, using the PERSONNEL database, if you want to find all the employees with the last name Smith and do not want to use wildcards, you must append nine blank spaces to the name Smith. This is because the LAST_NAME field is defined as TEXT 14 in the PERSONNEL database. If LAST_NAME were defined as TEXT 5 you would not need to append blank spaces to the name.

```
FOR E IN EMPLOYEES
   WITH E.LAST_NAME MATCHING "Smith            "
      .
      .
      .
END_FOR;
```

Digital recommends that you use the relational operator equals ($=$) instead of the MATCHING conditional expression if you do not need to use wildcards. The equals operator ignores trailing blanks. For example, the following record selection expression will retrieve all the records in the EMPLOYEES relation with the value Smith in the LAST_NAME field:

```
FOR E IN EMPLOYEES
   WITH E.LAST_NAME = "Smith"
      .
      .
      .
END_FOR;
```

If you used the MATCHING conditional expression instead of the equals operator in the previous code fragment, MATCHING would only retrieve employees with the last name of "Smith" if the definition for LAST_NAME was TEXT 5. If the definition is TEXT 10, the MATCHING conditional expression would retrieve all records with the name "Smith" only if you appended five trailing blanks to the name "Smith   ".

- The MATCHING conditional expression is not sensitive to diacritical markings nor is it case sensitive. Thus, "a" matches "A", "á", "à", "ä", "Á", "À", "Â", and so on. (Note that in Norwegian, "ä" is treated as if it were "ae".)

  In Spanish, the combinations "ch" and "ll" are each treated as individual unique single letters. If you define your collating sequence as SPANISH, the percent sign (%) matches any single letter, including "ch" and "ll". "C%" and "C*" do not match "CH", "ch", "Ch", or "cH".

Examples

Example 1

The following programs demonstrate the use of the MATCHING conditional expression and the SORTED BY clause. The programs declare a host language variable, match-string, to use in the MATCHING condition expression. Then the programs create a record stream that contains all the records in the EMPLOYEES relation in which the LAST_NAME field begins with the letter

MATCHING Conditional Expression

"R" (as specified in the host language variable). Next, the programs sort the record stream in ascending numerical order of the employee IDs. The programs print, in numerical order, the employee ID, followed by the last name and first name for each record in the record stream.

C Program

```
#include <stdio.h>
DATABASE PERS = FILENAME "PERSONNEL";

DECLARE_VARIABLE match_string SAME AS EMPLOYEES.LAST_NAME;

main()
{
strcpy(match_string,"R*");

READY PERS;
START_TRANSACTION READ_ONLY;

FOR E IN EMPLOYEES
   WITH E.LAST_NAME MATCHING match_string
   SORTED BY E.EMPLOYEE_ID
      printf ("%s  %s  %s",E.EMPLOYEE_ID,
                           E.LAST_NAME,
                           E.FIRST_NAME);
END_FOR;

COMMIT;
FINISH;
}
```

Pascal Program

```
program matching (input,output);
DATABASE PERS = FILENAME 'PERSONNEL';

var
  match_string: VARYING [10] OF CHAR;
begin

match_string := 'R*';

READY PERS;
START_TRANSACTION READ_ONLY;

FOR E IN EMPLOYEES
   WITH E.LAST_NAME MATCHING match_string
   SORTED BY E.EMPLOYEE_ID
      writeln (E.EMPLOYEE_ID,'     ', E.LAST_NAME, E.FIRST_NAME);
END_FOR;

COMMIT;
FINISH;
end.
```

MATCHING Conditional Expression

Example 2

The following programs demonstrate the use of the MATCHING conditional expression and the SORTED BY clause. The programs create a record stream that contains all the records in the EMPLOYEES relation in which the LAST_ NAME field has the string "on" anywhere within the last name. The record stream is sorted in ascending alphabetical order and the programs print the first five records from the sorted stream.

C Program

```
#include <stdio.h>
DATABASE PERS = FILENAME "PERSONNEL";

main()
{
READY PERS;
START_TRANSACTION READ_ONLY;

FOR FIRST 5 E IN EMPLOYEES
   WITH E.LAST_NAME MATCHING "*on*"
   SORTED BY E.LAST_NAME
      printf ("%s\n",E.LAST_NAME);
END_FOR;

COMMIT;
FINISH;
}
```

Pascal Program

```
program matching (input,output);
DATABASE PERS = FILENAME 'PERSONNEL';

begin
READY PERS;
START_TRANSACTION READ_ONLY;

FOR FIRST 5 E IN EMPLOYEES
    WITH E.LAST_NAME MATCHING '*on*'
    SORTED BY E.LAST_NAME
       writeln (E.LAST_NAME);
END_FOR;

COMMIT;
FINISH;
end.
```

MATCHING Conditional Expression

Example 3

The following programs demonstrate the use of the MATCHING conditional. The programs create a record stream that contains the records in the EMPLOYEES relation in which the LAST_NAME field has a name beginning with the string "Bl" and ending with the string "ck" with only one character between the two strings. These programs might retrieve names such as "Black" and "Block".

C Program

```
#include <stdio.h>
DATABASE PERS = FILENAME "PERSONNEL";

main()
{
READY PERS;
START_TRANSACTION READ_ONLY;

FOR E IN EMPLOYEES
    WITH E.LAST_NAME MATCHING "Bl%ck"
        printf ("%s\n",E.LAST_NAME);
END_FOR;

COMMIT;
FINISH;
}
```

Pascal Program

```
program matching_last (input,output);
DATABASE PERS = FILENAME 'PERSONNEL';

begin
READY PERS;
START_TRANSACTION READ_ONLY;

FOR E IN EMPLOYEES
    WITH E.LAST_NAME MATCHING 'Bl%ck'
        writeln (E.LAST_NAME);
END_FOR;

COMMIT;
FINISH;
end.
```

3.5 MISSING Conditional Expression

The MISSING conditional expression tests for the absence of a field value. A missing value expression will evaluate to true for a given field if no data is stored in the field.

If you precede MISSING with the optional NOT qualifier, the condition is true if the field contains a value.

Format

missing-cond-expr

$\longrightarrow$ db-field-expr $\longrightarrow$ ⌐ NOT ⌐ $\longrightarrow$ MISSING $\longrightarrow$
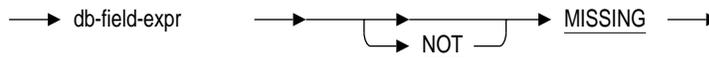
Argument

db-field-expr
A database field value expression. A database field value expression is a field name qualified with a context variable. See Chapter 2 for more information.

Usage Notes

- Use the MISSING conditional expression to test for the absence of a field value.

- Some of the conditions that result in a field being marked as missing are:

  - A STORE statement has been used to explicitly store the MISSING VALUE in a field of a record. For example, if ′Unknown′ is defined as the missing value for the DEGREE_FIELD field in the DEGREES relation, the following STORE statement will mark the DEGREE_ FIELD field as missing for the employee with an EMPLOYEE_ID of 00198.

MISSING Conditional Expression

```
STORE D IN DEGREES USING
 D.EMPLOYEE_ID := '00198';
 D.COLLEGE_CODE := 'PURD';
 D.YEAR_GIVEN := '1982';
 D.DEGREE := 'BA';
 D.DEGREE_FIELD := 'Unknown';
END_STORE;
```

– A STORE statement has been used to store a record, and the field has been omitted from the list of field values stored.

```
STORE D IN DEGREES USING
 D.EMPLOYEE_ID := '00198';
 D.COLLEGE_CODE := 'PURD';
 D.YEAR_GIVEN := '1982';
 D.DEGREE := 'BA';
END_STORE;
```

– A STORE statement has been used to store a record, and the field is assigned the RDB$MISSING value expression.

```
STORE D IN DEGREES USING
   D.EMPLOYEE_ID := '76156';
   D.COLLEGE_CODE := 'HVDU';
   D.YEAR_GIVEN := 1978;
   D.DEGREE := 'BA';
   D.DEGREE_FIELD := RDB$MISSING(D.DEGREE_FIELD);
END_STORE;
```

■ Rdb evaluates the MISSING conditional expression at run time to determine if a field's value is missing.

Examples

Example 1

The following programs demonstrate the use of the MISSING conditional expression. The programs form a record stream that contains the records in the COLLEGES relation that have nothing stored in the STATE field, but do have a college code stored in the COLLEGE_CODE field. Each record in the COLLEGES relation is tested for the previously stored condition; if a record meets the condition these programs print a message and the college code of this record.

MISSING Conditional Expression

C Program

```
#include <stdio.h>
DATABASE PERS = FILENAME "PERSONNEL";

main()
{
READY PERS;
START_TRANSACTION READ_WRITE;

FOR C IN COLLEGES
   WITH C.STATE MISSING
   AND C.COLLEGE_CODE NOT MISSING;
      printf ("State Missing for COLLEGE:  %s\n", C.COLLEGE_CODE);
END_FOR;

COMMIT;
FINISH;
}
```

Pascal Program

```
program missing (input,output);
DATABASE PERS = FILENAME 'PERSONNEL';

begin
READY PERS;
START_TRANSACTION READ_WRITE;

FOR C IN COLLEGES
   WITH C.STATE MISSING
   AND C.COLLEGE_CODE NOT MISSING;
      writeln ('State Missing for COLLEGE: ', C.COLLEGE_CODE);
END_FOR;

COMMIT;
FINISH;
end.
```

Example 2

The following programs demonstrate the use of the MISSING conditional expression. The programs create a record stream that contains the records in the EMPLOYEES relation in which the BIRTHDAY field is marked as empty. These programs then print a message and the last name from the records in the record stream.

MISSING Conditional Expression

C Program

```
#include <stdio.h>
DATABASE PERS = FILENAME "PERSONNEL";

main()
{
READY PERS;
START_TRANSACTION READ_ONLY;

FOR E IN EMPLOYEES
   WITH E.BIRTHDAY NOT MISSING
      printf ("%s exists.\n", E.LAST_NAME);
END_FOR;

COMMIT;
FINISH;
}
```

Pascal Program

```
program missing (input,output);
DATABASE PERS = FILENAME 'PERSONNEL';

begin
READY PERS;
START_TRANSACTION READ_ONLY;

FOR E IN EMPLOYEES
   WITH E.BIRTHDAY NOT MISSING
      writeln (E.LAST_NAME, ' exists');
END_FOR;

COMMIT;
FINISH;
end.
```

3.6 Relational Operators

Relational operators compare value expressions. Relational operators are used in conditional expressions. Table 3–3 lists the RDML relational operators and under which conditions their value is true.

Table 3–3   Relational Operators

Relational Operator		Value
EQ	=	True if the two value expressions are equal.
NE	<>	True if the two value expressions are not equal.
GT	>	True if the first value expression is greater than the second.
GE	>=	True if the first value expression is greater than or equal to the second.
LT	<	True if the first value expression is less than the second.
LE	<=	True if the first value expression is less than or equal to the second.

Note *In all cases, if either value expression is the missing value, the value of the condition is missing.*

Examples

The following programs demonstrate the use of the LE (less than or equal to) operator in a record selection expression. The programs find the employees with an employee ID number that is less than or equal to 00400. Then the programs print the selected employee IDs.

C Program

```
#include <stdio.h>
DATABASE PERS = FILENAME "PERSONNEL";

main()
{
READY PERS;
START_TRANSACTION READ_WRITE;

FOR E IN EMPLOYEES WITH E.EMPLOYEE_ID LE "00400"
   printf ("%s\n", E.EMPLOYEE_ID);
END_FOR;
```

Relational Operators

```
COMMIT;
FINISH;
}
```

Pascal Program

```
program relation (input,output);
DATABASE PERS = FILENAME 'PERSONNEL';

begin
READY PERS;
START_TRANSACTION READ_ONLY;

FOR E IN EMPLOYEES WITH E.EMPLOYEE_ID LE '00400'
    writeln (E.EMPLOYEE_ID);
END_FOR;

COMMIT;
FINISH;
end.
```

3.7 STARTING WITH Conditional Expression

The STARTING WITH conditional expression tests for the presence of a specified string at the beginning of a string expression. This expression is true if the first string expression begins with the characters specified in the second string expression.

If you precede the STARTING WITH expression by the optional NOT qualifier, the condition is true if the first string does not begin with the characters specified by the second string.

Note   *The STARTING WITH conditional expression is case sensitive; it considers uppercase and lowercase forms of the same character to be different.*

Format

starting-with-clause =

$$\longrightarrow \text{value-expr} \longrightarrow \begin{array}{c} \\ \underline{NOT} \end{array} \longrightarrow \underline{STARTING}\ \underline{WITH} \longrightarrow \text{value-expr} \longrightarrow$$

Argument

value-expr
A value expression. A symbol or a string of symbols used to calculate a value. When you use a value expression in a statement, Rdb calculates the value associated with the expression and uses that value when executing the statement.

Usage Notes

- To find records regardless of case:
    - Specify all possibilities in the search condition.
    - Use the CONTAINING conditional expression for searches that are not case sensitive.
- If either of the string expressions in a STARTING WITH conditional expression is missing, the result is missing.

STARTING WITH Conditional Expression

- The STARTING_WITH conditional expression will not execute properly in RDML/Pascal when you use a host language variable of data type PACKED ARRAY for comparison in this expression. For example, in the following code fragment, host-var is the comparison value.

```
FOR E IN EMPLOYEES
   E.LAST_NAME STARTING_WITH host-var
      .
      .
      .
END_FOR;
```

  Note that a PACKED ARRAY data type is generated by the DECLARE_VARIABLE, DEFINE_TYPE, and BASED_ON clauses for field values of data type TEXT.

  Therefore, when you declare a host language variable in an RDML/Pascal program as the comparison value in a STARTING WITH conditional expression, you should declare a variable of data type VARYING STRING. Do not use the DECLARE_VARIABLE, DEFINE_TYPE, or BASED_ON clause to declare this variable.

- Because the STARTING WITH conditional expression is case sensitive, searches for uppercase multinational characters will not include lowercase multinational characters, and vice versa. For example, STARTING WITH "Ç" will retrieve a set of records that is different from those retrieved by STARTING WITH "ç".

  In Spanish, the combinations "ch" and "ll" are treated as if they were individual unique single letters. For example, if a domain is defined with the collating sequence SPANISH, then STARTING WITH "c" will not retrieve the word "char", but it will retrieve the word "cat".

Examples

Example 1

The following programs demonstrate the use of the STARTING WITH conditional expression. The programs create a record stream that contains the records in the EMPLOYEES relation in which the LAST_NAME field contains a name that begins with the string "IACO". Because STARTING WITH is case sensitive, a last name starting with "Iaco" is not the same as a last name starting with "IACO". Names stored in the PERSONNEL database have only the first letter capitalized. Therefore, the programs create an empty record stream and nothing is printed.

STARTING WITH Conditional Expression

C Program

```
#include <stdio.h>
DATABASE PERS = FILENAME "PERSONNEL";

main()
{
READY PERS;
START_TRANSACTION READ_ONLY;

FOR E IN EMPLOYEES
   WITH E.LAST_NAME STARTING WITH "IACO"
      printf( "%s %s\n", E.EMPLOYEE_ID, E.LAST_NAME);
END_FOR;

COMMIT;
FINISH;
}
```

Pascal Program

```
program starting (input,output);
DATABASE PERS = FILENAME 'PERSONNEL';

begin
READY PERS;
START_TRANSACTION READ_ONLY;

FOR E IN EMPLOYEES
   WITH E.LAST_NAME STARTING WITH 'IACO'
      writeln (E.EMPLOYEE_ID, ' ', E.LAST_NAME);
END_FOR;

COMMIT;
FINISH;
end.
```

Example 2

The following programs demonstrate the use of the STARTING WITH
conditional expression. These programs create a record stream that contains
the records in the EMPLOYEES relation in which the LAST_NAME field has
a name that begins with the string "IACO" or "Iaco". The programs print the
employee IDs and last names from each record in the record stream.

STARTING WITH Conditional Expression

C Program

```c
#include <stdio.h>
DATABASE PERS = FILENAME "PERSONNEL";

DECLARE_VARIABLE name1, name2 SAME AS EMPLOYEES.LAST_NAME;

main()
{
strcpy(name1, "IACO");
strcpy(name2, "Iaco");

READY PERS;
START_TRANSACTION READ_ONLY;

FOR E IN EMPLOYEES
    WITH E.LAST_NAME STARTING WITH name1
    OR E.LAST_NAME STARTING WITH name2
        printf("%s %s\n", E.EMPLOYEE_ID, E.LAST_NAME);
END_FOR;

COMMIT;
FINISH;
}
```

Pascal Program

```pascal
program start_two_cond (input,output);
DATABASE PERS = FILENAME 'PERSONNEL';

var
  name1 : VARYING [10] OF CHAR;
  name2 : VARYING [10] OF CHAR;

begin

  name1 := 'IACO';
  name2 := 'Iaco';

READY PERS;
START_TRANSACTION READ_ONLY;

FOR E IN EMPLOYEES
    WITH E.LAST_NAME STARTING WITH 'IACO'
    OR E.LAST_NAME STARTING WITH 'Iaco'
        writeln (E.EMPLOYEE_ID, ' ', E.LAST_NAME);
END_FOR;

COMMIT;
FINISH;
end.
```

STARTING WITH Conditional Expression

Example 3

The following programs demonstrate the use of the NOT STARTING WITH conditional expression and the COUNT statistical function. The programs create a record stream that contains the records in the COLLEGES relation in which the value for the STATE field does not begin with the letter "M". The COUNT statistical function determines the number of records in the record stream and the print statement displays this number.

C Program

```
#include <stdio.h>
DATABASE PERS = FILENAME "PERSONNEL";

int atot;

main()
{
READY PERS;
START_TRANSACTION READ_ONLY;

GET
  atot = COUNT OF C IN COLLEGES WITH C.STATE NOT STARTING WITH "M";
END_GET;

COMMIT;

printf ("%d", atot);
FINISH;
}
```

Pascal Program

```
program starting (input,output);
DATABASE PERS = FILENAME 'PERSONNEL';

var
atot : integer;

begin
READY PERS;
START_TRANSACTION READ_ONLY;

GET
  atot = COUNT OF C IN COLLEGES WITH C.STATE NOT STARTING WITH 'M';
END_GET;

COMMIT;
writeln (atot);

FINISH;
end.
```

3.8 UNIQUE Conditional Expression

The UNIQUE conditional expression tests for the presence of a single record in a record stream. This expression is true if the record stream specified by the record selection expression consists of only one record.

If you precede UNIQUE with the optional NOT qualifier, the condition is true if more than one record is in the record stream or if the stream is empty.

Format

unique-clause =

handle-options =
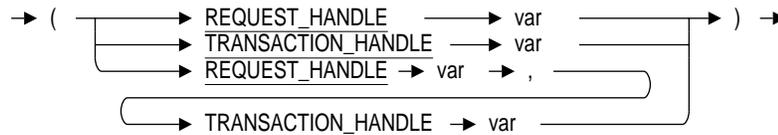
Arguments

handle-options
A request handle, a transaction handle, or both.

REQUEST_HANDLE var
A REQUEST_HANDLE keyword followed by a host language variable. A request handle identifies a compiled Rdb/VMS request. If you do not supply a request handle explicitly, RDML generates a unique request handle for the compiled request. See Section 6.20 for more information on request handles.

TRANSACTION_HANDLE var

A TRANSACTION_HANDLE keyword followed by a host language variable. A transaction handle identifies a transaction. If you do not supply a transaction handle explicitly, RDML associates a default transaction handle with the transaction.

rse

A record selection expression. A phrase that defines specific conditions that individual records must meet before Rdb includes them in a record stream. See Chapter 4 for more information.

Examples

Example 1

The following programs demonstrate the use of the UNIQUE conditional expression. The programs join the EMPLOYEES and DEGREES relations over the EMPLOYEE_ID common field. The UNIQUE expression limits the record stream to those records in the EMPLOYEES relation that have only one corresponding record in the DEGREES relation. These programs print an informational message and the selected employees' first and last names in alphabetical order based on the first name.

C Program

```
#include <stdio.h>
DATABASE PERS = FILENAME "PERSONNEL";

main()
{
READY PERS;
START_TRANSACTION READ_ONLY;

FOR E IN EMPLOYEES
    SORTED BY E.FIRST_NAME
    WITH UNIQUE D IN DEGREES WITH D.EMPLOYEE_ID = E.EMPLOYEE_ID
        printf("%s %s has one and only one college degree.\n",
                    E.FIRST_NAME, E.LAST_NAME);
END_FOR;

COMMIT;
FINISH;
}
```

UNIQUE Conditional Expression

Pascal Program

```
program unique_expr (input,output);
DATABASE PERS = FILENAME 'PERSONNEL';

begin
READY PERS;
START_TRANSACTION READ_ONLY;

FOR E IN EMPLOYEES
   WITH UNIQUE D IN DEGREES WITH D.EMPLOYEE_ID = E.EMPLOYEE_ID
      writeln (E.FIRST_NAME, ' ', E.LAST_NAME,
               ' has one and only one college degree.');
END_FOR;

COMMIT;
FINISH;
end.
```

Example 2

The following programs demonstrate the use of the NOT UNIQUE conditional expression. The programs join the EMPLOYEES and SALARY_HISTORY relations over the EMPLOYEE_ID common field. The NOT UNIQUE conditional expression limits the records in the record stream to those records in the EMPLOYEE relation that have more than one corresponding record in the SALARY_HISTORY relation. The SORTED BY clause sorts the records in alphabetical order. These programs print the last names of the employees in the record stream, and an informational message.

C Program

```
#include <stdio.h>
DATABASE PERS = FILENAME "PERSONNEL";

main()
{
READY PERS;
START_TRANSACTION READ_ONLY;

FOR E IN EMPLOYEES
   WITH (ANY SH IN SALARY_HISTORY
         WITH E.EMPLOYEE_ID = SH.EMPLOYEE_ID)
   AND  (NOT UNIQUE SH IN SALARY_HISTORY
         WITH E.EMPLOYEE_ID = SH.EMPLOYEE_ID)
   SORTED BY E.LAST_NAME
      printf("%s has had two or more salary reviews.\n", E.LAST_NAME);
END_FOR;

COMMIT;
FINISH;
}
```

UNIQUE Conditional Expression

Pascal Program

```
program unique_not (input,output);
DATABASE PERS = FILENAME 'PERSONNEL';

begin
READY PERS;
START_TRANSACTION READ_ONLY;

FOR E IN EMPLOYEES
   WITH (ANY SH IN SALARY_HISTORY
            WITH E.EMPLOYEE_ID = SH.EMPLOYEE_ID)
   AND (NOT UNIQUE SH IN SALARY_HISTORY
        WITH E.EMPLOYEE_ID = SH.EMPLOYEE_ID)
   SORTED BY E.LAST_NAME
      writeln (E.LAST_NAME, ' has had two or more salary reviews.');
END_FOR;

COMMIT;
FINISH;
end.
```

Example 3

The following programs demonstrate the use of the UNIQUE conditional
expression in a reflexive join. These programs create two record streams by
joining the EMPLOYEES relation with itself. This is achieved by declaring two
context variables, E and EMP, for the EMPLOYEES relation. RDML compares
the CITY field of each record in the EMPLOYEES relation with every other
record in the same relation. The UNIQUE conditional expression selects the
records in which one and only one employee lives in any given city. These
programs print an informational message and the city, first name, and last
name of each of those employees.

C Program

```
#include <stdio.h>
DATABASE PERS = FILENAME "PERSONNEL";

main()
{
READY PERS;
START_TRANSACTION READ_ONLY;

FOR E IN EMPLOYEES
   WITH UNIQUE EMP IN EMPLOYEES
        WITH E.CITY = EMP.CITY
      printf ("City is:  %s\n", E.CITY);
      printf ("Employee name is:  %s  %s\n\n",E.FIRST_NAME, E.LAST_NAME);
END_FOR;

COMMIT;
FINISH;
}
```

UNIQUE Conditional Expression

Pascal Program

```
program unique_cond_exp (input,output);
DATABASE PERS = FILENAME 'PERSONNEL';

begin
READY PERS;
START_TRANSACTION READ_ONLY;

FOR E IN EMPLOYEES
   WITH UNIQUE EMP IN EMPLOYEES
        WITH E.CITY = EMP.CITY
     writeln ('City is:   ', E.CITY);
     writeln ('Employee name is:   ',E.FIRST_NAME, E.LAST_NAME);
     writeln;
END_FOR;

COMMIT;
FINISH;
end.
```

4

RDML Record Selection Expressions

This chapter describes the Relational Data Manipulation Language (RDML) record selection expressions (RSEs) that can be used with embedded RDML statements in C and Pascal programs.

The C and Pascal programs in this chapter access the sample personnel database available with Rdb/VMS.

A **record selection expression** is an expression that defines specific conditions individual records must meet before Rdb includes them in a record stream. A **record stream** is a temporary group of related records that satisfy the conditions you specify in the record selection expression.

Record selection expressions let you:

- Include all records in the relation
- Eliminate duplicate records
- Limit the number of records displayed
- Test for values and conditions
- Sort the records in the record stream
- Combine records from the same or different relations

Format

rse =
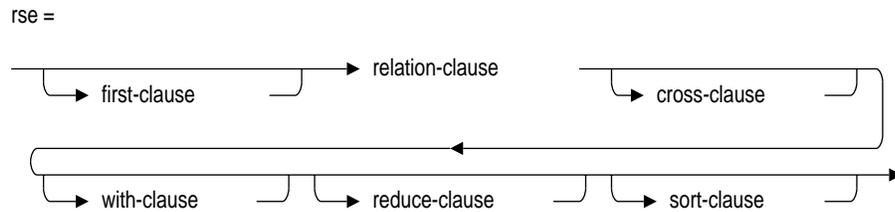
Table 4–1 summarizes the function of each record selection expression clause.

Table 4–1   Record Selection Expression Clause Functions

RSE Clause	Function
CROSS	Joins records from two or more relations.
FIRST *N*	Restricts the record stream to the number of records specified by "N".
REDUCED TO	Isolates unique field values within the record stream.
Relation	Declares context variable for the record stream.
SORTED BY	Sorts records in the record stream by values of specific fields.
WITH	Specifies conditions that must be met for records to be included in the record stream.

Usage Notes

- You can use simple and complex host language variables, such as arrays or records, in a record selection expression. However, do not use functions or procedures within the record selection expression. Instead, assign the result of a function to a host language variable and use that variable within the record selection expression. For example, the following Pascal code will not preprocess:

```
(* Bad code - will not preprocess *)
FOR FIRST 5 E IN EMPLOYEES WITH E.LAST_NAME = SUBSTR(STRING,1,24)
   writeln (E.LAST_NAME);
END_FOR;
```

However, this code will preprocess:

```
host_variable = SUBSTR(STRING,1,24)

FOR FIRST 5 E IN EMPLOYEES WITH E.LAST_NAME = host_variable
   writeln (E.LAST_NAME);
END_FOR;
```

■ Record selection expressions cause relations to be referred to in a request in specific ways. Because there is an implementation-specific limit on the number of relations that you can refer to in a request, you need to know that the following factors cause a relation to be referenced:

  – The name of a relation or view in a record selection expression.

  – The relations in a view (or virtual relation). Thus, if a view refers to three relations, referring to that view is the same as referring to four relations; one for the view, and one for each relation contained in the view. For example, the following data definition language (DDL) record selection expression defines a view, NULL_MANAGERS, derived from the DEPARTMENTS relation (note that the following is not an RDML statement):

```
DEFINE VIEW NULL_MANAGERS OF
   MGR IN JOB_HISTORY WITH MGR.DEPARTMENT_NAME MISSING.
END NULL_MANAGERS VIEW.
```

  The RDML statement that follows refers to three relations:

  1 The view, or virtual relation, NULL_MANAGERS

  2 The JOB_HISTORY relation referred to in NULL_MANAGERS

  3 The JOB_HISTORY relation in the CROSS clause

```
FOR MGR IN NULL_MANAGERS CROSS JH IN JOB_HISTORY
     WITH MGR.SUPERVISOR_ID = JH.EMPLOYEE_ID
   WRITE ('The manager with this ID number: ', MGR.SUPERVISOR_ID,
          'has an unknown department name');
END_FOR;
```

  – The relations in a DDL record selection expression that has a COMPUTED BY field. This includes COMPUTED BY fields that refer to other COMPUTED BY fields. For example, the DDL that follows defines a view of UNIQUE_DEGREES that refers to these three relations:

  1 The view, or virtual relation, UNIQUE_DEGREES

  2 The DEGREES relation that is referred to in the view UNIQUE_DEGREES

3　The DEGREES relation that is computed so a total of persons
　　holding a degree can be found

```
DEFINE VIEW UNIQUE_DEGREES OF
   D IN DEGREES REDUCED TO D.DEGREE.
  D.DEGREE.
  HOLDERS
     COMPUTED BY COUNT OF H IN DEGREES
          WITH H.DEGREE = D.DEGREE.
END UNIQUE_DEGREES VIEW.
```

- You should use parentheses to delineate multiple statistical functions in
  record selection expressions. Examples of statistical functions are COUNT,
  TOTAL, and MAX.

- If you use a statistical function (for example, COUNT) with a record
  selection expression, enclose it in a GET statement. Embedding the
  statistical function in a GET statement incurs less overhead than a
  statistical function embedded directly in the host language. The following
  Pascal example shows the use of the GET statement with the SORTED BY
  clause:

```
GET
 acnt =  COUNT OF POOR IN CURRENT_SALARY
          CROSS RICH IN CURRENT_SALARY
          WITH RICH.SALARY_AMOUNT > (10 * POOR.SALARY_AMOUNT)
          SORTED BY POOR.EMPLOYEE_ID;
END_GET;

   writeln ('There are', acnt, 'employees who deserve a raise');
```

If you must use a statistical function within the host language, use
parentheses to delineate both expressions from a program function if
necessary to enforce the order of precedence you desire.

Examples

Example 1

The following programs demonstrate the use of CROSS, WITH, and SORTED
BY record selection expression clauses. These programs generate a report for
the personnel department that shows important information about each active
employee, including salary level attained for each job and the department to
which the employee belongs.

The EMPLOYEES relation describes each employee in the company. The SALARY_HISTORY relation contains current salary information along with the salary start date and salary amount for that job. The JOB_HISTORY relation holds data about each job an employee holds and has held, including the department and job code. The JOBS relation contains information about each job in the company. Each of these relations supplies some data for the report.

To obtain the necessary fields from each, the programs contain a query to join the four relations. The WITH clause ensures that the query uses related fields in each relation.

Note that the SALARY_START field is a DATE data type. In the database, it is stored in an encoded binary format. To display it, the program must first convert the retrieved value into an ASCII string. This program calls the VMS system service ASCTIM to perform the conversion.

C Program

```
#include <stdio.h>
#include <descrip.h>
#include <ssdef.h>

DATABASE PERS = FILENAME "PERSONNEL";

extern int SYS$ASCTIM( );

main()
{
/* In the following declaration, note one extra space for EOS */
static $DESCRIPTOR(SAL_DATE,"dd-mmm-yyyy hh:mm:ss.cc ");

/* SYS$ASCTIM returns "len" in a 16-bit word    */
short len;
long  status;

READY PERS;
START_TRANSACTION READ_ONLY;

FOR E IN EMPLOYEES
   CROSS SH IN SALARY_HISTORY
   CROSS JH IN JOB_HISTORY
   CROSS J IN JOBS
   WITH JH.JOB_CODE = J.JOB_CODE
   AND SH.SALARY_END MISSING
   AND E.EMPLOYEE_ID = SH.EMPLOYEE_ID
   AND E.EMPLOYEE_ID = JH.EMPLOYEE_ID
   AND JH.JOB_END MISSING
   SORTED BY J.JOB_CODE,E.EMPLOYEE_ID;
```

```
        status = SYS$ASCTIM ( &len, &SAL_DATE, SH.SALARY_START, 0);
        if (status != SS$_NORMAL)
            {
            printf("Date conversion failed\n");
            continue;
            }

        /* Ensure that the returned string is null-terminated,    */
        /* so that we may use printf to display it.               */

        SAL_DATE.dsc$a_pointer[len - 1] = '\0';

        printf ("Job Code        %s\n",J.JOB_CODE);
        printf ("Employee ID     %s\n",E.EMPLOYEE_ID);
        printf ("Name            %s  %s\n",E.FIRST_NAME,E.LAST_NAME);
        printf ("Dept Code       %s\n",JH.DEPARTMENT_CODE);
        printf ("Job Title       %s\n",J.JOB_TITLE);
        printf ("Start Date      %s\n",SAL_DATE.dsc$a_pointer);
        printf ("Current Salary  $%f\n\n",SH.SALARY_AMOUNT);

END_FOR;

COMMIT;
FINISH;
}
```

Pascal Program

```
[inherit ('sys$library:starlet.pen')]

program salary_report (input,output);
DATABASE PERS = FILENAME 'PERSONNEL';

type date_asc_type = packed array [1..23] of char;
var sal_date  : date_asc_type;
 sys_stat : integer;

begin
READY PERS;
START_TRANSACTION READ_ONLY;

FOR E IN EMPLOYEES
   CROSS SH IN SALARY_HISTORY
   CROSS JH IN JOB_HISTORY
   CROSS J IN JOBS
   WITH JH.JOB_CODE = J.JOB_CODE
   AND SH.SALARY_END MISSING
   AND E.EMPLOYEE_ID = SH.EMPLOYEE_ID
   AND E.EMPLOYEE_ID = JH.EMPLOYEE_ID
   AND JH.JOB_END MISSING
   SORTED BY J.JOB_CODE,E.EMPLOYEE_ID


   writeln ('Job Code        ',J.JOB_CODE);
   writeln ('Employee ID     ',E.EMPLOYEE_ID);
   writeln ('Name            ',E.FIRST_NAME,' ',E.LAST_NAME);
   writeln ('Dept Code       ',JH.DEPARTMENT_CODE);
   writeln ('Job Title       ',J.JOB_TITLE);
```

```
            sys_stat := $ASCTIM( timbuf := sal_date, timadr := SH.SALARY_START);
            if (sys_stat <> SS$_NORMAL) then
               writeln ('Date conversion failed')
            else
                 writeln ('Start Date        ',sal_date);
        writeln ('Current Salary   $',SH.SALARY_AMOUNT : 10 : 2);
        writeln;

    END_FOR;

    COMMIT;
    FINISH;
    end.
```

4.1 Context Variable

A **context variable** is a temporary name that identifies a relation in an Rdb record stream. Once you have associated a context variable with a relation, you use the context variable to refer to fields from that relation. In this way, Rdb always identifies the specific field and its particular relationship to which you refer.

You must use a context variable in every data manipulation statement that uses a record selection expression.

If you access several record streams at once, the context variable lets you distinguish between fields from different record streams, even if different fields have the same name.

If you access several record streams at once that consist of the same relation and fields within that relation, context variables let you distinguish between the two record streams.

Format

context-variable =

$\longrightarrow$ identifier $\longrightarrow$

Argument

identifier
A valid alphanumeric host language identifier.

Usage Notes

- Context variables are defined explicitly by the record selection expression (one context variable for each instance of a participating relation).

- The context established by the context variable is valid during the execution of the statement or clause in which the context variable is declared.

Context Variable

- Context variables establish a context within which RDML resolves references to database fields. This context affects only the statement in which the context variable is declared. All inner (contained or nested) statements and all outer (containing or nesting) statements are not affected.

- Context variables are implicit in an OVER clause that names a common field. In the following example, a context variable is not used to identify EMPLOYEE_ID in the OVER clause:

```
FOR E IN EMPLOYEES
    CROSS D IN DEGREES
    OVER EMPLOYEE_ID
```

- The context established by a context variable is valid during the execution of the statement or clause in which the context variable is declared. For example, a context variable declared in a FOR statement is only valid within the FOR . . . END_FOR block, whereas the context variable declared by the DECLARE_STREAM statement is valid from the execution of the DECLARE_STREAM statement to the end of the program module.

- Context variables are referred to in the following clauses, statements, functions, and expressions:

  - ERASE statement

  - FOR statement

  - MODIFY statement

  - STORE statement

  - START_STREAM statements

  - Record selection expressions

  - Field reference

  - Database key value reference

  - Statistical and Boolean functions

Context Variable

Examples

Example 1

The following programs demonstrate the use of the context variable "CS" for the CURRENT_SALARY view. These programs:

- Use "CS" to qualify field names in the record selection expression, printf, and writeln statement

- Print the employee ID of all the employees who earn more than $40,000

C Program

```
#include <stdio.h>
DATABASE PERS = FILENAME "PERSONNEL";

main()
{
READY PERS;
START_TRANSACTION READ_ONLY;

FOR CS IN CURRENT_SALARY WITH CS.SALARY_AMOUNT > 40000.00
    printf ("%s\n",CS.EMPLOYEE_ID);
END_FOR;

COMMIT;
FINISH;
}
```

Pascal Program

```
program context_var (input,output);
DATABASE PERS = FILENAME 'PERSONNEL';

begin
READY PERS;
START_TRANSACTION READ_ONLY;

FOR CS IN CURRENT_SALARY WITH CS.SALARY_AMOUNT > 40000.00
    writeln (CS.EMPLOYEE_ID);
END_FOR;

COMMIT;
FINISH;
end.
```

Example 2

The following programs demonstrate the use of two context variables, E for the EMPLOYEES relation and SH for the SALARY_HISTORY relation, to qualify the EMPLOYEE_ID field used in both relations. The programs produce a report about each employee's starting and ending dates at the company.

Note that the SALARY_START and SALARY_END fields from the SALARY_ HISTORY relation are DATE data types. In the database, a DATE field is stored in an encoded binary format. To display it, the program must first convert the retrieved value into an ASCII string. This program calls the VMS system service ASCTIM to perform the conversion.

C Program

```
#include <stdio.h>
#include <descrip.h>
#include <ssdef.h>

DATABASE PERS = FILENAME "PERSONNEL";

extern int SYS$ASCTIM( );

main()
{
/* In following two declarations, note one extra space for EOS  */
static  $DESCRIPTOR(SAL_START, "dd-mmm-yyyy hh:mm:ss.cc ");
static  $DESCRIPTOR(SAL_END, "dd-mmm-yyyy hh:mm:ss.cc ");

/* SYS$ASCTIM returns len in a 16-bit word   */
short   len_start,len_end;
long    status;

READY PERS;
START_TRANSACTION READ_ONLY;

FOR E IN EMPLOYEES CROSS SH IN SALARY_HISTORY
   WITH E.EMPLOYEE_ID = SH.EMPLOYEE_ID
   SORTED BY E.LAST_NAME;

   status = SYS$ASCTIM ( &len_start, &SAL_START, SH.SALARY_START, 0);
      if (status != SS$_NORMAL)
         {
          printf("Date conversion failed\n");
          continue;
          }
   status = SYS$ASCTIM ( &len_end,   &SAL_END, SH.SALARY_END, 0);
      if (status != SS$_NORMAL)
         {
         printf("Date conversion failed\n");
         continue;
         }

   /* Ensure that the returned strings are null-terminated,    */
   /* so that we may use printf to print them out.             */
```

Context Variable

```
     SAL_START.dsc$a_pointer[len_start - 1] = '\0';
     SAL_END.dsc$a_pointer[len_end - 1]     = '\0';
     printf ("%s  %s  %s\n",
             E.LAST_NAME,
             SAL_START.dsc$a_pointer,
             SAL_END.dsc$a_pointer);

END_FOR;

COMMIT;
FINISH;
}
```

Pascal Program

```
[INHERIT ('SYS$LIBRARY:STARLET.PEN')]

program two_fields (input,output);
DATABASE PERS = FILENAME 'PERSONNEL';

type DATE_ASC_TYPE = PACKED ARRAY [1..23] OF CHAR;
var Sal_Start : DATE_ASC_TYPE;
        Sal_End  : DATE_ASC_TYPE;
 Sys_Stat1 : INTEGER;
 Sys_Stat2 : INTEGER;

begin
READY PERS;
START_TRANSACTION READ_ONLY;

FOR E IN EMPLOYEES CROSS SH IN SALARY_HISTORY
   WITH E.EMPLOYEE_ID = SH.EMPLOYEE_ID
   SORTED BY E.LAST_NAME

   Sys_Stat1  := $ASCTIM( timbuf := Sal_Start, timadr := SH.SALARY_START);
   Sys_Stat2  := $ASCTIM( timbuf := Sal_End,   timadr := SH.SALARY_END);
   if ((sys_stat1 <> SS$_NORMAL) OR (sys_stat2 <> SS$_NORMAL))
      then writeln ('Date conversion failed')
   else
      writeln (E.LAST_NAME, ' ', Sal_Start, '  ', Sal_End);

END_FOR;

COMMIT;
FINISH;
end.
```

4.2 CROSS Clause

The record selection expression's CROSS clause lets you combine records from two or more record streams. You can base such record combinations on the relationship between field values in separate record streams. This combination is called a **relational join**.

Format
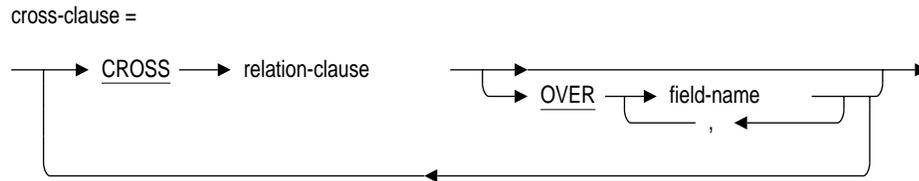
cross-clause =

Arguments

relation-clause
A clause that specifies a context variable for a stream or a loop. For more information on context variables see Section 4.1.

field-name
The name of a field common to both of the relations.

Usage Notes

- You cannot cross relations from different databases. A record selection expression may refer to only one database at a time. Instead, you can use a nested FOR loop to combine data from different databases.

- If you use the OVER clause when crossing more than two relations, the field name specified in the optional OVER clause must appear in only two of the relations. If the field name appears in more than two of the relations that you are crossing, RDML returns an error.

  For example, the clause "R0 IN REL0 CROSS R1 IN REL1 CROSS R2 IN REL2 OVER F1" is valid if, and only if, F1 is a field that appears in relation REL2 and in either relation REL0 or REL1, but not both.

CROSS Clause

- The CROSS clause is more efficient if the fields shared by the relations have indexes defined for them.

- The CROSS clause, used with neither the WITH nor the OVER clause, forms the cross product of relations. A cross product is the result of matching each record of one relation with each record of the other relation. In most cases, the cross product alone is not useful. Normally, you want to limit the returned records by using one or more of the following record selection expression clauses:

  - FIRST

  - WITH

  - SORTED BY

  - REDUCED

  - OVER

- Do not update a view that refers to more than one relation. Attempts to do so could cause unexpected results in your database.

- Using an OVER clause is equivalent to specifying a WITH clause that contains a conditional expression. For example, the following two RDML Pascal queries use WITH and OVER clauses, respectively, to achieve the same result;

  Query 1

  ```
  START_TRANSACTION READ_ONLY;

  FOR E IN EMPLOYEES CROSS JH IN JOB_HISTORY
  WITH E.EMPLOYEE_ID = JH.EMPLOYEE_ID
     WRITE (E.EMPLOYEE_ID, ' ', E.LAST_NAME, ' ');
     WRITE (JH.JOB_CODE, ' ', JH.DEPARTMENT_CODE);
     WRITELN;
  END_FOR;
  COMMIT;
  ```

  Query 2

  ```
  START_TRANSACTION READ_ONLY;

  FOR E IN EMPLOYEES CROSS JH IN JOB_HISTORY OVER EMPLOYEE_ID
     WRITE (E.EMPLOYEE_ID, ' ', E.LAST_NAME, ' ');
     WRITE (JH.JOB_CODE, ' ', JH.DEPARTMENT_CODE);
     WRITELN;
  END_FOR;

  COMMIT;
  ```

Examples

Example 1

The following programs demonstrate the use of the CROSS clause to join records from two relations. These programs join the relations CURRENT_JOB and JOBS over the common JOB_CODE field. This allows these programs to print a report that contains fields from both relations. Specifically, these fields are: LAST_NAME from the CURRENT_JOBS relation, JOB_CODE from the JOBS relation, and JOB_TITLE from the JOBS relation.

C Program

```
#include <stdio.h>
DATABASE PERS = FILENAME "PERSONNEL";

main()
{
READY PERS;
START_TRANSACTION READ_ONLY;

FOR CJ IN CURRENT_JOB
   CROSS J IN JOBS OVER JOB_CODE
      printf ("%s",CJ.LAST_NAME);
      printf ("  %s",J.JOB_CODE);
      printf ("  %s\n", J.JOB_TITLE);
END_FOR;

COMMIT;
FINISH;
}
```

Pascal Program

```
program person_job (input,output);
DATABASE PERS = FILENAME 'PERSONNEL';

begin
READY PERS;
START_TRANSACTION READ_ONLY;

FOR CJ IN CURRENT_JOB
   CROSS J IN JOBS OVER JOB_CODE
      writeln (CJ.LAST_NAME, ' ',J.JOB_CODE, ' ',J.JOB_TITLE);
END_FOR;

COMMIT;
FINISH;
end.
```

CROSS Clause

Example 2

The following programs demonstrate the use of the CROSS clause to join a relation with itself (a reflexive join). These programs:

- Join the JOBS relation on itself

- Specify two different context variables, STAFF and EXEC, for the JOBS relation

- Form a stream with records that contain data on pairs of employees, STAFF and EXEC

- Form these pairs when:

  - The wage class of a staff member is equal to 2, and the wage class of the executive is equal to 4

  - The staff member's maximum salary amount is greater than the minimum salary amount of an executive

- Print the job code of each staff member and the maximum salary he or she can be paid

- Print the job code of each executive and the minimum salary he or she can be paid

C Program

```
#include <stdio.h>
DATABASE PERS = FILENAME "PERSONNEL";

main()
{
READY PERS;
START_TRANSACTION READ_ONLY;

FOR EXEC IN JOBS
   CROSS STAFF IN JOBS
   WITH EXEC.WAGE_CLASS = '4'
   AND STAFF.WAGE_CLASS = '2'
   AND STAFF.MAXIMUM_SALARY > EXEC.MINIMUM_SALARY
      printf ("%s",STAFF.JOB_CODE);
      printf ("  %f\n",STAFF.MAXIMUM_SALARY);
      printf ("%s",EXEC.JOB_CODE);
      printf ("  %f\n",EXEC.MINIMUM_SALARY);
END_FOR;

COMMIT;
FINISH;
}
```

Pascal Program

```
program reflexive_join (input,output);
DATABASE PERS = FILENAME 'PERSONNEL';

begin
READY PERS;
START_TRANSACTION READ_ONLY;

FOR EXEC IN JOBS
   CROSS STAFF IN JOBS
   WITH EXEC.WAGE_CLASS = '4'
   AND STAFF.WAGE_CLASS = '2'
   AND STAFF.MAXIMUM_SALARY > EXEC.MINIMUM_SALARY
      writeln (STAFF.JOB_CODE);
      writeln (STAFF.MAXIMUM_SALARY:10:2);
      writeln (EXEC.JOB_CODE);
      writeln (EXEC.MINIMUM_SALARY:10:2);
      writeln
END_FOR;

COMMIT;
FINISH;
end.
```

Example 3

The following programs demonstrate the use of the CROSS clause and the
REDUCED TO clause in a reflexive join. These programs create two context
variables, POOR and RICH, for the CURRENT_SALARY view. This allows
the program to compare records in the CURRENT_SALARY relation to each
other. The WITH clause selects records from the EMPLOYEES relation in
which the salary amount of an employee in the POOR record stream is, at
most, 10 percent of the salary earned by any other employee in the relation.
The REDUCED TO clause ensures that duplicate records (based on employee
ID) are discarded from the selection. These programs print an informational
message and the employee IDs of the POOR employees.

C Program

```
#include <stdio.h>
DATABASE PERS = FILENAME "PERSONNEL";

main()
{
READY PERS;
START_TRANSACTION READ_ONLY;
```

CROSS Clause

```
FOR POOR IN CURRENT_SALARY
   CROSS RICH IN CURRENT_SALARY
   WITH RICH.SALARY_AMOUNT > (10.0 * POOR.SALARY_AMOUNT)
   REDUCED TO POOR.EMPLOYEE_ID
      printf ("%s deserves a raise\n",POOR.EMPLOYEE_ID);
END_FOR;

COMMIT;
FINISH;
}
```

Pascal Program

```
program salary_info (input,output);
DATABASE PERS = FILENAME 'PERSONNEL';

begin
START_TRANSACTION READ_ONLY;

FOR POOR IN CURRENT_SALARY
   CROSS RICH IN CURRENT_SALARY
   WITH RICH.SALARY_AMOUNT > (10.0 * POOR.SALARY_AMOUNT)
   REDUCED TO POOR.EMPLOYEE_ID
      writeln (POOR.EMPLOYEE_ID, ' deserves a raise.');
END_FOR;

COMMIT;
FINISH;
end.
```

Example 4

The following programs demonstrate the use of the CROSS clause to join fields
from two relations. These programs join the EMPLOYEES relation and the
DEGREES relation over the EMPLOYEE_ID field. The programs print a list of
all the employees' IDs and college degrees from the COLLEGES relation. The
REDUCED TO clause ensures that this list does not contain duplicate pairings
of employee IDs and degrees.

C Program

```
#include <stdio.h>
DATABASE PERS = FILENAME "PERSONNEL";

main()
{
READY PERS;
START_TRANSACTION READ_ONLY;
```

```
FOR E IN EMPLOYEES
   CROSS D IN DEGREES OVER EMPLOYEE_ID
   REDUCED TO E.EMPLOYEE_ID,D.DEGREE
       printf ("%s",E.EMPLOYEE_ID);
       printf ("   %s\n",D.DEGREE);
END_FOR;

COMMIT;
FINISH;
}
```

Pascal Program

```
program cross_with_reduced (input,output);
DATABASE PERS = FILENAME 'PERSONNEL';

begin
READY PERS;
START_TRANSACTION READ_ONLY;

FOR E IN EMPLOYEES
   CROSS D IN DEGREES OVER EMPLOYEE_ID
   REDUCED TO E.EMPLOYEE_ID, D.DEGREE
       write  (E.EMPLOYEE_ID,'  ');
       writeln (D.DEGREE);
END_FOR;

COMMIT;
FINISH;
end.
```

Example 5

The following programs demonstrate the use of the CROSS clause to join three relations over multiple join fields. These programs create a record stream that contains records from the EMPLOYEES, JOB_HISTORY, and JOBS relations. A record from the JOB_HISTORY relation is included in the record stream only if it has a corresponding record in EMPLOYEES relation (based on EMPLOYEE_ID) and a corresponding record in the JOBS relation (based on the JOB_CODE field). These programs print information from records in the record stream using fields from both the JOB_HISTORY and JOBS relations.

C Program

```
#include <stdio.h>
DATABASE PERS = FILENAME "PERSONNEL";

main()
{
READY PERS;
START_TRANSACTION READ_ONLY;
```

CROSS Clause

```
FOR E IN EMPLOYEES
   CROSS JH IN JOB_HISTORY
   CROSS J IN JOBS
   WITH E.EMPLOYEE_ID = JH.EMPLOYEE_ID
   AND JH.JOB_CODE = J.JOB_CODE
      printf ("%s",JH.EMPLOYEE_ID);
      printf ("   %s",JH.DEPARTMENT_CODE);
      printf ("   %s",JH.JOB_CODE);
      printf ("   %s\n",J.WAGE_CLASS);
      printf ("%s",J.JOB_TITLE);
      printf ("   %f",J.MINIMUM_SALARY);
      printf ("   %f\n\n",J.MAXIMUM_SALARY);
END_FOR;

COMMIT;
FINISH;
}
```

Pascal Program

```
program mult_join_fields (input,output);
DATABASE PERS = FILENAME 'PERSONNEL';

begin
READY PERS;
START_TRANSACTION READ_ONLY;

FOR E IN EMPLOYEES
   CROSS JH IN JOB_HISTORY
   CROSS J IN JOBS
   WITH E.EMPLOYEE_ID = JH.EMPLOYEE_ID
   AND JH.JOB_CODE = J.JOB_CODE
      write   (JH.EMPLOYEE_ID, '  ');
      write   (JH.DEPARTMENT_CODE,'  ');
      write   (JH.JOB_CODE,'  ');
      writeln (J.WAGE_CLASS);
      write   (J.JOB_TITLE);
      write   (J.MINIMUM_SALARY:10:2);
      writeln (J.MAXIMUM_SALARY:10:2);
      writeln;
END_FOR;

COMMIT;
FINISH;
end.
```

Example 6

The following programs demonstrate the use of the CROSS clause to join a relation with itself and with another relation. These programs:

- Join CURRENT_JOB with itself and then with JOBS on the JOB_CODE CJ2

- Select only those records for which the EMPLOYEE_ID in CJ1 is the same as the SUPERVISOR_ID in CJ2

- Display the employee's name, his or her supervisor's name, and his or her manager's title

C Program

```
#include <stdio.h>
DATABASE PERS = FILENAME "PERSONNEL";

main()
{
READY PERS;
START_TRANSACTION READ_ONLY;

FOR CJ1 IN CURRENT_JOB
   CROSS CJ2 IN CURRENT_JOB
   CROSS J  IN JOBS WITH J.JOB_CODE = CJ2.JOB_CODE
   AND CJ1.SUPERVISOR_ID = CJ2.EMPLOYEE_ID
      printf ("Employee:   %s  ", CJ1.LAST_NAME);
      printf ("Boss:    %s ", CJ2.LAST_NAME);
      printf ("Managers Title: %s\n ", J.JOB_TITLE);
END_FOR;

COMMIT;
FINISH;
}
```

Pascal Program

```
program self_and_another (input,output);
DATABASE PERS = FILENAME 'PERSONNEL';

begin
READY PERS;
START_TRANSACTION READ_ONLY;
```

CROSS Clause

```
FOR CJ1 IN CURRENT_JOB
   CROSS CJ2 IN CURRENT_JOB
   CROSS J IN JOBS
   WITH J.JOB_CODE = CJ2.JOB_CODE
   AND CJ1.SUPERVISOR_ID = CJ2.EMPLOYEE_ID
      writeln ('Employee: ', CJ1.LAST_NAME,
               ' Boss: ', CJ2.LAST_NAME,
               ' Manager''s Title: ', J.JOB_TITLE);
END_FOR;

COMMIT;
FINISH;
end.
```

4.3 FIRST Clause

The FIRST clause allows you to specify the maximum number of records to be included in a record stream formed by a record selection expression.

Format

first-clause =

$\longrightarrow$ FIRST $\longrightarrow$ value-expr $\longrightarrow$

Argument

value-expr
A value expression. A symbol or a string of symbols used to calculate a value. When you use a value expression in a statement, Rdb calculates the value associated with the expression and uses that value when executing the statement.

Usage Notes

- If the value expression is greater than the number of records that satisfy the conditions of the record selection expression, Rdb returns all the records it finds. For example:

```
FOR FIRST 50000 E IN EMPLOYEES
   WRITELN (E.LAST_NAME, E.EMPLOYEE_ID);
END_FOR;
```

  If only 10,000 records are in the EMPLOYEES relation, Rdb returns only those 10,000. It does not produce any informational messages that indicate the discrepancy between the requested number of records and the number actually returned.

- If the value expression evaluates to a zero or a negative number, the record stream is empty. Rdb will not return any records, nor will it generate an error.

FIRST Clause

- If you specify a sort order in the record selection expression, Rdb first sorts the records that satisfy the conditions of the record selection expression. Although many records may satisfy those conditions, the FIRST qualifier restricts the number of records in the record stream *after sorting*. For example:

```
FOR FIRST 10 E IN EMPLOYEES SORTED BY E.EMPLOYEE_ID
   WRITELN (E.LAST_NAME, E.EMPLOYEE_ID);
END_FOR;
```

  Rdb selects only the first 10 records in the sorted EMPLOYEES relation. See Section 4.6 for more information about the SORTED_BY relation.

- If you do not specify a sort order in the record selection expression, Rdb selects the qualifying records unpredictably and the records returned may change each time you use the FIRST clause. In other words, if you make the same query twice you may not get the same results both times, unless you use the SORTED BY clause.

- If the value expression is not an integer, Rdb truncates any fractional part of the value and uses the remaining integer as the number of records in the record stream. For example, a program might prompt a user for a value expression, compute a value, and use it in a record selection expression:

```
VAR productivity : REAL;
                .
                .
                .
WRITE  ('Productivity factor: ');
READLN (productivity);
FOR FIRST productivity E IN EMPLOYEES
   WRITELN (E.LAST_NAME, E.EMPLOYEE_ID);
END_FOR;
```

  Assume here that the value of PRODUCTIVITY is 2.5. Rdb performs all subsequent actions to the first two records retrieved in the FOR loop.

- The value expression cannot contain a database field unless you take one of the following actions:

  - Define a context variable in an outer loop, such as:

```
FOR E IN EMPLOYEES
    FOR FIRST E.EMPLOYEE_ID SH IN SALARY_HISTORY
    WITH SH.EMPLOYEE_ID = E.EMPLOYEE_ID
```

    —   Use a self-contained expression, such as:

```
FOR FIRST (COUNT OF E IN EMPLOYEES
           WITH E.STATE = "MA")
SH IN SALARY_HISTORY
```

See Section 2.2 for more information. Also refer to Chapter 5, which documents statistical functions.

Examples

Example 1

The following programs demonstrate the use of the FIRST clause and the SORTED BY clause. These programs sort the EMPLOYEES relation in ascending order based on the EMPLOYEE_ID field. The FIRST 50 statement creates a record stream that contains the first 50 records from the sorted EMPLOYEES relation. The programs then print the employee IDs and last names of these 50 employee records.

C Program

```
#include <stdio.h>
DATABASE PERS = FILENAME "PERSONNEL";

main ( )
{
READY PERS;
START_TRANSACTION READ_ONLY;

FOR FIRST 50 E IN EMPLOYEES
   SORTED BY E.EMPLOYEE_ID
      printf ("%s  ",E.EMPLOYEE_ID);
      printf ("%s\n",E.LAST_NAME);
END_FOR;

COMMIT;
FINISH;
}
```

Pascal Program

```
program first_clause (input,output);
DATABASE PERS = FILENAME 'PERSONNEL';

begin
READY PERS;
START_TRANSACTION READ_ONLY;
```

FIRST Clause

```
FOR FIRST 50 E IN EMPLOYEES
   SORTED BY E.EMPLOYEE_ID
      writeln (E.EMPLOYEE_ID, ' ', E.LAST_NAME);
END_FOR;

COMMIT;
FINISH;
end.
```

Example 2

The following programs demonstrate the use of the FIRST clause and the
SORTED BY clause with two sort keys. These programs sort the COLLEGES
relation in ascending order on the basis of the STATE and CITY fields.
Because STATE is the first sort key, records are sorted by state first. Then
the records are sorted by city within each state. The FIRST clause selects the
first 10 records from the sorted relation. These programs then print the college
name, city, and state of each of these 10 records.

C Program

```
#include <stdio.h>
DATABASE PERS = FILENAME "PERSONNEL";

main ()
{
READY PERS;
START_TRANSACTION READ_ONLY;

FOR FIRST 10 C IN COLLEGES
   SORTED BY C.STATE, C.CITY
      printf("%s %s %s\n", C.COLLEGE_NAME, C.CITY, C.STATE);
END_FOR;

COMMIT;
FINISH;
}
```

Pascal Program

```
program first_sorted (input,output);
DATABASE PERS = FILENAME 'PERSONNEL';

begin
READY PERS;
START_TRANSACTION READ_ONLY;

FOR FIRST 10 C IN COLLEGES
   SORTED BY C.STATE, C.CITY
      writeln (C.COLLEGE_NAME, ' ', C.CITY,' ', C.STATE);
END_FOR;
```

```
COMMIT;
FINISH;
end.
```

Example 3

The following programs demonstrate the use of a host language variable
and the FIRST clause. The programs obtain the value for the host language
variable, *how_many*, through interactive programming.

The C program uses the function read_int() to prompt for and store the value
for the host language variable. For more information and the source code for
read_int, see Appendix B. The Pascal program uses the readln and writeln
statements to serve a similar function. By doing the interactive processing
before attaching to the database, these programs keep the transaction as short
as possible.

The SORTED BY clause sorts the EMPLOYEES relation in ascending order,
based on the EMPLOYEE_ID field. The value for *how_many* determines
the number of records the FIRST clause selects from the sorted relation.
The selection process begins with the first record in the sorted relation and
continues selecting records until the specified number have been selected.
These programs print the employee IDs, first names, and last names from
these records.

C Program

```
#include <stdio.h>
DATABASE PERS = FILENAME "PERSONNEL";

extern int read_int();

main ( )
{
int how_many;
how_many = read_int("Enter number of employees to display: ");

READY PERS;
START_TRANSACTION READ_ONLY;

FOR FIRST how_many E IN EMPLOYEES
   SORTED BY E.EMPLOYEE_ID
      printf("%s  %s  %s\n", E.EMPLOYEE_ID, E.FIRST_NAME, E.LAST_NAME);
END_FOR;

COMMIT;
FINISH;
}
```

FIRST Clause

Pascal Program

```
program first_with_host (input,output);
DATABASE PERS = FILENAME 'PERSONNEL';

var
   how_many : integer;

begin
write  ('Enter number of employees to display: ');
readln (how_many);

READY PERS;
START_TRANSACTION READ_ONLY;

FOR FIRST how_many E IN EMPLOYEES
   SORTED BY E.EMPLOYEE_ID
      writeln (E.EMPLOYEE_ID, ' ', E.FIRST_NAME, ' ', E.LAST_NAME);
END_FOR;

COMMIT;
FINISH;
end.
```

Example 4

The following programs demonstrate the use of the FIRST clause with an arithmetic operator. These programs sort the records in the CURRENT_SALARY relation in descending order of salary amount. The FIRST clause selects the first quarter of the total number of the sorted CURRENT_SALARY records. These programs then print the last name of the employees in the selected records and the number of records that the FIRST clause selected.

C Program

```
#include <stdio.h>
DATABASE PERS = FILENAME "PERSONNEL";

main()
{
READY PERS;
START_TRANSACTION READ_ONLY;

FOR FIRST (0.25 * COUNT OF CS IN CURRENT_SALARY) EMP IN CURRENT_SALARY
   SORTED BY DESCENDING EMP.SALARY_AMOUNT
      printf ("%s %d\n",EMP.LAST_NAME,
                        COUNT OF SH IN SALARY_HISTORY
                         WITH SH.EMPLOYEE_ID = EMP.EMPLOYEE_ID);
END_FOR;

COMMIT;
FINISH;
}
```

FIRST Clause

Pascal Program

```
program first_with_stat (input,output);
DATABASE PERS = FILENAME 'PERSONNEL';

begin
READY PERS;
START_TRANSACTION READ_ONLY;

FOR FIRST (0.25 * COUNT OF CS IN CURRENT_SALARY) EMP IN CURRENT_SALARY
    SORTED BY DESCENDING EMP.SALARY_AMOUNT
        writeln (EMP.LAST_NAME, ' ', COUNT OF SH IN SALARY_HISTORY
                                    WITH SH.EMPLOYEE_ID = EMP.EMPLOYEE_ID);
END_FOR;

COMMIT;
FINISH;
end.
```

4.4 REDUCED TO Clause

The REDUCED TO clause of the record selection expression lets you eliminate duplicate values for fields in a record stream. You can use this expression to eliminate redundancy in the results of a query and to group the records in a relation according to unique field values.

Format

reduce-clause =

$$\longrightarrow \underline{\text{REDUCED}}\ \text{TO}\ \longrightarrow\ \text{db-field}\ \longrightarrow$$
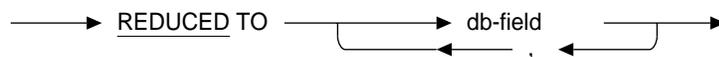
Argument

db-field
A database field value expression. A database field value expression is a field name qualified with a context variable. The database field must be defined in the current context of the statement. For example, the database field must be defined in the context of the START_STREAM or DECLARE_STREAM statements or the FOR loop. See Section 2.2 for more information.

Usage Notes

■ The use of the SORTED BY clause with the REDUCED TO clause is highly recommended. Without it, you cannot be sure of the order in which records will be retrieved. If you do not use a SORTED BY clause in the record selection expression, Rdb selects the qualifying records unpredictably and the records returned may change. In other words, if you make this query twice you may not get the same results both times, unless you use the SORTED BY clause.

■ If you use the REDUCED TO clause, do not retrieve any fields that you do not specify in the list of db-fields within the record selection expression. If you retrieve other fields, the results are unpredictable. In other words, it is not useful to have a REDUCED TO clause in this case:

```
FOR SH IN SALARY_HISTORY REDUCED TO SH.EMPLOYEE_ID
     SORTED BY SH.EMPLOYEE_ID

   WRITELN (SH.LAST_NAME,FIRST_NAME);
END_FOR;
```

The preceding example reduces the record stream from the SALARY_
HISTORY relation to a record stream that consists of a list of unique
employee identification numbers. If you want to display fields other than
EMPLOYEE_ID, you should include additional REDUCED TO fields.

■   In general, the more REDUCED TO keys you use, the more records you
    retrieve.

    For example, if Ingrid Smith and William Smith are both employees
    with records in the EMPLOYEES relation, the following record selection
    expression will retrieve one record, while the succeeding record selection
    expression will retrieve two:

```
FOR E IN EMPLOYEES
   REDUCED TO E.LAST_NAME
   writeln (E.LAST_NAME);
END_FOR;

FOR E IN EMPLOYEES
   REDUCED TO E.LAST_NAME, E.FIRST_NAME
     writeln (E.LAST_NAME);
END_FOR;
```

■   If you use a statistical function (for example, COUNT) with the REDUCED
    TO clause, embed it in a GET statement. A statement embedded in the
    GET statement incurs less overhead than a statistical function embedded
    directly in the host language.

Examples

Example 1

The following programs demonstrate the use of the REDUCED TO clause and
the SORTED BY clause with a single relation. These programs sort the records
in the EMPLOYEES relation on the basis of the STATE field. The REDUCED
TO clause limits the record stream so that each record in the stream has a
different value for the STATE field. These programs then display the list of
states represented in the EMPLOYEES relation.

REDUCED TO Clause

C Program

```
#include <stdio.h>
DATABASE PERS = FILENAME "PERSONNEL";

main ()
{
READY PERS;
START_TRANSACTION READ_ONLY;

FOR E IN EMPLOYEES
   REDUCED TO E.STATE
   SORTED BY E.STATE
      printf("%s\n", E.STATE);
END_FOR;

COMMIT;
FINISH;
}
```

Pascal Program

```
program reduced_one_rel (input,output);
DATABASE PERS = FILENAME 'PERSONNEL';

begin
READY PERS;
START_TRANSACTION READ_ONLY;

FOR E IN EMPLOYEES
   REDUCED TO E.STATE
   SORTED BY E.STATE
      writeln (E.STATE);
END_FOR;

COMMIT;
FINISH;
end.
```

Example 2

The following programs demonstrate the use of the REDUCED TO clause and the SORTED BY clause with multiple relations. These programs:

- Print an informational message

- Cross CURRENT_JOB and DEGREES relations over the common EMPLOYEE_ID field

- Limit the record stream to those records in the DEGREES relation that have the same employee ID as the records in the CURRENT_JOB relation with the department code of "ENG" (engineer)

REDUCED TO Clause

- Sort the records in the stream in ascending order based on the COLLEGE_ CODE field and within each college code, sort by DEGREE (also in ascending order)

- Reduce the record stream to those records that have unique combinations of college code and degree

- Print the unique combinations of the COLLEGE_CODE and DEGREE fields for engineers

C Program

```
#include <stdio.h>
DATABASE PERS = FILENAME "PERSONNEL";

main()
{
printf ("List unique combinations of COLLEGE CODE and");
printf (" DEGREE for all engineers");

READY PERS;
START_TRANSACTION READ_ONLY;

FOR CJ IN CURRENT_JOB
   CROSS D IN DEGREES
   OVER EMPLOYEE_ID
   WITH CJ.DEPARTMENT_CODE = "ENG"
   REDUCED TO D.COLLEGE_CODE, D.DEGREE
   SORTED BY D.COLLEGE_CODE, D.DEGREE
      printf ("%s  %s", D.COLLEGE_CODE, D.DEGREE);
END_FOR;

COMMIT;
FINISH;
}
```

Pascal Program

```
program reduced_clause (input,output);
DATABASE PERS = FILENAME 'PERSONNEL';

begin
writeln ('List unique combinations of COLLEGE_CODE and ',
         'DEGREE for all engineers');

READY PERS;
START_TRANSACTION READ_ONLY;
```

REDUCED TO Clause

```
FOR CJ IN CURRENT_JOB
   CROSS D IN DEGREES
   OVER EMPLOYEE_ID
   WITH CJ.DEPARTMENT_CODE = 'ENG'
   REDUCED TO D.COLLEGE_CODE, D.DEGREE
   SORTED BY D.COLLEGE_CODE, D.DEGREE
      writeln (D.COLLEGE_CODE, ' ', D.DEGREE);
END_FOR;

COMMIT;
FINISH;
end.
```

Example 3

The following programs demonstrate the use of the REDUCED TO clause with a reflexive join. These programs:

- Limit the record stream to those records in the DEGREES relation with a degree starting with "M" (master's) or containing "D" (doctorate)

- Sort the records by descending EMPLOYEE_ID

- Further limit the record stream to those records with unique employee IDs

- Print an informational message and the employee ID of those employees with either a master's or doctorate degree, or both

C Program

```
#include <stdio.h>
DATABASE PERS = FILENAME "PERSONNEL";

main ()
{
READY PERS;
START_TRANSACTION READ_ONLY;

FOR D IN DEGREES
   WITH D.DEGREE STARTING WITH "M"
   OR D.DEGREE CONTAINING "D"
   REDUCED TO D.EMPLOYEE_ID
   SORTED BY DESCENDING D.EMPLOYEE_ID
      printf("%s has an advanced degree.\n", D.EMPLOYEE_ID);
END_FOR;

COMMIT;
FINISH;
}
```

REDUCED TO Clause

Pascal Program

```
program reduced_one_relation (input,output);
DATABASE PERS = FILENAME 'PERSONNEL';

begin
READY PERS;
START_TRANSACTION READ_ONLY;

FOR D IN DEGREES
   WITH D.DEGREE STARTING WITH 'M'
   OR D.DEGREE CONTAINING 'D'
   REDUCED TO D.EMPLOYEE_ID
   SORTED BY DESCENDING D.EMPLOYEE_ID
      writeln (D.EMPLOYEE_ID, ' has an advanced degree.');
END_FOR;

COMMIT;
FINISH;
end.
```

4.5 Relation Clause

The record selection expression's relation clause lets you declare a context variable for a stream or a loop. Once you associate a context variable with a relation, you can use only that context variable to refer to records from that relation in the record stream you created. Each relation (including multiple uses of the same relation) in the record stream must have a unique context variable. See Section 4.1 for more information.

Format

relation-clause =

```
───▶ context-var ───▶ IN ──┬──────────────────────┬──▶ relation-name ───▶
                            └──▶ db-handle ──▶ . ───┘
```

Arguments

context-var
A context variable. You define a context variable in a relation clause. See Section 4.1 for more information.

db-handle
Database handle. A host language variable used to refer to a specific database you have invoked. If you do not supply a database handle, a default database handle is declared for you by RDML. However, if you are using more than one database in your program, you should declare database handles for all the databases.

relation-name
The name of a relation in a database.

Usage Notes

- You must use a relation clause in the following RDML statements and functions:

  - DECLARE_STREAM

  - FOR

  - START_STREAM (Declared and Undeclared)

  - STORE

  - DECLARE_STREAM

  - Statistical and Boolean functions

- You must associate a different context variable with each relation you refer to in the same query. If you access several relations at once, the context variable lets you distinguish between fields from different relations within the same statements.

- Once you associate a context variable with a relation, you must use it in other statements to qualify field names. For instance, once you declare a context variable in a FOR statement, you must use it in other statements within the FOR . . . END_FOR block (for example, a MODIFY statement) to qualify field names.

Examples

Example 1

The following programs demonstrate the use of the relation clause with a FOR loop. These programs declare a context variable E for EMPLOYEES. This allows the programs to refer to records from the EMPLOYEES relation by using the context variable E in the host language print statements.

C Program

```
#include <stdio.h>
DATABASE PERS = FILENAME "PERSONNEL";

main()
{
READY PERS;
START_TRANSACTION READ_ONLY;
```

Relation Clause

```
FOR E IN EMPLOYEES
   printf ("%s  %s  %s\n", E.LAST_NAME,
                          E.EMPLOYEE_ID,
                          E.SEX);
END_FOR;

COMMIT;
FINISH;
}
```

Pascal Program

```
program context_variable (input,output);
DATABASE PERS = FILENAME 'PERSONNEL';

begin
READY PERS;
START_TRANSACTION READ_ONLY;

FOR E IN EMPLOYEES
   writeln (E.LAST_NAME, ' ', E.EMPLOYEE_ID, ' ', E.SEX);
END_FOR;

COMMIT;
FINISH;
end.
```

Example 2

The following programs demonstrate the use of a relation clause with a STORE statement.

The C program uses the function pad_string to append trailing blanks to the last name. This ensures that the last name matches the length defined for the field. For more information and the source code for pad_string, see Appendix B. Pascal does not require a special function to pad strings; the Pascal writeln function pads strings for you.

These programs declare a context variable C for the COLLEGES relation. This allows the programs to refer to the fields in the COLLEGES relation with the context variable C during the STORE operation.

C Program

```
#include <stdio.h>
DATABASE PERS = FILENAME "PERSONNEL";

extern void pad_string();

main()
{
READY PERS;
START_TRANSACTION READ_WRITE;
```

```
STORE C IN COLLEGES USING
   pad_string("PURD", C.COLLEGE_CODE, sizeof(C.COLLEGE_CODE));
   pad_string("Purdue University",C.COLLEGE_NAME, sizeof(C.COLLEGE_NAME));
   pad_string("West Lafayette", C.CITY, sizeof(C.CITY));
   pad_string("IN", C.STATE, sizeof(C.STATE));
   pad_string("01760", C.POSTAL_CODE, sizeof(C.POSTAL_CODE));
END_STORE;

ROLLBACK;
FINISH;
}
```

Pascal Program

```
program context_store (input,output);
DATABASE PERS = FILENAME 'PERSONNEL';

begin
READY PERS;
START_TRANSACTION READ_WRITE;

STORE C IN COLLEGES USING
   C.COLLEGE_CODE := 'PURD';
   C.COLLEGE_NAME := 'Purdue University';
   C.CITY := 'West Lafayette';
   C.STATE := 'IN';
   C.POSTAL_CODE := '01760';
END_STORE;

ROLLBACK;
FINISH;
end.
```

Example 3

The following programs demonstrate the use of the relation clause with the
START_STREAM statement and the FETCH statement. The START_STREAM
statement declares and opens the record stream, EMP_STREAM. The FOR
statement determines the records to be included in the stream. These records
are all the records in the EMPLOYEES relation sorted in descending order,
based on the employee ID. The FETCH statement advances an internal pointer
to the first record in the record stream, gets the record, and the programs print
the last name of this employee.

C Program

```
#include <stdio.h>
DATABASE PERS = FILENAME "PERSONNEL";

main()
{
READY PERS;
START_TRANSACTION READ_ONLY;
```

Relation Clause

```
START_STREAM EMP_STREAM USING E IN EMPLOYEES
    SORTED BY DESCENDING E.EMPLOYEE_ID, E.LAST_NAME;
    FETCH EMP_STREAM;
        printf("%s has the largest badge number\n", E.LAST_NAME);
END_STREAM EMP_STREAM;

COMMIT;
FINISH;
}
```

Pascal Program

```
program stream (input,output);
DATABASE PERS = FILENAME 'PERSONNEL';

begin
READY PERS;
START_TRANSACTION READ_ONLY;

START_STREAM EMP_STREAM USING E IN EMPLOYEES
    SORTED BY DESCENDING E.EMPLOYEE_ID, E.LAST_NAME;
    FETCH EMP_STREAM;
        writeln (E.LAST_NAME, ' has the largest badge number');
END_STREAM EMP_STREAM;

COMMIT;
FINISH;
end.
```

Example 4

The following programs demonstrate the use of the relation clause to qualify fields with the same names from different relations. The programs:

- Join the EMPLOYEES relation and the DEGREES relation over the common EMPLOYEE_ID field

- Join the COLLEGES relation with the DEGREES relation over the common COLLEGE_CODE field

The joins create a record stream that contains records from the EMPLOYEES, DEGREES and COLLEGES relations. The SORTED BY clause sorts the records in ascending order, based on the COLLEGE_CODE, DEGREE, DEGREE FIELD, and EMPLOYEE_ID fields. Note that these fields are contained in more than one relation. The programs use the relation clause to qualify from which relation the program must obtain a specified field value.

These programs print a report using fields from all three relations.

C Program

```
#include <stdio.h>
DATABASE PERS = FILENAME "PERSONNEL";

main ()
{
READY PERS;
START_TRANSACTION READ_ONLY;

FOR E IN EMPLOYEES
   CROSS D IN DEGREES OVER EMPLOYEE_ID
   CROSS C IN COLLEGES OVER COLLEGE_CODE
   SORTED BY D.COLLEGE_CODE, D.DEGREE, D.DEGREE_FIELD, E.EMPLOYEE_ID;
      printf ("%s %s %s %d %s %s\n", C.COLLEGE_NAME,
                                     D.DEGREE,
                                     D.DEGREE_FIELD,
                                     D.YEAR_GIVEN,
                                     E.EMPLOYEE_ID,
                                     E.LAST_NAME);
 END_FOR;

COMMIT;
FINISH;
}
```

Pascal Program

```
program qualify_fields (input,output);
DATABASE PERS = FILENAME 'PERSONNEL';

begin
READY PERS;
START_TRANSACTION READ_ONLY;

FOR E IN EMPLOYEES
   CROSS D IN DEGREES OVER EMPLOYEE_ID
   CROSS C IN COLLEGES OVER COLLEGE_CODE
   SORTED BY D.COLLEGE_CODE, D.DEGREE, D.DEGREE_FIELD, E.EMPLOYEE_ID
      writeln (C.COLLEGE_NAME, ' ',
               D.DEGREE, ' ',
               D.DEGREE_FIELD, ' ',
               D.YEAR_GIVEN, ' ',
               E.EMPLOYEE_ID, ' ',
               E.LAST_NAME);
END_FOR;

COMMIT;
FINISH;
end.
```

Relation Clause

Example 5

The following programs demonstrate the use of the relation clause in a CROSS clause. These programs:

- Cross the CURRENT_JOB view over itself

- Declare the context variables BOSS and WORKER in a relation clause to qualify two record streams with the same field, LAST_NAME

- Display the bosses' and the workers' names

C Program

```c
#include <stdio.h>
DATABASE PERS = FILENAME "PERSONNEL";

main()
{
READY PERS;
START_TRANSACTION READ_ONLY;

FOR BOSS IN CURRENT_JOB
   CROSS WORKER IN CURRENT_JOB
   WITH BOSS.EMPLOYEE_ID = WORKER.SUPERVISOR_ID
   SORTED BY BOSS.LAST_NAME, WORKER.LAST_NAME
      printf ("Boss:  %s ", BOSS.LAST_NAME);
      printf ("Worker:  %s\n", WORKER.LAST_NAME);
END_FOR;

COMMIT;
FINISH;
}
```

Relation Clause

Pascal Program

```
program qualify_same_field (input,output);
DATABASE PERS = FILENAME 'PERSONNEL';

begin
READY PERS;
START_TRANSACTION READ_ONLY;

FOR BOSS IN CURRENT_JOB
   CROSS WORKER IN CURRENT_JOB
   WITH BOSS.EMPLOYEE_ID = WORKER.SUPERVISOR_ID
   SORTED BY BOSS.LAST_NAME, WORKER.LAST_NAME
      writeln ('Boss: ', BOSS.LAST_NAME, ' ',
               'Worker: ', WORKER.LAST_NAME);
END_FOR;

COMMIT;
FINISH;
end.
```

4.6 SORTED BY Clause

The SORTED BY clause of the record selection expression lets you sort the records in the record stream by the values of specific fields. You sort on a database field value expression, called a **sort key**. The sort key determines the order in which Rdb returns the records in the record stream.

Format

sort-clause =

```
        SORTED BY                        db-field
                        ASCENDING
                        DESCENDING
                              ,
```
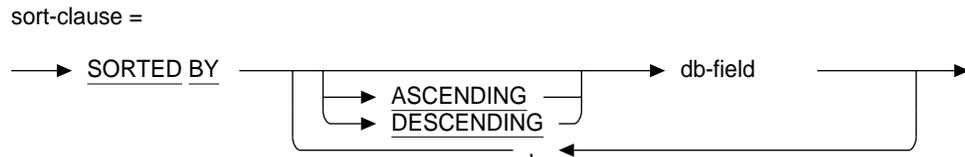
Arguments

ASCENDING
The default sorting order. Rdb sorts the records in ascending order ("A" precedes "B", 1 precedes 2, and so on). Missing values appear as the last items in this list of sorted values. You can abbreviate the ASCENDING keyword to ASC.

DESCENDING
Rdb sorts strings in ASCII sequence, and numbers in numeric order. ("A" follows "B", 1 follows 2, and so on). Missing values appear as the first items in this list of sorted values. You can abbreviate the DESCENDING keyword to DESC.

db-field
A database field value expression within the query. A database field value expression is a field name qualified with a context variable. See Section 2.2 for more information.

Usage Notes

- The sort order for strings is by byte value (ASCII). This order treats uppercase characters as being greater than lowercase characters. When you specify the sort order to be ascending, strings beginning with lowercase characters will appear after strings beginning with uppercase letters.

- The value expression is the sort key. For example, the following FOR statement sorts employees by last name:

```
FOR E IN EMPLOYEES SORTED BY E.LAST_NAME
                .
                .
                .
END_FOR;
```

- If you do not specify the sorting order with the first sort key, the default order is ascending. In the following example, because the sorting order is not specified, Rdb automatically sorts the EMPLOYEES records in ascending order by EMPLOYEE_ID.

```
FOR E IN EMPLOYEES SORTED BY E.EMPLOYEE_ID
                .
                .
                .
END_FOR;
```

- If you do not specify ASCENDING or DESCENDING for the second or subsequent sort keys, Rdb uses the order you specified for the preceding sort key. In the example that follows, Rdb sorts both the EMPLOYEE_ID and JOB_CODE fields in descending order. The sort order for the EMPLOYEE_ID and SUPERVISOR_ID fields is explicit; Rdb automatically determines the sort order for the JOB_CODE field by the preceding sort key (DESCENDING E.EMPLOYEE_ID).

```
FOR E IN EMPLOYEES
    CROSS JH IN JOB_HISTORY OVER EMPLOYEE_ID
    SORTED BY DESCENDING E.EMPLOYEE_ID, JH.JOB_CODE,
          ASCENDING  JH.SUPERVISOR_ID
                .
                .
                .
END_FOR;
```

SORTED BY Clause

- When you use multiple sort keys, Rdb treats the first field or value expression in the list of sort keys as the major sort key and successive field or value expressions as minor sort keys. That is, Rdb first sorts the records into groups based on the first field or value expression. Then Rdb uses the second key to sort the records within each group, and so on.

- Missing values always sort as the highest items in a sorted list. They are the first items in a list sorted in descending order, and the last items in a list sorted in ascending order.

Examples

Example 1

The following programs demonstrate the use of the SORTED BY clause using the default sort order, ascending. The programs:

- Sort the records in CURRENT_INFO

- Sort in ascending order because no sort order is specified

- Print the last names and salaries stored in the sorted records

C Program

```
#include <stdio.h>
DATABASE PERS = FILENAME "PERSONNEL";

main()
{
READY PERS;
START_TRANSACTION READ_ONLY;

FOR CI IN CURRENT_INFO
   SORTED BY CI.SALARY
      printf ("%s $%f\n",CI.LAST_NAME, CI.SALARY);
END_FOR;

COMMIT;
FINISH;
}
```

Pascal Program

```
program sort_single_field (input,output);
DATABASE PERS = FILENAME 'PERSONNEL';

begin
READY PERS;
START_TRANSACTION READ_ONLY;

FOR CI IN CURRENT_INFO
   SORTED BY CI.SALARY
      writeln (CI.LAST_NAME, ' $', CI.SALARY :10:2);
END_FOR;

COMMIT;
FINISH;
end.
```

Example 2

The following programs demonstrate the use of the SORTED BY clause to arrange records in descending order. The programs:

- Arrange the records in the JOBS relation in descending order on the basis of the MAXIMUM_SALARY field

- Print the JOB_CODE, MAXIMUM_SALARY, and MINIMUM_SALARY fields from the sorted list

C Program

```
#include <stdio.h>
DATABASE PERS = FILENAME "PERSONNEL";

main()
{
READY PERS;
START_TRANSACTION READ_ONLY;

FOR J IN JOBS
   SORTED BY DESCENDING J.MAXIMUM_SALARY
      printf ("%s $%f $%f\n ", J.JOB_CODE,
                               J.MAXIMUM_SALARY,
                               J.MINIMUM_SALARY);

END_FOR;

COMMIT;
FINISH;
}
```

SORTED BY Clause

Pascal Program

```
program sort_descending (input,output);
DATABASE PERS = FILENAME 'PERSONNEL';

begin
READY PERS;
START_TRANSACTION READ_ONLY;

FOR J IN JOBS
   SORTED BY DESCENDING J.MAXIMUM_SALARY
      writeln (J.JOB_CODE,
               ' $', J.MAXIMUM_SALARY : 10 : 2,
               ' $', J.MINIMUM_SALARY : 10 : 2);
END_FOR;

COMMIT;
FINISH;
end.
```

Example 3

The following programs demonstrate the use of the SORTED BY clause and sort keys. The programs:

- Create a record stream that contains all records in the employees relation

- Sort this record stream in ascending order by state

- Sort by descending city within each state

- Print the states, cities, and employee IDs from the sorted record stream

C Program

```
#include <stdio.h>
DATABASE PERS = FILENAME "PERSONNEL";

main()
{
READY PERS;
START_TRANSACTION READ_ONLY;

FOR E IN EMPLOYEES
   SORTED BY ASCENDING E.STATE,
   DESCENDING E.CITY
      printf ("%s  %s  %s\n", E.STATE, E.CITY, E.EMPLOYEE_ID);
END_FOR;

COMMIT;
FINISH;
}
```

Pascal Program

```
program matching_all (input,output);
DATABASE PERS = FILENAME 'PERSONNEL';

begin
READY PERS;
START_TRANSACTION READ_ONLY;

FOR E IN EMPLOYEES
   SORTED BY ASCENDING E.STATE,
   DESCENDING E.CITY
      writeln ( E.STATE, ' ', E.CITY,' ', E.EMPLOYEE_ID);
END_FOR;

COMMIT;
FINISH;
end.
```

4.7 WITH Clause

The record selection expression's WITH clause contains a conditional expression that allows you to specify conditions that must be true for a record to be included in a record stream.

Format

with-clause =

⟶ WITH ⟶ conditional-expr ⟶

Argument

conditional-expr
Conditional expression. An expression that evaluates to true or false. See Chapter 3 for more information.

Usage Notes

- A record becomes part of a record stream only when its values satisfy the conditions you specified in the conditional expression (that is, only when the conditional expression evaluates to true).

- If the conditional expression evaluates to false or missing for a record, that record is not included in the record stream.

Examples

Example 1

The following programs demonstrate the use of the WITH clause in a record selection expression. The programs:

- Create a record stream of all those records in the EMPLOYEES relation with an employee ID of 00169

- Print the employee IDs and last names from each record in the record stream

C Program

```
#include <stdio.h>
DATABASE PERS = FILENAME "PERSONNEL";

main()
{
READY PERS;
START_TRANSACTION READ_ONLY;

FOR E IN EMPLOYEES
   WITH E.EMPLOYEE_ID = "00169"
      printf ("%s  ", E.EMPLOYEE_ID);
      printf ("%s", E.LAST_NAME);
END_FOR;

COMMIT;
FINISH;
}
```

Pascal Program

```
program with_clause (input,output);
DATABASE PERS = FILENAME 'PERSONNEL';

begin
READY PERS;
START_TRANSACTION READ_ONLY;

FOR E IN EMPLOYEES
   WITH E.EMPLOYEE_ID = '00169'
      writeln (E.EMPLOYEE_ID, ' ', E.LAST_NAME);
END_FOR;

COMMIT;
FINISH;
end.
```

Example 2

The following programs demonstrate the use of the WITH clause with multiple conditions. The record selection expression finds all employees who have only one degree. The record selection expression limits the stream further by specifying that these employees must have received their degrees from Stanford University in the field of Arts. These programs print the employee ID of the employees who fit these conditions.

WITH Clause

C Program

```c
#include <stdio.h>
DATABASE PERS = FILENAME "PERSONNEL";

main()
{
READY PERS;
START_TRANSACTION READ_ONLY;

FOR E IN EMPLOYEES CROSS D1 IN DEGREES OVER EMPLOYEE_ID
   WITH (UNIQUE D2 IN DEGREES WITH D2.EMPLOYEE_ID = E.EMPLOYEE_ID)
   AND D1.DEGREE_FIELD = "Arts"
   AND D1.COLLEGE_CODE = "STAN"
      printf ("%s\n", E.EMPLOYEE_ID);
END_FOR;

COMMIT;
FINISH;
}
```

Pascal Program

```pascal
program multiple_cond (input,output);
DATABASE PERS = FILENAME 'PERSONNEL';

begin
READY PERS;
START_TRANSACTION READ_ONLY;

FOR E IN EMPLOYEES CROSS D1 IN DEGREES OVER EMPLOYEE_ID
   WITH (UNIQUE D2 IN DEGREES WITH D2.EMPLOYEE_ID = E.EMPLOYEE_ID)
   AND D1.DEGREE_FIELD = 'Arts'
   AND D1.COLLEGE_CODE = 'STAN'
      writeln (E.EMPLOYEE_ID);
END_FOR;

COMMIT;
FINISH;
end.
```

5

RDML Statistical Functions

This chapter describes the Relational Data Manipulation Language (RDML) statistical functions that can be used with embedded RDML statements in C and Pascal programs.

The C and Pascal programs in this chapter access the sample personnel database available with Rdb/VMS.

Statistical functions calculate values based on a value expression for every record in a record stream. Expressions that use statistical functions are sometimes called aggregate expressions, because they calculate a single value for a collection of records. When you use a statistical function you specify a value expression (except for COUNT), and a record selection expression (RSE). Rdb/VMS then performs the following steps:

1. Evaluates the value expression for each record in the record stream formed by the record selection expression

2. Calculates a single value based on the results of the first step

You can also use a value expression to group records within a relation and then use a statistical function to calculate a single value for the group. This operation is often called a **global aggregate** because you can group records by a value in any relation in a database. For example, you can use the DEPARTMENT_CODE field in the DEPARTMENTS relation to group records in the SALARY_HISTORY relation in order to get the average salary for each department.

The following syntax diagram shows the syntax for all the statistical functions. Refer to the section on each function in this chapter for additional information.

Format

statistical-expr =
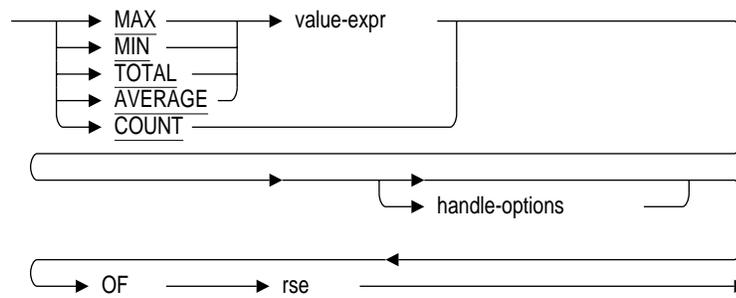
handle-options =
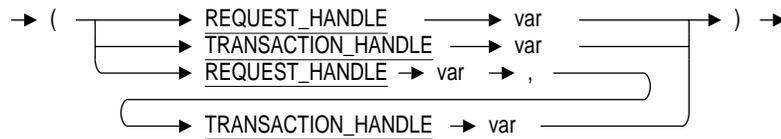
Arguments

handle-options
A transaction handle, a request handle, or both.

value-expr
A value expression. A symbol or a string of symbols used to calculate a value. When you use a value expression in a statement, Rdb calculates the value associated with the expression and uses that value when executing the statement.

rse
A record selection expression. A phrase that defines specific conditions that individual records must meet before Rdb includes them in a record stream.

Table 5–1 summarizes the values returned by statistical functions.

Table 5–1   Statistical Functions

Statistical Function	Value of Statistical Function
AVERAGE	Arithmetic mean of values specified by value expression for all records indicated by record selection expression. Value expression must be numeric data.
COUNT	Number of records in stream specified by record selection expression.
MAX	Largest of values specified by value expression for all records indicated by record selection expression.
MIN	Smallest of values specified by value expression for all records indicated by record selection expression.
TOTAL	Sum of values specified by value expression for all records indicated by record selection expression. Value expression must be numeric data.

When RDML returns the results of a statistical function, it may assign a result data type that is different from the field data type referred to in the expression. See Table 5–2 for a summary of these assignments.

Table 5–2   Statistical Expression Data Type Conversions for RDML

Statistical Function	Field Data Type	Result Data Type	C Equivalent	Pascal Equivalent	EPascal Equivalent
MIN, MAX	Any	Same as field	Same as field	Same as field	Same as field
COUNT	Any	LONGWORD	int, long	INTEGER	INTEGER
AVERAGE	Any	F_FLOATING	float	SINGLE, REAL	REAL
TOTAL	Any	G_FLOATING	double	DOUBLE	DOUBLE

The G_floating data types require the use of the /G_FLOATING qualifier at compile time.

5.1 AVERAGE Statistical Function

The AVERAGE statistical function determines the arithmetic mean of values for all records specified by a record selection expression.

Format

Arguments

handle-options
A request handle, a transaction handle, or both.

REQUEST_HANDLE var
A REQUEST_HANDLE keyword followed by a host language variable. A request handle identifies a compiled Rdb/VMS request. If you do not supply a request handle explicitly, RDML generates a unique request handle for the compiled request. See Section 6.20 for more information on request handles.

TRANSACTION_HANDLE var
A TRANSACTION_HANDLE keyword followed by a host language variable. A transaction handle identifies a transaction. If you do not supply a transaction handle explicitly, RDML uses the default transaction handle.

value-expr

A value expression. A symbol or a string of symbols used to calculate a value. When you use a value expression in a statement, Rdb calculates the value associated with the expression and uses that value when executing the statement.

rse

A record selection expression. A phrase that defines specific conditions that individual records must meet before Rdb includes them in a record stream.

Usage Notes

- If a field value in the value expression is missing, Rdb does not include that record in its calculation of the average value.

- You can use the AVERAGE function only with numeric data types. You can find the average of all salaries, but you cannot find the average employee's name.

- If the record stream is empty or all the values in the record stream are missing, the AVERAGE value is zero in the floating-point form: $0.0000000E+00$ if the field is a floating-point data type

Examples

Example 1

The following programs demonstrate the use of the AVERAGE function in a display statement. These programs:

- Use a record selection expression to form a record stream from a view. The record stream consists of the records for which the value in the SALARY field is greater than $50,000.00.

- Calculate the average salary for these selected records.

- Use a GET statement to place the average in a host language variable.

- Print this average.

AVERAGE Statistical Function

C Program

```
#include <stdio.h>
DATABASE PERS = FILENAME "PERSONNEL";

double mean;
main()
{
READY PERS;
START_TRANSACTION READ_ONLY;

GET
 mean = AVERAGE CI.SALARY OF CI IN CURRENT_INFO
        WITH CI.SALARY > 50000.00;
END_GET;

COMMIT;

printf ("Average is: %f\n",mean);

FINISH;
}
```

Pascal Program

```
program average_function (input,output);
DATABASE PERS = FILENAME 'PERSONNEL';

var
mean : double;
begin
READY PERS;
START_TRANSACTION READ_ONLY;

GET
  mean = AVERAGE SH.SALARY_AMOUNT OF SH IN SALARY_HISTORY
        WITH SH.SALARY_AMOUNT > 50000.00;
END_GET;

COMMIT;

writeln ('Average is: ', mean:10:2);

FINISH;
end.
```

Example 2

The following programs demonstrate the use of the AVERAGE function in a record selection expression. These programs:

- Perform a reflexive join on the CURRENT_INFO view so that each record in the view can be compared to all the records in the view

AVERAGE Statistical Function

- Use the AVERAGE function to determine the average salary of the employees in the CURRENT_INFO view

- Compare the value of the SALARY field in each record to this average

- Print the IDs and last names of those employees whose salaries are greater than or equal to this average

C Program

```
#include <stdio.h>
DATABASE PERS = FILENAME "PERSONNEL";

main()
{
READY PERS;
START_TRANSACTION READ_ONLY;

FOR CI1 IN CURRENT_INFO
   WITH CI1.SALARY >= AVERAGE CI2.SALARY OF CI2 IN CURRENT_INFO
      printf ("%s  %s\n", CI1.ID, CI1.LAST_NAME);
END_FOR;

COMMIT;
FINISH;
}
```

Pascal Program

```
program average_with_rse (input,output);
DATABASE PERS = FILENAME 'PERSONNEL';

begin
READY PERS;
START_TRANSACTION READ_ONLY;

FOR CI1 IN CURRENT_INFO
   WITH CI1.SALARY >= AVERAGE CI2.SALARY OF CI2 IN CURRENT_INFO
      writeln (CI1.ID, ' ', CI1.LAST_NAME);
END_FOR;

COMMIT;
FINISH;
end.
```

5.2 COUNT Statistical Function

The COUNT statistical function returns the number of records in a record stream specified by a record selection expression. The COUNT function differs from other statistical functions because it operates on the record stream defined by the record selection expression, rather than on the values in that record stream.

Format

handle-options =

Arguments

handle-options
A request handle, a transaction handle, or both.

_REQUEST_HANDLE var_
A REQUEST_HANDLE keyword followed by a host language variable. A request handle identifies a compiled Rdb/VMS request. If you do not supply a request handle explicitly, RDML generates a unique request handle for the compiled request. See Section 6.20 for more information on request handles.

_TRANSACTION_HANDLE var_
A TRANSACTION_HANDLE keyword followed by a host language variable. A transaction handle identifies a transaction. If you do not supply a transaction handle explicitly, RDML uses the default transaction handle.

rse

A record selection expression. A phrase that defines specific conditions that individual records must meet before Rdb includes them in a record stream. See Chapter 4 for more information.

Usage Notes

- The count equals zero if no records are in the record stream.

- If any field value is missing from a record in the record stream, the COUNT function still includes the record in the record stream because COUNT does not access field values.

- Use the GET statement rather than a host language statement to retrieve a statistical value. The GET statement produces more efficient code than a host language statement. See Example 1.

Examples

Example 1

The following programs demonstrate the use of the COUNT function in a display statement. These programs:

- Use the COUNT function to compute the number of records stored in the EMPLOYEES relation

- Use the GET statement to place the count in a host language variable

- Print an informational message and the computed number of records

C Program

```
#include <stdio.h>
DATABASE PERS = FILENAME "PERSONNEL";

int num;
main()
{
READY PERS;
START_TRANSACTION READ_ONLY;

GET
   num = COUNT OF E IN EMPLOYEES;
END_GET;

printf ("The number of employees is %d", num);
```

COUNT Statistical Function

```
COMMIT;
FINISH;
}
```

Pascal Program

```
program display_count (input,output);
DATABASE PERS = FILENAME 'PERSONNEL';

var
num : integer;

begin
READY PERS;
START_TRANSACTION READ_ONLY;

GET
  num = COUNT OF E IN EMPLOYEES;
END_GET;

writeln ('The number of employees is', num);

COMMIT;
FINISH;
end.
```

Example 2

The following programs demonstrate the use of the COUNT function in a
record selection expression. These programs cross the CURRENT_JOB view
and the DEPARTMENTS relation over the DEPARTMENT_CODE field. The
COUNT function keeps track of how many times the department codes in the
CURRENT_JOBS records match a department code in the DEPARTMENTS
relation. These programs print every department code that has seven or more
employees associated with it.

C Program

```
#include <stdio.h>
DATABASE PERS = FILENAME "PERSONNEL";

main()
{
READY PERS;
START_TRANSACTION READ_ONLY;

printf ("List large departments.");

FOR D IN DEPARTMENTS
   WITH (COUNT OF CJ IN CURRENT_JOB
         WITH CJ.DEPARTMENT_CODE = D.DEPARTMENT_CODE) >= 7
         printf ("%s\n",D.DEPARTMENT_CODE);
END_FOR;
```

```
COMMIT;
FINISH;
}
```

Pascal Program

```
program count_function (input,output);
DATABASE PERS = FILENAME 'PERSONNEL';

begin
READY PERS;
START_TRANSACTION READ_ONLY;

writeln ('List large departments.');

FOR D IN DEPARTMENTS
   WITH (COUNT OF CJ IN CURRENT_JOB
         WITH CJ.DEPARTMENT_CODE = D.DEPARTMENT_CODE) >= 7
         writeln (D.DEPARTMENT_CODE);
END_FOR;

COMMIT;
FINISH;
end.
```

5.3 MAX Statistical Function

The MAX statistical function returns the highest value for a value expression for all records specified by a record selection expression.

Format

Arguments

handle-options
A request handle, a transaction handle, or both.

REQUEST_HANDLE var
A REQUEST_HANDLE keyword followed by a host language variable. A request handle identifies a compiled Rdb/VMS request. If you do not supply a request handle explicitly, RDML generates a unique request handle for the compiled request. See Section 6.20 for more information on request handles.

TRANSACTION_HANDLE var
A TRANSACTION_HANDLE keyword followed by a host language variable. A transaction handle identifies a transaction. If you do not supply a transaction handle explicitly, RDML uses the default transaction handle.

value-expr

A value expression. A symbol or a string of symbols used to calculate a value. When you use a value expression in a statement, Rdb calculates the value associated with the expression and uses that value when executing the statement. See Chapter 2 for more information.

rse

A record selection expression. A phrase that defines specific conditions that individual records must meet before Rdb includes them in a record stream. See Chapter 4 for more information.

Usage Notes

- If a field value is missing from a record, Rdb does not include that record in its calculation of the MAX value.

- If the record stream is empty or all the values in the record stream are missing, the MAX value is:

  - Blanks if the data type of the field is TEXT

  - Zeros in the floating-point form: 0.0000000E+00 if the field is a floating-point data type

  - 17-NOV-1858 00:00:00.00 if the data type of the field is DATE

- The ASCII collating sequence is used to determine the maximum value for TEXT and VARYING STRING. For example, the MAX of "zebra," "bear," and "pelican" is "zebra."

- Date chronology is used to determine the maximum value for the DATE data type. For example, the MAX of 05-NOV-1917, 06-NOV-1917, and 07-NOV-1917 is 07-NOV-1917.

- Dates are stored in the database in encoded binary format. Therefore, when using the MAX function with dates you must be certain that your application converts these dates to a binary format. See Section 5.4 for an example of a date conversion.

MAX Statistical Function

Examples

Example 1

The following programs demonstrate the use of the MAX function in a display statement. These programs:

- Use the MAX function to compute the highest salary stored in the view CURRENT_INFO

- Use the GET statement to place this value in a host language variable

- Print this computed value

C Program

```
#include <stdio.h>
DATABASE PERS = FILENAME "PERSONNEL";

DECLARE_VARIABLE maxi SAME AS PERS.CURRENT_INFO.SALARY;

main()
{
READY PERS;
START_TRANSACTION READ_ONLY;

GET
    maxi = MAX CI.SALARY OF CI IN CURRENT_INFO;
END_GET;

printf ("%f",maxi);
COMMIT;
FINISH;
}
```

Pascal Program

```
program max_function (input,output);
DATABASE PERS = FILENAME 'PERSONNEL';

DECLARE_VARIABLE maxi SAME AS PERS.CURRENT_INFO.SALARY;

begin
READY PERS;
START_TRANSACTION READ_ONLY;

GET
  maxi = MAX CI.SALARY OF CI IN CURRENT_INFO;
END_GET;

writeln (maxi:10:2);
```

```
COMMIT;
FINISH;
end.
```

Example 2

The following programs demonstrate the use of the MAX function in an assignment statement. These programs:

- Declare a host language variable, *latest_degree*

- Use the MAX function to compute the highest number stored in YEAR_ GIVEN in the DEGREES relation

- Use the GET statement to assign this computed value to the host language variable

- Print an informational message and the value computed by the MAX function

C Program

```
#include <stdio.h>
DATABASE PERS = FILENAME 'PERSONNEL';

main()
{
DECLARE_VARIABLE latest_degree SAME AS DEGREES.YEAR_GIVEN;

READY PERS;
START_TRANSACTION READ_ONLY;

GET
   latest_degree = MAX D.YEAR_GIVEN OF D IN DEGREES;
END_GET;

printf ("Latest Degree was awarded in: %d\n", latest_degree);

COMMIT;
FINISH;
}
```

Pascal Program

```
program assignmax (input,output);
DATABASE PERS = FILENAME 'PERSONNEL';

var
DECLARE_VARIABLE latest_degree SAME AS DEGREES.YEAR_GIVEN;

begin
READY PERS;
START_TRANSACTION READ_ONLY;
```

MAX Statistical Function

```
GET
 latest_degree = MAX D.YEAR_GIVEN OF D IN DEGREES;
END_GET;

writeln ('Latest Degree was awarded in: ', latest_degree);

COMMIT;
FINISH;
end.
```

5.4 MIN Statistical Function

The MIN statistical function returns the lowest value for a value expression for all records specified by a record selection expression.

Format

Arguments

handle-options
A request handle, a transaction handle, or both.

REQUEST_HANDLE var
A REQUEST_HANDLE keyword followed by a host language variable. A request handle identifies a compiled Rdb/VMS request. If you do not supply a request handle explicitly, RDML generates a unique request handle for the compiled request. See Section 6.20 for more information on request handles.

TRANSACTION_HANDLE var
A TRANSACTION_HANDLE keyword followed by a host language variable. A transaction handle identifies a transaction. If you do not supply a transaction handle explicitly, RDML uses the default transaction handle.

MIN Statistical Function

value-expr

A value expression. A symbol or a string of symbols used to calculate a value. When you use a value expression in a statement, Rdb calculates the value associated with the expression and uses that value when executing the statement. See Chapter 2 for more information.

rse

A record selection expression. A phrase that defines specific conditions that individual records must meet before Rdb includes them in a record stream. See Chapter 4 for more information.

Usage Notes

- If a field value is missing, Rdb does not include that record in its calculation of the MIN value.

- If the record stream is empty or all the values in the record stream are missing, the MIN value is:

  - Blanks if the data type of the field is TEXT

  - Zeros in the floating-point form: 0.0000000E+00 if the field is a floating-point data type

  - 17-NOV-1858 00:00:00.00 if the data type of the field is DATE

- The ASCII collating sequence is used to determine the minimum value for TEXT and VARYING STRING. For example, the MIN of the fields "zebra," "bear," and "pelican" is "bear."

- Date chronology is used to determine the minimum value for the DATE data type. For example, the MIN of 09-APR-1954, 10-APR-1954, and 11-APR-1954 is 09-APR-1954.

- Dates are stored in the database in encoded binary format. Therefore, when using the MIN function with dates you must be certain that your application converts these dates to a binary format. See Example 1 for an example of a date conversion.

Examples

Example 1

The following programs list the first SALARY_HISTORY record for each
employee, using the MIN function to determine the oldest salary review date.
Note that the SALARY_HISTORY.SALARY_START field is a DATE data
type. In the database, it is stored in encoded binary format. To display it,
the program must convert the retrieved value into an ASCII string. These
programs call the VMS system service routine ASCTIM to perform the
conversion.

Before converting the SALARY_START DATE field, though, the MIN function
is used. The binary value returned by the MIN function is stored temporarily
in a host language variable. This variable is then converted by ASCTIM. This
process is straightforward in Pascal. The C program must define a pointer to
the variable. In C and Pascal, the host language variable is defined using the
DECLARE_VARIABLE clause.

C Program

```
#include <stdio.h>
#include <descrip.h>
#include <ssdef.h>

DATABASE PERS = FILENAME "PERSONNEL";

extern int SYS$ASCTIM ();

main()
{
DECLARE_VARIABLE start_binary_date SAME AS SALARY_HISTORY.SALARY_START;

/* In the following declaration, note one extra space for EOS  */

static $DESCRIPTOR(sal_ascii_date_desc, "dd-mmm-yyyy hh:mm:ss:cc ");

/* SYS$ASCTIM returns len in a 16-bit word  */

short    len;
long     status;

READY PERS;
START_TRANSACTION READ_ONLY;

FOR E IN EMPLOYEES
   WITH (ANY SH IN SALARY_HISTORY
     WITH SH.EMPLOYEE_ID = E.EMPLOYEE_ID)
   SORTED BY E.EMPLOYEE_ID;

   GET
      start_binary_date = MIN SH.SALARY_START OF SH IN SALARY_HISTORY
                          WITH SH.EMPLOYEE_ID = E.EMPLOYEE_ID;
   END_GET;
```

MIN Statistical Function

```
    status = SYS$ASCTIM( &len, &sal_ascii_date_desc, &start_binary_date, 0);
    if (status != SS$_NORMAL)
        {
         printf ( "Date conversion failed\n");
         continue;
        }
/* Ensure that the returned strings are null-terminated,   */
/* so that we may use printf to print them out.           */

    sal_ascii_date_desc.dsc$a_pointer[len - 1] = '\0';

    printf ("%s  %s First Salary Review was: %s\n",
        E.EMPLOYEE_ID,
        E.LAST_NAME,
        sal_ascii_date_desc.dsc$a_pointer);

END_FOR;

COMMIT;
FINISH;
}
```

Pascal Program

```
[inherit ('sys$library:starlet.pen')]

program min_function (input,output);
DATABASE PERS = FILENAME 'PERSONNEL';

var
   DECLARE_VARIABLE SAL_START_DATE SAME AS SALARY_HISTORY.SALARY_START;
   sal_date    : packed array [1..23] of char;
   status      : integer;

begin
READY PERS;
START_TRANSACTION READ_ONLY;

FOR E IN EMPLOYEES
   WITH (ANY SH IN SALARY_HISTORY
   WITH SH.EMPLOYEE_ID = E.EMPLOYEE_ID)
   SORTED BY E.EMPLOYEE_ID
   WRITELN;
   WRITELN (E.EMPLOYEE_ID, ' ', E.LAST_NAME);

   GET
      sal_start_date := MIN SH.SALARY_START OF SH IN SALARY_HISTORY
                           WITH SH.EMPLOYEE_ID = E.EMPLOYEE_ID;
   END_GET;

   status := $ASCTIM( timbuf := sal_date, timadr := sal_start_date);
   if (status <> SS$_NORMAL)
      then writeln (' Date conversion failed')
      else writeln (' First Salary Review was: ',sal_date);

END_FOR;
```

```
COMMIT;
FINISH;
end.
```

Example 2

The following programs demonstrate the use of the MIN function in an assignment statement. These programs:

- Use the MIN function to compute the lowest salary in the existing records of the JOBS relation for which the wage class is "1"

- Use the GET statement to assign this value to a host language variable

- Store a literal value into all fields for a record in the JOBS relation, except the MINIMUM_SALARY field

- Assign the value stored in the host language variable into the record currently being stored

The C program uses the function pad_string to append trailing blanks and the null terminator to the strings being stored. This ensures that the strings match the length defined for the field. For more information and the source code for pad_string, see Appendix B.

C Program

```
#include <stdio.h>
DATABASE PERS = FILENAME "PERSONNEL";

DECLARE_VARIABLE min SAME AS PERS.JOBS.MINIMUM_SALARY;

extern void pad_string();
main()
{
READY PERS;
START_TRANSACTION READ_WRITE;

GET
  min = MIN J2.MINIMUM_SALARY OF J2 IN JOBS
        WITH J2.WAGE_CLASS = "1";
END_GET;

STORE J IN JOBS USING
    pad_string ("SWPR", J.JOB_CODE, sizeof(J.JOB_CODE));
    pad_string ("1", J.WAGE_CLASS, sizeof(J.WAGE_CLASS));
    pad_string ("Sweeper", J.JOB_TITLE, sizeof(J.JOB_TITLE));
    J.MAXIMUM_SALARY = 10000.00;
    J.MINIMUM_SALARY = min;
END_STORE;
```

MIN Statistical Function

```
ROLLBACK;
FINISH;
}
```

Pascal Program

```
program store_with_min (input,output);
DATABASE PERS = FILENAME 'PERSONNEL';

DECLARE_VARIABLE mini SAME AS PERS.JOBS.MINIMUM_SALARY;

begin
READY PERS;
START_TRANSACTION READ_WRITE;

GET
  mini = MIN J2.MINIMUM_SALARY OF J2 IN JOBS
         WITH J2.WAGE_CLASS = '1';
END_GET;

STORE J IN JOBS USING
    J.JOB_CODE := 'SWPR';
    J.WAGE_CLASS := '1';
    J.JOB_TITLE := 'Sweeper';
    J.MINIMUM_SALARY := mini;
    J.MAXIMUM_SALARY := 10000.00;
END_STORE;

ROLLBACK;
FINISH;
end.
```

5.5 TOTAL Statistical Function

The TOTAL statistical function returns the sum of the values specified by a
record selection expression. The value expression must be a numeric data type.

Format
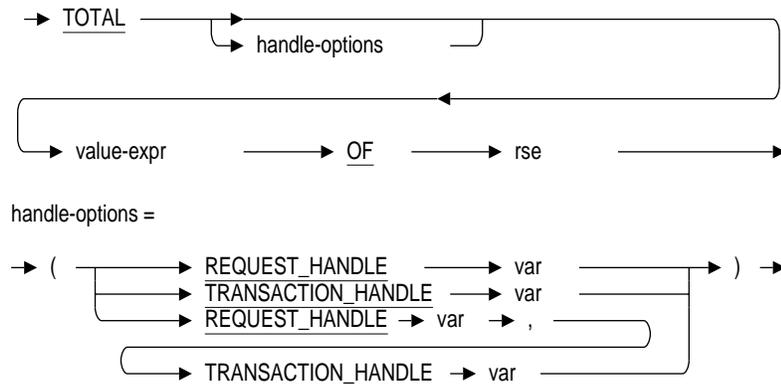
handle-options =

Arguments

handle-options
A request handle, a transaction handle, or both.

REQUEST_HANDLE var
A REQUEST_HANDLE keyword followed by a host language variable. A
request handle identifies a compiled Rdb/VMS request. If you do not supply
a request handle explicitly, RDML generates a unique request handle for the
compiled request. See Section 6.20 for more information on request handles.

TRANSACTION_HANDLE var
A TRANSACTION_HANDLE keyword followed by a host language variable. A
transaction handle identifies a transaction. If you do not supply a transaction
handle explicitly, RDML uses the default transaction handle.

TOTAL Statistical Function

value-expr
A value expression. A symbol or a string of symbols used to calculate a value. When you use a value expression in a statement, Rdb calculates the value associated with the expression and uses that value when executing the statement. See Chapter 2 for more information.

rse
A record selection expression. A phrase that defines specific conditions that individual records must meet before Rdb includes them in a record stream. See Chapter 4 for more information.

Usage Notes

- You can use the TOTAL function only with numeric data types. The value expression that follows the TOTAL function cannot use host variables. You can find the total of all salaries, but you cannot find the total LAST_NAME in a relation.

- The TOTAL value equals zero if no records are in the record stream.

- The TOTAL value equals zero if all values are missing.

- If a field value is missing, Rdb does not include that record in its calculation of the TOTAL value.

Examples

Example 1

The following programs demonstrate the use of the TOTAL function in an assignment statement. These programs:

- Use the TOTAL function to compute the total amount budgeted for all departments in the DEPARTMENTS relation

- Print this computed value

C Program

```
#include <stdio.h>
DATABASE PERS = FILENAME "PERSONNEL";

DECLARE_VARIABLE all SAME AS PERS.DEPARTMENTS.BUDGET_ACTUAL;
main()
{
READY PERS;
START_TRANSACTION READ_ONLY;
```

```
GET
    all = TOTAL D.BUDGET_ACTUAL OF D IN DEPARTMENTS;
END_GET;

printf ("%f", all);

COMMIT;
FINISH;
}
```

Pascal Program

```
program total_function (input,output);
DATABASE PERS = FILENAME 'PERSONNEL';

all : double;

begin
READY PERS;
START_TRANSACTION READ_ONLY;

GET
    all = TOTAL D.BUDGET_ACTUAL OF D IN DEPARTMENTS;
END_GET;

writeln (all:10:2);

COMMIT;
FINISH;
end.
```

Example 2

The following programs demonstrate the use of the TOTAL function in a record selection expression. The programs perform a reflexive join on the CURRENT_INFO view. This results in two record streams, WORKERS and DEPT. The TOTAL function adds the salary of each worker with a common department name and compares the totals for each department with the value $1,000,000,000.00. These programs print an informational message and all the departments that expend $1,000,000,000.00 or more in salaries.

C Program

```
#include <stdio.h>
DATABASE PERS = FILENAME "PERSONNEL";

main()
{
READY PERS;
START_TRANSACTION READ_ONLY;
```

TOTAL Statistical Function

```
FOR DEPT IN CURRENT_INFO
WITH (TOTAL WORKERS.SALARY OF WORKERS IN CURRENT_INFO
      WITH WORKERS.DEPARTMENT = DEPT.DEPARTMENT) >= 1000000000.00
        printf ("Department %s  %s\n",DEPT.DEPARTMENT, "makes large salaries");
END_FOR;

COMMIT;
FINISH;
}
```

Pascal Program

```
program total_function (input,output);
DATABASE PERS = FILENAME 'PERSONNEL';

begin
READY PERS;
START_TRANSACTION READ_ONLY;

FOR DEPT IN CURRENT_INFO
WITH (TOTAL WORKERS.SALARY OF WORKERS IN CURRENT_INFO
      WITH WORKERS.DEPARTMENT = DEPT.DEPARTMENT) >= 1000000000.00
        writeln ('Department ',DEPT.DEPARTMENT,' makes large salaries.');
END_FOR;

COMMIT;
FINISH;
end.
```

6

RDML Clauses and Statements

This chapter describes the Relational Data Manipulation Language (RDML) clauses and statements that can be embedded in C and Pascal programs. These programs can be processed by the RDML preprocessor.

The C and Pascal programs in this chapter access the sample personnel database available with Rdb/VMS.

Table 6–1 summarizes the functions of the statements and clauses in this chapter.

Table 6–1   Functions of RDML Statements and Clauses

Clause or Statement	Function
BASED ON	Extracts the data type and size of a field, allowing you to declare a host languages type: Pascal TYPE(s) and C typedef(s).
COMMIT	Ends a transaction by making permanent all changes performed during that transaction.
DATABASE	Names the database to be accessed in the program module in which this statement appears (does not cause an attach to the database).
Database handle	Specifies a database context to the RDML preprocessor. Necessary when you access two or more databases in the same program.
DECLARE_STREAM	Declares the context of a record stream. Only has meaning when used with the declared START_STREAM statement.

<div align="right">(continued on next page)</div>

Table 6–1 (Cont.)   Functions of RDML Statements and Clauses

Clause or Statement	Function
DECLARE_VARIABLE	Declares a host language variable to have the same data type and size as a specified database field. Has the same function and effects as DEFINE_TYPE clause.
DEFINE_TYPE	Declares a host language variable to have the same data type and size as a specified database field.
END_STREAM, declared	Closes a stream that was previously declared and opened with the declared START_STREAM statement.
END_STREAM, undeclared	Specifies and closes a record stream.
ERASE	Deletes records from a relation in an open stream.
Evaluating clause	Allows you to specify the point at which the named constraints are evaluated.
FETCH	Retrieves the next record from a record stream. The record stream must be started with the DECLARE_STREAM or START_STREAM statement.
FINISH	Explicitly ends your access to a database.
FOR	Executes a statement or group of statements once for each record in a record stream formed by a record selection expression.
FOR statement with segmented strings	Sets up a record stream that consists of segments from a segmented string field. Provides a means for retrieving the segments of a segmented string.
GET	Assigns values to host variables.
MODIFY	Changes the value in one or more fields in one or more records from an open stream.
ON ERROR	Specifies the statement(s) the host language executes if an error occurs during the execution of the associated RDML statement.
PREPARE	Signals to Rdb/ELN that you intend to commit a transaction. Useful only in an Rdb/ELN environment.
READY	Causes an attach to the database(s).

(continued on next page)

Table 6–1 (Cont.)   Functions of RDML Statements and Clauses

Clause or Statement	Function
REQUEST_HANDLE	Identifies a compiled Rdb request. A request handle is a host language variable.
ROLLBACK	Terminates a transaction and undoes all changes made to the database since the start of the transaction.
START_STREAM, declared	Opens a record stream that has been previously declared with the DECLARE_STREAM statement.
START_STREAM, undeclared	Specifies and opens a record stream.
START_TRANSACTION	Starts a transaction.
STORE	Inserts a record into an existing relation.
STORE with segmented strings	Inserts a segment into a segmented string field.
TRANSACTION_HANDLE	Identifies a transaction. If you do not supply a handle name explicitly, uses the default transaction handle.

6.1 BASED ON Clause

The BASED ON clause lets you extract from the database the data type and size of a field and then use it to declare host language types. The type is defined as TYPE in Pascal and typedef in C. When you preprocess your program, the RDML preprocessor assigns the data type and size attributes associated with the field to the type you declare using the BASED ON clause.

See Section 6.6 for information on declaring host language variables.

Format

Arguments

db-handle
Database handle. A host language variable used to refer to a specific database you have invoked. For more information, see Section 6.4.

relation-name
The name of a relation in a database.

field-name
The name of a field in a relation.

Usage Notes

- Do not use the BASED ON clause to declare host language variables; instead, use the DECLARE_VARIABLE clause, which is described Section 6.6.

- If a relation name exists in more than one database being accessed by your program, you must specify the database handle to allow RDML to determine which relation you mean.

■ In RDML/C, when the field in the relation is of the TEXT, DATE, SIGNED QUADWORD, or SEGMENTED STRING data type, the BASED ON clause generates a C data type of pointer to char (char *). This allows you to return pointers to strings as shown in Example 1.

Examples

Example 1

The following programs demonstrate the use of the BASED ON clause to declare function variables. The programs use the BASED ON clause to declare the function types *job_title_type* and *job_code_type*. The programs pass the value of the JOB_CODE field to the JOB_NAME function. This function determines the job title associated with the job code and passes the job title back to the calling program. Note that in the C program, a host language variable, *temp_job_name*, is required so that space is allocated to receive the results of the strcpy function and the function can return the job title to the calling program. In Pascal, you assign a value to the function name to return the job title to the calling program.

C Program

```
#include <stdio.h>
DATABASE PERS = FILENAME "PERSONNEL";

typedef  BASED ON JOBS.JOB_CODE job_code_type;
typedef  BASED ON JOBS.JOB_TITLE job_title_type;
DECLARE_VARIABLE temp_job_name SAME AS JOBS.JOB_TITLE;

job_title_type job_name(job)
job_code_type job;
{ /* begin  function */
    READY PERS;
    START_TRANSACTION READ_ONLY;

    FOR FIRST 1 J IN JOBS
    WITH J.JOB_CODE = job
        strcpy (temp_job_name, J.JOB_TITLE);
    END_FOR;

    COMMIT;
    FINISH;
    return temp_job_name;
} /* end of function */

main ()
{
printf ("%s\n",job_name("APGM"));
}
```

BASED ON Clause

Pascal Program

```
program based_on_clause (INPUT,OUTPUT);
DATABASE PERS = FILENAME 'PERSONNEL';

type
    job_code_type  = BASED ON JOBS.JOB_CODE;
    job_title_type = BASED ON JOBS.JOB_TITLE;

function job_name (job : JOB_CODE_TYPE ) : JOB_TITLE_TYPE;

begin {* function *}
    READY PERS;
    START_TRANSACTION READ_ONLY;

    FOR FIRST 1 J IN JOBS
    WITH J.JOB_CODE = job
            job_name := J.JOB_TITLE;
    END_FOR;

    COMMIT;
    FINISH;
end; {* function *}

begin {* main *}
writeln (job_name ('APGM'));
end.
```

6.2 COMMIT Statement

The COMMIT statement ends a transaction and makes permanent any changes to the database that you made during that transaction.

Format
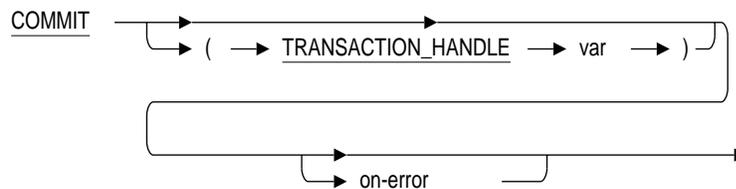
```
COMMIT ──┬──────────────────────────────────────────────┬──
         └──► ( ──► TRANSACTION_HANDLE ──► var ──► ) ──┘

         ┌──────────────────────────────────────────────┐
         └──────────────┬─────────────┬─────────────────►
                        └──► on-error ─┘
```

Arguments

TRANSACTION_HANDLE var
A TRANSACTION_HANDLE keyword followed by a host language variable. A transaction handle identifies a transaction. If you do not supply a transaction handle explicitly, RDML uses the default transaction handle.

If an application explicitly calls the DECdtm services to start a distributed transaction, it must also explicitly call DECdtm services to complete the transaction, rather than using the COMMIT statement.

See the *VAX Rdb/VMS Guide to Distributed Transactions* for more information on using the DECdtm service calls in distributed transactions.

on-error
The ON ERROR clause. Specifies a host language statement or an RDML statement or both to be performed if an error occurs during the COMMIT operation. See Section 6.17 for details.

COMMIT Statement

Usage Notes

- Digital recommends that you preprocess your program with the /NODEFAULT_TRANSACTIONS qualifier. When you use the /NODEFAULT_TRANSACTIONS qualifier, you reduce the overhead associated with the work that RDML must do to check the state of the database (for example, if the program has attached to the database, if a transaction has started, or if a transaction has ended). When you use the /NODEFAULT_TRANSACTIONS qualifier, you must explicitly start and commit or roll back your transaction or you will receive an error when you preprocess your program.

- By default, the COMMIT statement affects all readied databases (whether implicitly readied by a reference to the database or explicitly readied with the READY statement).

- The COMMIT statement writes to the database all changes to data made with the ERASE, MODIFY, and STORE statements during the transaction.

- If you start a transaction without specifying a transaction handle, you use the default transaction handle (see Section 6.27 for more information on transaction handles). There is one default transaction handle.

- By default, when the RDML preprocessor encounters a statement without a transaction handle, it uses the default transaction handle. However, Digital recommends that you preprocess your program with the /NODEFAULT_TRANSACTIONS qualifier.

- If you start a transaction and specify a transaction handle, you must use that transaction handle to commit that transaction. If the COMMIT statement succeeds, it automatically initializes the transaction handle to zero.

- The COMMIT statement also:
  - Flushes all modified buffers
  - Closes open streams created by FOR and START_STREAM statements
  - Releases all locks if you are using Rdb/VMS
  - Reduces the lock level if you are using the CONSISTENCY option of the START_TRANSACTION statement in the Rdb/ELN environment

- Because the COMMIT statement ends a stream, do not explicitly end a stream (using the END_STREAM statement) after a COMMIT statement has been executed, or Rdb will return an error.

However, your source program can place a declared END_STREAM statement after a COMMIT statement, as long as it is executed before the COMMIT statement at run time.

- Your program cannot continue in a FOR loop after a COMMIT statement.

Examples

Example 1

The following programs demonstrate the use of the COMMIT statement to make permanent changes to a field value in a database. The programs:

- Use a record selection expression to find an employee in the EMPLOYEES relation with the ID number 00193

- Use a MODIFY statement to change the field value of LAST_NAME for this employee

Although this change is written to the database at the time of the MODIFY statement, the change is not permanent until the programs issue a COMMIT statement. After the programs issue the COMMIT statement, the old value for LAST_NAME is not available.

The C program uses the function pad_string to append trailing blanks to the last name. This ensures that the last name matches the length defined for the field. For more information and the source code for pad_string, see Appendix B.

C Program

```
#include <stdio.h>
DATABASE PERS = FILENAME "PERSONNEL";

extern void pad_string();

main()
{
READY PERS;
START_TRANSACTION READ_WRITE;

FOR E IN EMPLOYEES WITH E.EMPLOYEE_ID = '00193'
   MODIFY E USING
      pad_string ("Smith-Fields", E.LAST_NAME, sizeof(E.LAST_NAME));
   END_MODIFY;
END_FOR;
```

COMMIT Statement

```
COMMIT;
FINISH;
}
```

Pascal Program

```
program commit_changes (input,output);
DATABASE PERS = FILENAME 'PERSONNEL';

begin
READY PERS;
START_TRANSACTION READ_WRITE;

FOR E IN EMPLOYEES WITH E.EMPLOYEE_ID = '00193'
   MODIFY E USING
       E.LAST_NAME := 'Smith-Fields';
   END_MODIFY;
END_FOR;

COMMIT;
FINISH;
end.
```

6.3 DATABASE Statement

The DATABASE statement names the database to be accessed in a program or program module and specifies to RDML which database to use and where to declare variables. RDML does not generate code to attach to the database when it encounters the DATABASE statement. The READY statement causes an attach to the database.

The only required parameter for the DATABASE statement is the database name. The name must be the file name that represents the database file or a logical name that translates to a file name.

You can also specify the following:

- A database handle

  A database handle is a name that you can associate with a database so that your program can refer to more than one database in a module.

- The scope of the database handle

  A database handle can be either local to the module that declared it, global to all the modules that refer to the same database, or external to the module that refers to the same database.

- Different sources of the database definition for compilation and execution

  This option allows you to compile the program using one database definition and run the program using another. You must use at least the COMPILETIME option with a file specification or logical name or a VAX CDD/Plus path name. If you also use the RUNTIME option, you can use either a file specification or a host language variable. The host language variable must be initialized to contain a file specification or a logical name that translates to a file specification before any operations can be performed against the database.

- The database key (dbkey) scope

  This option allows you to specify through a COMMIT statement or a FINISH statement whether the scope of each record's database key (dbkey) is valid. See the explanations of the DBKEY SCOPE IS COMMIT and DBKEY SCOPE IS FINISH options in the Arguments section for details.

DATABASE Statement

- The request handle scope

  This option allows you to specify the scope of system or user request handles. See the explanations of the REQUEST_HANDLE SCOPE IS DEFAULT and REQUEST_HANDLE SCOPE IS FINISH options in the Arguments section for details.

Format

INVOKE DATABASE

→ db-handle-options

COMPILETIME

PATHNAME → path-name
FILENAME → file-spec

RUNTIME FILENAME

file-spec
host-variable

DBKEY SCOPE IS

COMMIT
FINISH

REQUEST_HANDLE SCOPE IS

DEFAULT
FINISH

db-handle-options =

db-handle → =
handle-scope
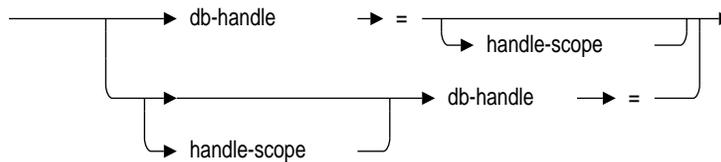
handle-scope → db-handle → =

handle-scope =

If you choose to use a bracket, you must enclose the handle scope in both the right-hand and left-hand brackets.

Arguments

db-handle-options
Database handle options. Allows you to specify the scope and name of a database handle.

db-handle
Database handle. A host language variable used to refer to a specific database you have invoked. For more information, see Section 6.4.

handle-scope
Specifies the scope of the database handle.

LOCAL
GLOBAL
EXTERNAL

- LOCAL specifies that the database will be accessed only from the current module, and that its database handle will be declared local to the current module.

- GLOBAL (the default) specifies that the database will be accessed from several modules, including the current module, and that the database handle will be declared in this module as globally visible.

- EXTERNAL specifies that the database will be accessed from several modules, including the current module, and that the database handle will be declared in this module as external.

DATABASE Statement

Note that GLOBAL and EXTERNAL are equivalent when you use the
/LINKAGE=PROGRAM_SECTIONS qualifier (the default). When you use
the /LINKAGE=GLOBAL_SYMBOLS qualifier, there must be one (and only
one) module where a given database handle is declared GLOBAL; all other
modules that access the database by means of that database handle must
declare it as EXTERNAL.

COMPILETIME (FILENAME or PATHNAME)
The source of the database definitions when the program is compiled. For
Rdb/VMS this can be either a CDD/Plus path name or a file specification. For
Rdb/ELN this should be a file specification; Rdb/ELN does not support the
data dictionary. If you specify only the compile-time identifier and omit the
run-time identifier, Rdb uses the compile-time identifier for both preprocessing
and running the program.

RUNTIME FILENAME
The source of the database definitions when the program is run. This can be
either a file specification or a host language variable. If you do not specify this
parameter, Rdb uses the compile-time identifier for both preprocessing and
running the program.

path-name
A full or relative CDD/Plus path name, enclosed in quotation marks, specifying
the source of the database definition. Use single quotation marks (' ') when
the host language is Pascal. Use double quotation marks (" ") when the host
language is C. Use only with Rdb/VMS; Rdb/ELN does not have access to the
data dictionary.

file-spec
File specification. A full or partial file specification, or logical name enclosed in
quotation marks, specifying the source of the database. Use single quotation
marks (' ') when the host language is Pascal. Use double quotation marks
(" ") when the host language is C.

host-variable
A valid host language variable that equates to a database file specification.
This variable must be declared before the DATABASE statement is issued.

DBKEY SCOPE IS COMMIT (default)
Controls when the dbkey of an erased record can be reused by Rdb. When the
DBKEY SCOPE is COMMIT, Rdb can reuse a dbkey (to store another record)
when the user who erased the original record commits his or her transaction.

DBKEY SCOPE IS FINISH

Controls when the dbkey of an erased record can be reused by Rdb. When the DBKEY SCOPE is FINISH, Rdb cannot reuse the dbkey (to store another record) until the user who erased the original record detaches from the database (by issuing a FINISH statement).

REQUEST_HANDLE SCOPE IS DEFAULT (default)

The REQUEST_HANDLE SCOPE clause is used by RDBPRE and RDML preprocessors. When a FINISH statement is issued, any request handles that were used in queries against that database during that attach become invalid. If you wish to reuse any of those request handles in a subsequent database attach, you must first initialize them.

With the REQUEST_HANDLE SCOPE IS DEFAULT option, RDML automatically initializes any request handles it generates that are in the same compilation unit as the FINISH statement. RDML does not reinitialize any user-specified request handles nor does it reinitialize any request handles that are outside of the compilation unit where the request is initiated. With this option, the value of the request handle is not set to zero after the RDML FINISH statement executes.

The REQUEST_HANDLE SCOPE IS FINISH option causes all request handles to be set to zero automatically when a FINISH statement is issued. Using this option means that you have less need to use explicit request handles.

The default option is DEFAULT.

REQUEST_HANDLE SCOPE IS FINISH

When the REQUEST_HANDLE SCOPE is FINISH, the value of the request handle is set to zero after the RDML FINISH statement executes.

The SQL FINISH statement initializes all request handles in all compilation units in a program. The RDBPRE and RDML preprocessors allow programs to define and manipulate request handles. If you do not want your request handles to be reinitialized, then you must use RDML or RDBPRE (not SQL) to do the attach, and you must use REQUEST_HANDLE SCOPE IS DEFAULT.

For more information on request handles, see the *VAX Rdb/VMS Guide to Using RDO, RDBPRE, and RDML.*

DATABASE Statement

Usage Notes

- The common data dictionary is not supported on VAXELN. Therefore, you cannot specify a path name in the DATABASE statement in the Rdb/ELN environment. Specify a file name instead.

- You must issue a DATABASE statement before you access a database and the DATABASE statement must appear before any other RDML statement in your program.

- The compile-time database must exist at preprocess time. Otherwise, the RDML preprocessor returns an error.

- The run-time database you declare must exist when you run your program. Otherwise, Rdb returns an error.

- The DATABASE statement declares a database to the program.

- In VAXELN Pascal programs, place the DATABASE statement after the MODULE statement and before the PROGRAM statement.

- In VAX Pascal programs, place the DATABASE statement after the MODULE or PROGRAM statement, and after the declaration of the host language variable that equates to a database file specification (if such a variable is used) and before any procedure or function declarations.

- In C programs, place the DATABASE statement before any function declarations; for example, before the "main" function and after the declaration of the host language variable that equates to a database file specification (if such a variable is used) and before any procedure or function declarations.

- You must declare each database that you plan to access in a module (compilation unit).

- The DATABASE statement adds a number of declarations to your program. The declarations, including variable and request definitions, are automatically included in the output file produced by the RDML preprocessor.

- The DBKEY SCOPE clause controls when the dbkey of an erased record can be reused by Rdb. When the DBKEY SCOPE is COMMIT Rdb will not reuse the dbkey of an erased record (to store another record) until the transaction that erased the original record completes when the user enters a COMMIT statement. If the user who erased the original record enters a

ROLLBACK statement, then the dbkey for that record cannot be reused by Rdb.

The DBKEY SCOPE IS FINISH clause specifies that the dbkey of each record used is guaranteed not to change until this user detaches from the database (usually, by issuing a FINISH statement). With the DBKEY SCOPE IS FINISH clause, an RDML program can complete one or several transactions and, while still attached to the database, use the dbkey obtained during a STORE operation to directly access those records.

Note that if you specify DBKEY SCOPE is FINISH and a record you accessed earlier is erased by another user, you will receive a message to indicate that that record is no longer available if you attempt to retrieve that record with the dbkey.

Also, if you specify DBKEY SCOPE IS COMMIT, and you are accessing records by means of dbkeys that you have stored in a host language variable, it is possible for you to retrieve a different (new) record than the record for which you originally saved the dbkey. This occurs when the original record is erased by another user, you commit the transaction in which you retrieved the dbkey, start another transaction, and then attempt to access records with the dbkeys you have stored in host language variables.

Examples

Example 1

The following programs demonstrate how to specify a compile-time database and a run-time database as the same database.

C Program

```
#include <stdio.h>
DATABASE PERS = FILENAME "PERSONNEL";

main ()
{
READY PERS;
START_TRANSACTION READ_ONLY;

/* perform some action on the database */

COMMIT;
FINISH;
}
```

DATABASE Statement

Pascal Program

```
program db (input,output);
DATABASE PERS = FILENAME 'PERSONNEL';

begin
READY PERS;
START_TRANSACTION READ_ONLY;

(* perform some actions on the database *)

COMMIT;
FINISH;
end.
```

Example 2

The following programs demonstrate how to specify a database handle along with naming the DECnet node, device name, directory, and file name for the compile-time database, and using a host language variable for the run-time database.

C Program

```
#include <stdio.h>

char *PRODUCTION_DATABASE;

DATABASE PERS = COMPILETIME FILENAME "DECVAX::DUA1:[DATABASE]PERSONNEL"
                RUNTIME FILENAME PRODUCTION_DATABASE;

main ()
{

PRODUCTION_DATABASE = "PERSONNEL";

READY PERS;
START_TRANSACTION READ_ONLY;

/* perform some database actions */

COMMIT;
FINISH;
}
```

Pascal Program

```
program db (input,output);

VAR PRODUCTION_DATABASE : VARYING [20] OF CHAR;

DATABASE PERS = COMPILETIME FILENAME 'DECVAX::DUA1:[DATABASE]PERSONNEL'
                RUNTIME FILENAME PRODUCTION_DATABASE;


begin

PRODUCTION_DATABASE := 'PERSONNEL';
```

```
READY PERS;
START_TRANSACTION READ_ONLY;

{* perform some actions on the database *}

COMMIT;
FINISH;
end.
```

Example 3

The following program fragments demonstrate how to specify a compile-time
database that is global to all modules. Both programs, one using the GLOBAL
database scope and the other using the EXTERNAL database scope, can access
a database.

C Program

```
/* global declarations file */

DATABASE PERS = [GLOBAL] FILENAME "PERSONNEL";

main()
{
READY PERS;
START_TRANSACTION READ_ONLY;

/* perform some actions on the database */

COMMIT;
FINISH;
}
```

Pascal Program

```
program db (input,output);
DATABASE PERS = [EXTERNAL] FILENAME 'PERSONNEL';

begin
READY PERS;
START_TRANSACTION READ_ONLY;

(* perform some actions on the database *)

COMMIT;
FINISH;
end.
```

6.4 Database Handle Clause

Rdb uses the database handle to identify the particular database that is referred to by a database request. The database handle provides context to any statement that uses it. When your program accesses a single database you do not have to include database handles or scopes in the DATABASE statement. Unlike transaction handles and request handles, database handles do not have to be declared in your programs. The RDML preprocessor automatically generates the data declaration for the database handle.

Format

db-handle =

⟶ host-variable ⟶

Argument

host-variable
A valid host language variable name.

Usage Notes

- You can use a database handle with the following RDML statements and clauses to identify a database:
  - DATABASE
  - FINISH
  - READY
  - Relation clause of the record selection expression
  - DECLARE_VARIABLE
  - DEFINE_TYPE
  - BASED ON clause
  - START_TRANSACTION statement

Database Handle Clause

- Rdb lets you attach to more than one database at a given time. You use the database handle to distinguish among the different databases in RDML statements.

- Do not change the value of a database handle after you have declared it in the database statement; RDML will maintain the handle's value for you.

- By default, the scope of a database handle is GLOBAL.

- Rdb/ELN lets separately compiled modules participate in a single transaction if the scope of a database handle has been declared as GLOBAL or EXTERNAL and the modules run synchronously. This means programmers can write code in functional modules without segregating database access or adding the overhead of multiple attaches to a database.

  Rdb/ELN processes that run asynchronously must maintain separate database handles and attach to the database separately. Rdb/ELN maintains state information about each process accessing the database. Two asynchronous processes that share a database handle will overwrite each other's state and cause errors.

- If you use GLOBAL and EXTERNAL database handles, Digital recommends that you do not place the two types of database handles in the same module. Placing the two types in a single module will not allow your applications to share a single message vector and default transaction handle, and may return ambiguous results or errors at link time. Place all GLOBAL database handles in one module to avoid any ambiguity.

  Table 6–2 summarizes how to declare database handles in a precompiled program.

Database Handle Clause

Table 6–2   Summary of Database Handle Usage in Preprocessed Programs

Number of Databases	Number of Modules	Handle Scope in Main Module	Handle Scope in Second Module	Handle Scope in Additional Modules
One	One	Not required	Not applicable	Not applicable
One	Multiple	GLOBAL	EXTERNAL	EXTERNAL
One	Multiple	EXTERNAL	GLOBAL	EXTERNAL
Multiple	One	LOCAL	Not applicable	Not applicable
Multiple	Multiple	GLOBAL	EXTERNAL	EXTERNAL
Multiple	Multiple	EXTERNAL	GLOBAL	EXTERNAL

Examples

Example 1

The following programs demonstrate how to declare a database handle, PERS, for the PERSONNEL database.

C Program

```
#include <stdio.h>
DATABASE PERS = FILENAME 'PERSONNEL';

main ()
{
READY PERS;
START_TRANSACTION READ_ONLY;

/* perform some actions on the database */

COMMIT;
FINISH PERS;
}
```

Pascal Program

```
program dbhandle (input,output);
DATABASE PERS = FILENAME 'PERSONNEL';

begin
READY PERS;
START_TRANSACTION READ_ONLY;

{* perform some actions on the database *}

COMMIT;
FINISH PERS;
end.
```

Example 2

The following program segments show how to use a database handle in a READY statement. The segments invoke a database and ready it.

C Program

```
#include <stdio.h>
DATABASE PERS = COMPILETIME FILENAME "PERSONNEL"
                RUNTIME "WORK::DUA1:[RDB.DEMO]PERSONNEL";

main()
{
READY PERS;
   .
   .
   .
}
```

Pascal Program

```
program demoprog (input,output);
DATABASE PERS = COMPILETIME FILENAME 'PERSONNEL'
                RUNTIME 'WORK::DUA1:[RDB.DEMO]PERSONNEL';

begin
READY PERS;
   .
   .
   .
end.
```

Database Handle Clause

Example 3

The following programs demonstrate the use of the database handle to resolve possible ambiguities when you invoke more than one database. The programs:

- Declare two host language variables, DB1 and DB2, as database handles for the PERSONNEL and PAYROLL databases respectively

- Use DB1 to qualify the outer FOR statement and DB2 to qualify the inner FOR statement

By matching the employee IDs from the CURRENT_INFO view in each database, the programs can print salaries stored in the PAYROLL database for the EMPLOYEES record in the PERSONNEL database.

Because no sample database named PAYROLL is provided with the software, you cannot run these programs. However, by replacing PAYROLL with PERSONNEL, you can run the programs to demonstrate the results of using two database handles.

C Program

```
#include <stdio.h>
DATABASE DB1 = FILENAME "PERSONNEL";
DATABASE DB2 = FILENAME "WORK$DISK:PAYROLL";

main ()
{
READY DB1, DB2;

START_TRANSACTION READ_ONLY;

FOR CI IN DB1.CURRENT_INFO
   printf  ("%s  %s\n", CI.ID, CI.LAST_NAME);
   FOR CI2 IN DB2.CURRENT_INFO WITH CI2.ID = CI.ID
      printf ("Actual Year-to-Date Salary = %f\n",
              CI2.SALARY);
   END_FOR; /* CI2 IN DB2.CURRENT_INFO */
END_FOR; /* CI IN DB1.CURRENT_INFO */

COMMIT;
FINISH;
}
```

Database Handle Clause

Pascal Program

```
program  ytd_salary_report (output);
DATABASE DB1 = FILENAME 'PERSONNEL';
DATABASE DB2 = FILENAME 'WORK$DISK:PAYROLL';

begin
READY DB1, DB2;

START_TRANSACTION READ_ONLY;

FOR CI IN DB1.CURRENT_INFO
   writeln (CI.ID,' ', CI.LAST_NAME);
   FOR CI2 IN DB2.CURRENT_INFO WITH CI2.ID = CI.ID
      writeln ('Actual Year-to-Date Salary = ',
               CI2.SALARY);
   END_FOR; (* CI2 IN DB2.CURRENT_INFO *)
END_FOR; (* CI IN DB1.CURRENT_INFO *)

COMMIT;
FINISH;
end.
```

6.5 DECLARE_STREAM Statement

The DECLARE_STREAM statement declares a stream name and associates that name with a record selection expression. This statement allows you to place the START_STREAM, FETCH, and END_STREAM statements in any order within your module, and within separate procedures in the same module. A stream is limited to a single module.

Format
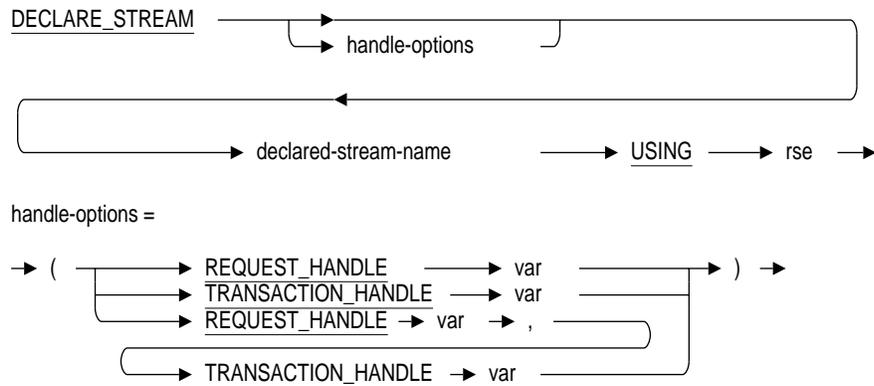
Arguments

handle-options
A request handle, a transaction handle, or both.

REQUEST_HANDLE var
A REQUEST_HANDLE keyword followed by a host language variable. A request handle identifies a compiled Rdb request. If you do not supply a request handle explicitly, RDML associates a unique request handle for the compiled request. See Section 6.20 for more information on request handles.

TRANSACTION_HANDLE var
A TRANSACTION_HANDLE keyword followed by a host language variable. A transaction handle identifies a transaction. If you do not supply a transaction handle explicitly, RDML uses the default transaction handle.

declared-stream-name
A name you give to the stream you declare. The declared-stream-name must be a valid host language name.

rse
A record selection expression. A phrase that defines the specific conditions that individual records must meet before Rdb includes them in a record stream.

Usage Notes

- The DECLARE_STREAM statement does not require that the same number of END_STREAM statements and START_STREAM statements appear within the same procedure, as long as at execution time exactly one END_STREAM statement is executed for each START_STREAM statement. You may find this feature particularly helpful when you are using host language conditional statements.

- The DECLARE_STREAM statement must be used in conjunction with the declared START_STREAM statement. The DECLARE_STREAM statement will not work in conjunction with the undeclared START_STREAM statement, and the reverse is also true.

- The DECLARE_STREAM statement must precede any other reference to the stream that it declares.

- The stream name must not conflict with any RDML keywords. See Table 1–1 for the list of RDML keywords.

- Digital recommends that all programs use the DECLARE_STREAM statement (with the declared START_STREAM statement) in place of the undeclared START_STREAM statement. The declared START_STREAM statement provides all the functionality of the undeclared START_STREAM statement and provides more flexibility in programming than the undeclared START_STREAM statement.

- Any host language variables that appear in the record selection expression only need to be declared within the program code that contains the START_STREAM statement declared by the DECLARE_STREAM statement.

DECLARE_STREAM Statement

Examples

Example 1

The following programs demonstrate how you can place the START_STREAM, FETCH, and END_STREAM statements in any order in a module. These programs are not intended to show good programming style, but rather the flexibility that the DECLARE_STREAM statement allows in programming.

C Program

```
#include <stdio.h>
#define   TRUE 1
#define   FALSE 0

DATABASE PERS = FILENAME "PERSONNEL";

DECLARE_STREAM by_first_name USING
    E2 IN EMPLOYEES SORTED BY E2.FIRST_NAME, E2.LAST_NAME;

DECLARE_STREAM by_last_name USING
    E1 IN EMPLOYEES SORTED BY E1.LAST_NAME, E1.FIRST_NAME;

int end_of_stream = FALSE;

close_last()
{
END_STREAM by_last_name;
}

close_first()
{
END_STREAM by_first_name;
}

read_first()
{
FETCH by_first_name;
}

read_last()
{
FETCH by_last_name
AT END
  end_of_stream = TRUE;
END_FETCH;
}
```

DECLARE_STREAM Statement

```
open_first()
{
START_STREAM by_first_name;
}

open_last()
{
START_STREAM by_last_name;
}


main()
{
READY PERS;
START_TRANSACTION READ_ONLY;
open_first();
open_last();
/* The streams BY_LAST_NAME and BY_FIRST_NAME will contain the
   same number of records.  It is only necessary to test
   for AT END once. */

    end_of_stream = FALSE;

read_last();
read_first();

while (!end_of_stream)
    {
       /* Alphabetical listing by last name down left column */

       printf ("%s%s",E1.LAST_NAME,E1.FIRST_NAME);
       printf ("                    "); /* skip 20 spaces */

       /* Alphabetical listing by first name down right column */

       printf ("%s%s\n",E2.FIRST_NAME, E2.LAST_NAME);

       read_last();

       if (!end_of_stream)
           {
           read_first();
           }
}
close_last();
close_first();

COMMIT;
FINISH;
}
```

DECLARE_STREAM Statement

Pascal Program

```
[inherit ('sys$library:starlet.pen')]

program new_start (input, output);
DATABASE PERS = FILENAME 'PERSONNEL';


var
end_of_stream : BOOLEAN;

DECLARE_STREAM BY_LAST_NAME USING
    E1 IN EMPLOYEES SORTED BY E1.LAST_NAME, E1.FIRST_NAME;

DECLARE_STREAM BY_FIRST_NAME USING E2 IN EMPLOYEES SORTED BY
    E2.FIRST_NAME, E2.LAST_NAME;

procedure close_last;
begin
END_STREAM BY_LAST_NAME;
end;

procedure close_first;
begin
END_STREAM BY_FIRST_NAME;
end;

procedure read_first;
begin
FETCH BY_FIRST_NAME;
end;

procedure read_last;
begin
FETCH BY_LAST_NAME
AT END
  end_of_stream := TRUE;
END_FETCH;
end;

procedure open_first;
begin
START_STREAM by_first_name;
end;

procedure open_last;
begin
START_STREAM by_last_name;
end;
```

DECLARE_STREAM Statement

```
begin
READY PERS;
START_TRANSACTION READ_ONLY;
open_first;
open_last;
(* The streams BY_LAST_NAME and BY_FIRST_NAME will contain the
  same number of records.  It is only necessary to test
   for AT END once. *)

    end_of_stream := FALSE;

read_last;
read_first;

while not end_of_stream do
   begin
      (* Alphabetical listing by last name down left column *)

      write (E1.LAST_NAME,E1.FIRST_NAME);
      write ('                   '); (* skip 20 spaces *)

      (* Alphabetical listing by first name down right column *)

      writeln (E2.FIRST_NAME, E2.LAST_NAME);

      read_last;

      if not end_of_stream then
            read_first;
   end;

close_last;
close_first;

COMMIT;
FINISH;
end.
```

6.6 DECLARE_VARIABLE Clause

The DECLARE_VARIABLE clause lets you declare a host language variable that is compatible with a field associated with a database relation. The variable inherits the data type and size attributes associated with the field.

Note *The DECLARE_VARIABLE and DEFINE_TYPE clauses have exactly the same function. Digital renamed the clause to clarify that its function is to declare host language variables, not define host language types. Note that the DEFINE_TYPE clause can still be used; however, Digital recommends that all new applications use the DECLARE_VARIABLE clause in place of the DEFINE_TYPE clause.*

Format
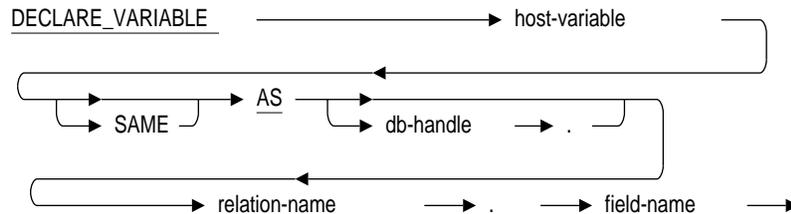
Arguments

host-variable
A valid host language variable.

db-handle
A database handle. A host language variable used to refer to a specific database your program uses. The database handle must be the same database handle specified in the DATABASE statement.

relation-name
The name of a relation in a database.

field-name
The name of a field in a relation.

Usage Notes

- You should not use the DECLARE_VARIABLE clause to declare program functions TYPE (in Pascal) or typedef (in C); use the BASED ON clause instead.

Examples

Example 1

The following example demonstrates the use of the DECLARE_VARIABLE clause to declare a host language variable that is intended to hold database values. The programs:

- Declare the variable, *badge*, to have the same data type and size attributes as the EMPLOYEE_ID field in the EMPLOYEES relation.

- Use this variable for interactive processing. Note that the interactive portion of the programs appears before the READY statement. This keeps locks on the database to a minimum.

- Select the record from the EMPLOYEES relation that has the same value for EMPLOYEE_ID as is stored in *badge*.

- Modify the status code of this employee.

Note that the C program uses the function read_string to prompt for and receive a value for *badge*. For more information and the source code for read_string, see Appendix B.

C Program

```
#include <stdio.h>
DATABASE PERS = FILENAME "PERSONNEL";

extern void read_string();
static DECLARE_VARIABLE badge SAME AS EMPLOYEES.EMPLOYEE_ID;

main()
{
read_string ("Employee ID: ", badge, sizeof(badge));

READY PERS;
START_TRANSACTION READ_WRITE;
```

DECLARE_VARIABLE Clause

```
FOR E IN EMPLOYEES WITH E.EMPLOYEE_ID = badge
   MODIFY E USING
      strcpy(E.STATUS_CODE,"1");
   END_MODIFY;
END_FOR;

ROLLBACK;
FINISH;
}
```

Pascal Program

```
program modify_with_host (input,output);
DATABASE PERS = FILENAME 'PERSONNEL';

var
   DECLARE_VARIABLE badge SAME AS EMPLOYEES.EMPLOYEE_ID;

begin
write  ('Employee ID: ');
readln (badge);

READY PERS;
START_TRANSACTION READ_WRITE;

FOR E IN EMPLOYEES WITH E.EMPLOYEE_ID = badge
   MODIFY E USING
      E.STATUS_CODE := '1';
   END_MODIFY;
END_FOR;

ROLLBACK;
FINISH;
end.
```

6.7 DEFINE_TYPE Clause

The DECLARE_VARIABLE and DEFINE_TYPE clauses have exactly the same function. Digital renamed the clause to clarify that its function is to declare host language variables, not to define host language types. Note that the DEFINE_TYPE clause can still be used; however, Digital recommends that all new applications use the DECLARE_VARIABLE clause in place of the DEFINE_TYPE clause. Refer to Section 6.6 for more information.

6.8 END_STREAM Statement, Declared

The declared END_STREAM statement ends a declared stream.

Note *Digital recommends that all programs use the declared START_STREAM statement (with the DECLARE_STREAM statement) in place of the un-declared START_STREAM statement. The declared START_STREAM statement provides all the functionality of the undeclared START_STREAM statement and provides more flexibility in programming than the undeclared START_STREAM statement.*

Format

Arguments

stream-name
A valid host language variable. This name must be the same name used in the associated DECLARE_STREAM statement.

on-error
The ON ERROR clause. Specifies host language statements or RDML statement or both to be performed if an error occurs during the END_STREAM operation. See Section 6.17 for details.

Usage Notes

- You can have more or fewer declared END_STREAM statements than declared START_STREAM statements in your program, as long as the structure of the program ensures that exactly one END_STREAM statement is executed for each START_STREAM statement that is executed.

END_STREAM Statement, Declared

■ You can issue several declared END_STREAM statements in a module, and as long as you use the same declared stream name in each declared END_STREAM statement, they will all refer to the same stream.

Examples

Example 1

The following examples demonstrate the use of the declared END_STREAM clause. The programs:

■ Declare a stream *sal* with the DECLARE_STREAM statement that limits the stream to those records with a value less than $10,000.00 in the SALARY_AMOUNT field

■ Start a read/write transaction

■ Fetch the first record in the stream

■ Modify that record so that the value in the SALARY_AMOUNT field is increased by 50 percent

■ Fetch and modify records in the stream until all the records have been modified

■ End the stream with the declared END_STREAM statement

C Program

```
#include <stdio.h>
#define TRUE 1
#define FALSE 0

DATABASE PERS = FILENAME "PERSONNEL";

DECLARE_STREAM sal USING SH IN SALARY_HISTORY
     WITH SH.SALARY_AMOUNT LT 10000;

int end_of_stream;

main()
{
READY PERS;
START_TRANSACTION READ_WRITE;

    START_STREAM sal;

    FETCH sal
     AT END
        end_of_stream = TRUE;
    END_FETCH;
```

END_STREAM Statement, Declared

```
        while (! end_of_stream)
            {
            MODIFY SH USING
              SH.SALARY_AMOUNT = SH.SALARY_AMOUNT * (1.5);
            END_MODIFY;

            FETCH sal
              AT END
                end_of_stream = TRUE;
            END_FETCH;
            }

        END_STREAM sal;

        COMMIT;
        FINISH;
}
```

Pascal Program

```
program anycond (input,output);
DATABASE PERS = FILENAME 'PERSONNEL';

var
end_of_stream : boolean;

DECLARE_STREAM sal USING SH IN SALARY_HISTORY
     WITH SH.SALARY_AMOUNT LT 10000;

begin
    READY PERS;
    START_TRANSACTION READ_WRITE;

    START_STREAM sal;

    FETCH sal
     AT END
       end_of_stream := TRUE;
    END_FETCH;

    while not end_of_stream do
     begin
       MODIFY SH USING
         SH.SALARY_AMOUNT := SH.SALARY_AMOUNT * (1.5);
       END_MODIFY;

       FETCH sal
         AT END
           end_of_stream := TRUE;
       END_FETCH;

     end;
```

END_STREAM Statement, Declared

```
      END_STREAM sal;
      COMMIT;
      FINISH;
end.
```

6.9 END_STREAM Statement, Undeclared

The undeclared END_STREAM statement ends an undeclared stream.

Format

END_STREAM ⟶ stream-name

on-error

Arguments

stream-name
A valid host language variable. This name must be the same name used in the associated START_STREAM statement.

on-error
The ON ERROR clause. Specifies host language statements or RDML statements or both to be performed if an error occurs during the END_STREAM operation. See Section 6.17 for details.

Usage Notes

- The END_STREAM statement for an undeclared stream must follow the corresponding START_STREAM statement in the source program.

- There must be one and only one END_STREAM statement for every undeclared START_STREAM statement. If you have fewer END_STREAM statements than undeclared START_STREAM statements, you will receive the error message: "%RDML-W-UNBALSTRM, Undeclared stream "stream name" has no END_STREAM statement".

Examples

Example 1

C Program

```
#include <stdio.h>
DATABASE PERS = FILENAME "PERSONNEL";

main()
{
READY PERS;
START_TRANSACTION READ_ONLY;

START_STREAM CURRENT_INF_STREAM USING
   CI IN CURRENT_INFO SORTED BY DESC CI.SALARY;
   FETCH CURRENT_INF_STREAM;
      printf ("%s makes the largest salary!\n", CI.LAST_NAME);
END_STREAM CURRENT_INF_STREAM;

COMMIT;
FINISH;
}
```

Pascal Program

```
program record_stream (input,output);
DATABASE PERS = FILENAME 'PERSONNEL';

begin
READY PERS;
START_TRANSACTION READ_ONLY;

START_STREAM CURRENT_INF_STREAM USING
   CI IN CURRENT_INFO SORTED BY DESC CI.SALARY;
   FETCH CURRENT_INF_STREAM;
      writeln (CI.LAST_NAME, ' makes the largest salary!');
END_STREAM CURRENT_INF_STREAM;

COMMIT;
FINISH;
end.
```

6.10 ERASE Statement

The ERASE statement deletes a record from a relation or an open stream.

Format
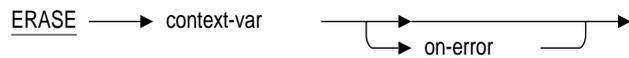
ERASE ⟶ context-var ⟶ on-error ⟶

Arguments

context-var
A context variable. A temporary name that you associate with a relation.
You define a context variable in a relation clause. See Section 4.1 for more
information on context variables.

on-error
The ON ERROR clause. Specifies host language statements or RDML
statements or both to be performed if an error occurs during the ERASE
operation. See Section 6.17 for details.

Usage Notes

- Before using the ERASE statement, you must start a read/write
  transaction and establish a record stream using a context variable
  with a FOR statement or a START_STREAM statement.

- Because the effects of erasing some records in one relation and others in a
  second can be unpredictable, you should not erase records from views that
  refer to more than one relation.

Examples

Example 1

The following programs demonstrate the use of the ERASE statement to delete records from a relation. The programs:

- Start a read/write transaction

- Find the records in the COLLEGES relation with the college code "PURD"

- Delete those records from the COLLEGES relation

- Roll back the transaction

C Program

```
#include <stdio.h>
DATABASE PERS = FILENAME "PERSONNEL";

main()
{
READY PERS;
START_TRANSACTION READ_WRITE;

FOR C IN COLLEGES WITH C.COLLEGE_CODE = "PURD"
    ERASE C;
END_FOR;

ROLLBACK;
FINISH;
}
```

Pascal Program

```
program erase_record (input,output);
DATABASE PERS = FILENAME 'PERSONNEL';

begin
READY PERS;
START_TRANSACTION READ_WRITE;

FOR C IN COLLEGES WITH C.COLLEGE_CODE = 'PURD'
    ERASE C;
END_FOR;

ROLLBACK;
FINISH;
end.
```

ERASE Statement

Example 2

The following programs demonstrate the use of the ERASE statement to delete all records with a particular field value from a relation. The programs delete all the employee records from the JOB_HISTORY relation that have a department code of "ELMC." The programs use the ANY statement to find any records in the JOB_HISTORY relation that have the value "ELMC" in the DEPARTMENT_CODE field. If there is no record with this value, the programs print the message "There are no employees in department ELMC." If at least one record has this value then the programs:

- Use the COUNT function to compute the number of records with this value

- Print this computed value

- Use the FIRST statement to find the first record in the DEPARTMENTS relation with the value "ELMC" to determine the department name associated with this department code

- Print this department name

- Use a FOR statement to find all the records in the JOB_HISTORY relation with the job code "ELMC"

- Print a message noting the employee ID of the employee about to be deleted from the relation

- Use the ERASE statement to delete the employees from the relation

C Program

```
#include <stdio.h>
DATABASE PERS = FILENAME "PERSONNEL";

int who;
int num;

main()

{
READY PERS;
START_TRANSACTION READ_WRITE;

GET
  who = ANY JH IN JOB_HISTORY
        WITH JH.DEPARTMENT_CODE = "ELMC";
END_GET;
```

```
if (who)
   {
   GET
      num =    COUNT OF JH IN JOB_HISTORY
               WITH JH.DEPARTMENT_CODE = "ELMC";
   END_GET;

   printf ("Deleting %d", num);
   printf (" employees in ");
   printf ("%s\n\n", FIRST D.DEPARTMENT_NAME FROM D IN DEPARTMENTS
                     WITH D.DEPARTMENT_CODE = "ELMC");

   FOR JH IN JOB_HISTORY WITH JH.DEPARTMENT_CODE = "ELMC"
      printf ( "Deleting  %s\n", JH.EMPLOYEE_ID);
      ERASE JH;
   END_FOR; /* JH IN JOB_HISTORY*/
   }
else
   printf ("There are no employees in department ELMC");

ROLLBACK;
FINISH;
}
```

Pascal Program

```
program delete_all (input,output);
DATABASE PERS = FILENAME 'PERSONNEL';

var
 who : boolean;
 num : integer;

begin
READY PERS;
START_TRANSACTION READ_WRITE;

GET
  who = ANY JH IN JOB_HISTORY
        WITH JH.DEPARTMENT_CODE = 'ELMC';
END_GET;

if (who) then
   begin
   GET
     num = COUNT OF JH IN JOB_HISTORY
           WITH JH.DEPARTMENT_CODE = 'ELMC';
   END_GET;

   write (' Deleting',num,' employees in ');
   writeln (FIRST D.DEPARTMENT_NAME FROM D IN DEPARTMENTS
           WITH D.DEPARTMENT_CODE = 'ELMC');
   writeln;
```

ERASE Statement

```
    FOR JH IN JOB_HISTORY WITH JH.DEPARTMENT_CODE = 'ELMC'
        writeln ( 'Deleting ', JH.EMPLOYEE_ID);
        ERASE JH;
    END_FOR; (* JH IN JOB_HISTORY*)
    end
else
    writeln ('There are no employees in department ELMC');

ROLLBACK;
FINISH;
end.
```

Example 3

The following programs demonstrate the use of the ERASE statement
to remove a specific employee's records from multiple relations. The
programs remove an existing employee's EMPLOYEE, JOB_HISTORY,
and SALARY_HISTORY records from the database. If the employee has any
DEGREE records, the DEGREE records are also removed. After prompting
the user for the employee's ID, the program locates the records that contain
that ID number and uses the ERASE statement to delete the records. The
FOR loop ensures that all the records with that ID in the specified relation are
deleted.

C Program

```
#include <stdio.h>
#define TRUE 1
#define FALSE 0
DATABASE PERS = FILENAME 'PERSONNEL';

int employee_found;
extern void read_string();

DECLARE_VARIABLE id SAME AS EMPLOYEES.EMPLOYEE_ID;
DECLARE_STREAM emp_stream USING E IN EMPLOYEES WITH E.EMPLOYEE_ID = id;

main()
{
employee_found = FALSE;
read_string("Enter ID of employee to be deleted from database: ",
            id, sizeof(id) );

READY PERS;
START_TRANSACTION READ_WRITE RESERVING
  EMPLOYEES       FOR SHARED WRITE,
  DEGREES         FOR SHARED WRITE,
  JOB_HISTORY     FOR SHARED WRITE,
  SALARY_HISTORY FOR SHARED WRITE;
```

```
FOR E2 IN EMPLOYEES WITH E2.EMPLOYEE_ID = id
  employee_found = TRUE;
  if (employee_found)
    {
    FOR JH IN JOB_HISTORY WITH JH.EMPLOYEE_ID = id
        printf ("\n Deleting employee's job history record(s)");
        ERASE JH;
    END_FOR;

    FOR SH IN SALARY_HISTORY WITH SH.EMPLOYEE_ID = id
        printf ("\n Deleting employee's salary history record(s)");
        ERASE SH;
    END_FOR;

    if (ANY D IN DEGREES WITH D.EMPLOYEE_ID = id)
        {
         FOR D IN DEGREES WITH D.EMPLOYEE_ID = id
             ERASE D;
             printf ("\n Deleting employee's degree record(s)");
         END_FOR
        }
    else
        {
         printf ("\n Employee with ID %s has no DEGREE record.", id);
         printf ("\n Continuing transaction.");
        }
    printf ("\n Employee %s %s deleted from database.",
                E.FIRST_NAME,E.LAST_NAME);

     ERASE E2;
     }
END_FOR;

if (! employee_found)
  {
   ROLLBACK;
   printf ("Employee not found with ID = %s", id);
  }
else
  {
  COMMIT;
  printf("Employee with ID %s deleted from database.", id);
  }

FINISH;
}
```

Pascal Program

```
program remove_emp (input,output);
DATABASE PERS = FILENAME 'PERSONNEL';

var
   DECLARE_VARIABLE id SAME AS EMPLOYEES.EMPLOYEE_ID;
   DECLARE_STREAM EMP_STREAM USING E IN EMPLOYEES WITH E.EMPLOYEE_ID = id;
   emp_found : boolean;
```

ERASE Statement

```
Begin
emp_found := FALSE;

write ('Enter ID of employee to be deleted from database: ');
readln (id);

READY PERS;
START_TRANSACTION READ_WRITE RESERVING
  EMPLOYEES       FOR SHARED WRITE,
  DEGREES         FOR SHARED WRITE,
  JOB_HISTORY     FOR SHARED WRITE,
  SALARY_HISTORY  FOR SHARED WRITE;

FOR E2 IN EMPLOYEES WITH E2.EMPLOYEE_ID = id
 emp_found := true;

 if emp_found = true
 then
   begin
   FOR JH IN JOB_HISTORY WITH JH.EMPLOYEE_ID = id
       writeln ('Deleting employee''s job history record(s)');
       ERASE JH;
   END_FOR;

   FOR SH IN SALARY_HISTORY WITH SH.EMPLOYEE_ID = id
       writeln ('Deleting employee''s salary history record(s)');
       ERASE SH;
   END_FOR;

   if (ANY D IN DEGREES WITH D.EMPLOYEE_ID = id) then
       FOR D IN DEGREES WITH D.EMPLOYEE_ID = id
           ERASE D;
           writeln ('Deleting employee''s degree record(s)');
       END_FOR
   else
       begin
           writeln ('Employee with ID ',id, ' has no DEGREE record.');
           writeln ('Continuing transaction.');
       end;

    ERASE E2;
    end;
END_FOR;

if emp_found = false
then
  begin
    writeln ('Employee not found with ID = ',id);
    ROLLBACK;
  end
else
  begin
    COMMIT;
    writeln ('Employee with ID', id, 'deleted from database.');
  end;

FINISH;
end.
```

6.11 FETCH Statement

The FETCH statement retrieves the next record from a record stream. The FETCH statement is used:

- With an undeclared START_STREAM statement

  - After the START_STREAM statement

  - Before any other RDML statements that affect the context established by the START_STREAM statement

- With a declared START_STREAM statement

  - After the DECLARE_STREAM statement

  - Either before or after the declared START_STREAM statement as long as it is executed after the declared START_STREAM statement has executed. (The FETCH statement may physically appear in the source file before or after the declared START_STREAM statement, but must be executed after the declared START_STREAM statement.)

The FETCH statement advances the pointer for a record stream to the next record of a relation. Unlike the FOR statement, which advances to the next record automatically, the FETCH statement allows you explicit control of the record stream. For instance, you might use the FETCH statement to print a report where the first six rows have five columns, and the seventh row only three. Note that the FETCH statement syntax is the same when used in either a declared or undeclared stream.

Format

```
FETCH ──→ stream-name ──────────────────────────────
                              ┌──────────────────┐
                              └─→ on-error ───────┘

     ┌────────────────────────────────────────────────────
     ├─→ AT END ────→ statement ──→ END_FETCH ──┐
     │              └──────←───────┘            │
     └─→ END_FETCH ─────────────────────────────┘
```
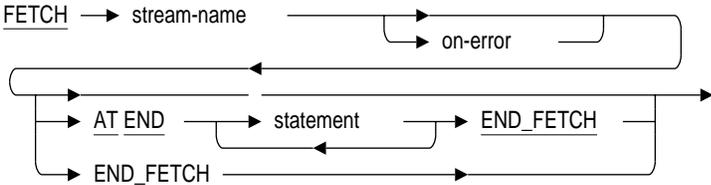
FETCH Statement

Arguments

stream-name
The stream from which you want to FETCH the next record.

on-error
The ON ERROR clause. Specifies host language statements or RDML statements or both to be performed if an error occurs during the FETCH operation. See Section 6.17 for details.

statement
Any valid RDML or host language statement to be executed when your program reaches the end of a record stream. Use a semicolon (;) at the end of each RDML, Pascal, or C statement.

Usage Notes

- Once you establish and open a stream with the START_STREAM statement, use the FETCH statement to establish the first record in the record stream as the current record. After that, each FETCH statement makes the next record in the stream the current record.

- The FETCH statement only advances the pointer in a record stream. You must use other data manipulation statements to manipulate each record in the stream. For example, you might use FETCH to advance the pointer, and the GET statement to assign values from that record to host language variables.

- Your program can use either FOR statements or START_STREAM statements to establish record streams. Furthermore, you can use both methods in one program. However, you cannot use the FETCH statement to advance the pointer in a record stream established by a FOR statement. The FOR statement advances to the next record automatically.

- You must always use a FETCH statement before a MODIFY or an ERASE statement if you want to modify or erase a record in a stream created by the START_STREAM statement. The START_STREAM statement does not retrieve the first record in a stream automatically.

- The AT END clause allows you to include statements to be executed when there are no more records in a record stream. For example, if you embed the FETCH statement in a host language loop structure, you probably want your program to stop looping when there are no more records in the

stream. You can set the conditions for terminating the loop based on a flag that is set by the AT END clause. For example, in pseudo code:

```
while flag = true
  FETCH stream_name
      AT END
          flag = false;
  END_FETCH;
end while_loop
```

Examples

Example 1

The following examples demonstrate the use of the FETCH statement. The programs:

- With the DECLARE_STREAM statement, declare a stream *sal* that limits the stream to those records with a value less than ten thousand in the SALARY_AMOUNT field

- Start a read/write transaction

- Fetch the first record in the stream

- Modify that record so that the value in the SALARY_AMOUNT field is increased by fifty percent

- Fetch and modify records in the stream until all the records have been modified

- End the stream with the declared END_STREAM statement

C Program

```
#include <stdio.h>
#define TRUE 1
#define FALSE 0

DATABASE PERS = FILENAME "PERSONNEL";

DECLARE_STREAM sal USING SH IN SALARY_HISTORY
     WITH SH.SALARY_AMOUNT LT 10000;

int end_of_stream;

main()
{
READY PERS;
START_TRANSACTION READ_WRITE;
```

FETCH Statement

```
        START_STREAM sal;

        FETCH sal
         AT END
            end_of_stream = TRUE;
        END_FETCH;

        while (! end_of_stream)
            {
            MODIFY SH USING
              SH.SALARY_AMOUNT = SH.SALARY_AMOUNT * (1.5);
            END_MODIFY;

            FETCH sal
              AT END
                 end_of_stream = TRUE;
            END_FETCH;
            }

        END_STREAM sal;

        COMMIT;
        FINISH;
    }
```

Pascal Program

```
program anycond (input,output);
DATABASE PERS = FILENAME 'PERSONNEL';

var
end_of_stream : boolean;

DECLARE_STREAM sal USING SH IN SALARY_HISTORY
      WITH SH.SALARY_AMOUNT LT 10000;

begin
    READY PERS;
    START_TRANSACTION READ_WRITE;

    START_STREAM sal;

    FETCH sal
     AT END
        end_of_stream := TRUE;
    END_FETCH;

    while not end_of_stream do
     begin
       MODIFY SH USING
         SH.SALARY_AMOUNT := SH.SALARY_AMOUNT * (1.5);
       END_MODIFY;
```

FETCH Statement

```
      FETCH sal
        AT END
           end_of_stream := TRUE;
        END_FETCH;

     end;

    END_STREAM sal;
    COMMIT;
    FINISH;
end.
```

6.12 FINISH Statement

The FINISH statement explicitly detaches from a database. By default,
FINISH, with no parameters, also commits all transactions that have not
been committed or rolled back.

Format

FINISH ──────┬──→──── db-handle ────┬──────┬──→── on-error ──┬────→
             └────────←──────────────┘   └─────────────────┘
                         ,

Arguments

db-handle
A host language variable that identifies the database to be closed. Use
the database handle you associated with the database in the DATABASE
statement.

on-error
The ON ERROR clause. Specifies host language statements or RDML
statements or both to be performed if an error occurs during the FINISH
operation. See Section 6.17 for details.

Usage Notes

- By default, an unqualified FINISH statement (no specified database
  handle) automatically closes all databases known to the module, commits
  the default transaction and re-initializes all RDML-specified handles
  (database, transaction, and request handles) to zero.

- Digital recommends that you use the /NODEFAULT_TRANSACTIONS
  qualifier when you preprocess your program. When you use the
  /NODEFAULT_TRANSACTIONS qualifier, you reduce the overhead
  associated with the work that RDML must do to check the state of the
  database (for example, if the program has attached to the database, if a
  transaction has started, or if a transaction has ended). When you use the
  /NODEFAULT_TRANSACTIONS qualifier, you must explicitly attach to
  the database with a READY statement, and explicitly start a transaction

with the START_TRANSACTION statement. The /NODEFAULT_
TRANSACTIONS qualifier will not affect the re-initialization of RDML-
specified handles.

If you specify the /NODEFAULT_TRANSACTIONS qualifier and you
use a FINISH statement without first committing or rolling back your
transaction, Rdb returns an error. If you are using Rdb/VMS, refer to
the *VAX Rdb/VMS Guide to Using RDO, RDBPRE, and RDML* for more
information on the /NODEFAULT_TRANSACTIONS qualifier. Refer to the
Rdb/ELN documentation set if you are using Rdb/ELN.

- A FINISH statement *will never* initialize user-supplied handles to zero a
  second time.

- If you do not use the /NODEFAULT_TRANSACTIONS qualifier and
  you issue a FINISH statement without specifying a database handle, it
  will cause your program to detach from all the databases invoked in the
  module.

- Once a database is opened, the program must enter a FINISH statement
  before the program ends or exits. A database is considered open if the
  program has issued a READY statement (or if you do not specify the
  /NODEFAULT_TRANSACTIONS qualifier and the program has issued a
  START_TRANSACTION statement, or the database has been referred to
  in another RDML statement). Whether you access a single database or
  multiple databases, this means you must execute a FINISH statement just
  prior to exiting your program. You can use one FINISH statement for all
  databases, or you can use a single FINISH statement for each database by
  using database handles.

- For the best performance, attach to a database once and finish it once
  within a program. Attaching to a database several times within your
  application program degrades performance.

- Close the database before you exit your program to avoid an error.

Examples

Example 1

The following programs:

- Declare a database

- Enter an RDML FOR loop, implicitly attaching to the database

- Print the last name of each employee in the EMPLOYEES relation

FINISH Statement

- Commit the transaction

- Close the database

C Program

```
#include <stdio.h>
DATABASE PERS = FILENAME "PERSONNEL";

main()
{
FOR E IN EMPLOYEES
   printf ("%s\n", E.LAST_NAME);
END_FOR;

COMMIT;
FINISH;
}
```

Pascal Program

```
program empupdate (input,output);
DATABASE PERS = FILENAME 'PERSONNEL';

begin
FOR E IN EMPLOYEES
   writeln (E.LAST_NAME);
END_FOR;

COMMIT;
FINISH;
end.
```

Example 2

The following program fragments:

- Declare two databases, CUSTORDER and PARTSBOM

- Assign database handles to each

- Open both databases with the READY statement

- Perform some action (indicated by vertical ellipsis)

- Finish both databases

Between the second READY statement and the first FINISH statement, you can access both databases at once.

FINISH Statement

C Program

```
#include <stdio.h>
DATABASE ORDER_DB =  FILENAME "WORK$DISK:CUSTORDER";
DATABASE PARTS_DB =  FILENAME "WORK$DISK:PARTSBOM";

main()
{
READY ORDER_DB;
   .
   .
   .
READY PARTS_DB;
   .
   .
   .
FINISH ORDER_DB;
   .
   .
   .
FINISH PARTS_DB;
}
```

Pascal Program

```
program declare_two_db;
DATABASE ORDER_DB =  FILENAME 'WORK$DISK:CUSTORDER';
DATABASE PARTS_DB =  FILENAME 'WORK$DISK:PARTSBOM';

begin
READY ORDER_DB;
   .
   .
   .
READY PARTS_DB;
   .
   .
   .
FINISH ORDER_DB;
   .
   .
   .
FINISH PARTS_DB;
end.
```

6.13 FOR Statement

The FOR statement executes a statement or group of statements once for each record in a record stream formed by a record selection expression. You can nest FOR statements within other FOR statements.

You can use either FOR statements or START_STREAM statements to establish record streams in your program. Furthermore, you can use both methods in one program. However, you cannot use the FETCH statement to advance the pointer in a record stream established by a FOR statement. The FOR statement automatically advances to the next record for each iteration.

Format

handle-options =

Arguments

handle-options
A request handle, a transaction handle, or both.

REQUEST_HANDLE var
A REQUEST_HANDLE keyword followed by a host language variable. A request handle identifies a compiled Rdb/VMS request. If you do not supply a request handle explicitly, RDML generates a unique request handle for the compiled request. See Section 6.20 for more information on request handles.

TRANSACTION_HANDLE var

A TRANSACTION_HANDLE keyword followed by a host language variable. A transaction handle identifies a transaction. If you do not supply a transaction handle explicitly, RDML uses the default transaction handle.

rse

A record selection expression. A phrase that defines specific conditions that individual records must meet before Rdb includes them in the record stream. See Chapter 4 for more information.

on-error

The ON ERROR clause. Specifies host language statements or RDML statements or both to be performed if an error occurs during the FOR operation. See Section 6.17 for details.

statement

Any valid RDML or host language statement to be executed within the FOR loop. Use a semicolon (;) at the end of each RDML, Pascal, or C statement.

Usage Notes

- You can use nested FOR loops to form outer joins. In a common type of join, such as an equijoin, certain values in a field from one relation are matched with those in another relation. Values that do not match are not included in the join. An outer join also establishes relationships between data items by matching fields, but it includes the unmatched values by adding them to the result of the equijoin.

  To accomplish an outer join, you must include the MISSING clause in the record selection expression so the unmatched values are added at the end of the join.

- For best results, do not use nested FOR loops unless you are referring to more than one database, or performing outer joins.

- You can use a context variable from a FOR statement again, as soon as you end the FOR loop with the END_FOR statement.

FOR Statement

Examples

Example 1

The following programs demonstrate the use of the FOR statement to create a record stream. The programs:

- Declare a host language variable *dept_code*

- Prompt for a value for *dept_code*

- Start a read-only transaction

- Create a record stream defined by a record selection expression that uses the value of *dept_code*

- Display the department name for each record in that stream

The C program uses the function read_string to prompt for and receive a value for *dept_code*. For more information and the source code for read_string, see Appendix B. The Pascal writeln and readln functions serve a similar function.

C Program

```
#include <stdio.h>
DATABASE PERS = FILENAME "PERSONNEL";

extern void read_string ();
DECLARE_VARIABLE dept_code SAME AS DEPARTMENTS.DEPARTMENT_CODE;

main ()
{
read_string ("Department Code: ",dept_code, sizeof(dept_code));

READY PERS;
START_TRANSACTION READ_ONLY;

FOR D IN DEPARTMENTS
   WITH D.DEPARTMENT_CODE = dept_code
      printf ("Department name =  %s\n ", D.DEPARTMENT_NAME);
END_FOR;

COMMIT;
FINISH;
}
```

Pascal Program

```
program for_in_rse (input,output);
DATABASE PERS = FILENAME 'PERSONNEL';

var
   DECLARE_VARIABLE dept_code SAME AS DEPARTMENTS.DEPARTMENT_CODE;

begin
write  ('Department Code: ');
readln (dept_code);

READY PERS;
START_TRANSACTION READ_ONLY;

FOR D IN DEPARTMENTS
   WITH D.DEPARTMENT_CODE = dept_code
      writeln ('Department name =  ', D.DEPARTMENT_NAME);
END_FOR;

COMMIT;
FINISH;
end.
```

Example 2

The following programs demonstrate the use of the FOR statement to create a record stream. The programs:

- Declare a host language variable, *dept_name*, to be the same as CURRENT_INFO.DEPARTMENT using the DECLARE_VARIABLE clause

- Start a read-only transaction

- Prompt for a value for *dept_name*

- Create a record stream that consists of two passes of the CURRENT_INFO view

- Find the employee with the highest salary

- Print the salary and department name of that employee, and then the employee's last name

FOR Statement

C Program

```
#include <stdio.h>
DATABASE PERS = FILENAME "PERSONNEL";

extern void read_string();
DECLARE_VARIABLE dept_name SAME AS  CURRENT_INFO.DEPARTMENT;

main()
{
read_string("Department Name: ", dept_name, sizeof(dept_name));

READY PERS;
START_TRANSACTION READ_ONLY;

FOR CI1 IN CURRENT_INFO
   WITH CI1.DEPARTMENT = dept_name
   AND CI1.SALARY = (MAX CI2.SALARY OF CI2 IN CURRENT_INFO
                     WITH CI2.DEPARTMENT = dept_name)
      printf ("The biggest salary in department %s", dept_name);
      printf (" is $%f\n", CI1.SALARY);
      printf ("The rich employee is  %s", CI1.LAST_NAME);
END_FOR;

COMMIT;
FINISH;
}
```

Pascal Program

```
program for_in_rse (input,output);
DATABASE PERS = FILENAME 'PERSONNEL';

var
   DECLARE_VARIABLE dept_name SAME AS  CURRENT_INFO.DEPARTMENT;

begin
write   ('Department Name: ');
readln  (dept_name);

READY PERS;
START_TRANSACTION READ_ONLY;

FOR CI1 IN CURRENT_INFO
    WITH CI1.DEPARTMENT = dept_name
    AND CI1.SALARY = (MAX CI2.SALARY OF CI2 IN CURRENT_INFO
                      WITH CI2.DEPARTMENT = dept_name)
       writeln ('The biggest salary in department ',
                dept_name, ' is $', CI1.SALARY : 10 : 2);
       writeln ('The rich employee is ', CI1.LAST_NAME);
END_FOR; {CI1 IN EMPLOYEES}
```

```
COMMIT;
FINISH;
end.
```

Example 3

The following programs demonstrate the use of the FOR statement. The
programs:

- Sort the EMPLOYEES relation by last name (ascending order)

- Find and print information on all employees with degrees

- Use the NOT ANY clause to find those employees with a record stored in
  the DEGREES relation, but with no value stored in the degree_field

C Program

```
#include <stdio.h>
DATABASE PERS = FILENAME "PERSONNEL";

main()
{
READY PERS;
START_TRANSACTION READ_ONLY;

FOR E IN EMPLOYEES SORTED BY E.LAST_NAME
   FOR D IN DEGREES WITH D.EMPLOYEE_ID = E.EMPLOYEE_ID
      printf ("%s  %s\n", E.LAST_NAME, E.FIRST_NAME);
      printf ("%s  %s\n\n", D.DEGREE, D.DEGREE_FIELD);
   END_FOR;

   FOR FIRST 1 D IN DEGREES
      WITH NOT ANY D1 IN DEGREES
      WITH D1.EMPLOYEE_ID = E.EMPLOYEE_ID
         printf ("%s  %s", E.LAST_NAME, E.FIRST_NAME);
         printf ("no degree stored %s", RDB$MISSING(D.DEGREE_FIELD) );
   END_FOR;
END_FOR;

COMMIT;
FINISH;
}
```

Pascal Program

```
program outer_join (input,output);
DATABASE PERS = FILENAME 'PERSONNEL';

begin
READY PERS;
START_TRANSACTION READ_ONLY;
```

FOR Statement

```
FOR E IN EMPLOYEES SORTED BY E.LAST_NAME
   FOR D IN DEGREES WITH D.EMPLOYEE_ID = E.EMPLOYEE_ID
      writeln (E.LAST_NAME, '  ', E.FIRST_NAME);
      writeln (D.DEGREE, '  ', D.DEGREE_FIELD);
      writeln;
   END_FOR;

   FOR FIRST 1 D IN DEGREES
      WITH NOT ANY D1 IN DEGREES
      WITH D1.EMPLOYEE_ID = E.EMPLOYEE_ID
         writeln (E.LAST_NAME, ' ', E.FIRST_NAME);
         writeln ('no degree stored', '  ', RDB$MISSING(D.DEGREE_FIELD) );
   END_FOR;
END_FOR;

ROLLBACK;
FINISH;
end.
```

Example 4

The following programs demonstrate the use of the FOR statement and host language print statements to print a data type of varying text.

The C program:

- Declares a host language variable, *candidate_status*, to hold the value of the varying text field.

- Uses the macro, RDB$VARYING_TO_CSTRING, to copy the data from the database and store it in *candidate_status*. This macro is in the RDMLVAXC.H file, which is automatically included (#include) into your program by RDML.

- Prints the value for *candidate_status*.

The Pascal program requires no special macro to perform this operation. Pascal supports varying strings as a native data type.

C Program

```
#include <stdio.h>
DATABASE PERS = FILENAME "PERSONNEL";

main()
{
char candidate_status[255];

READY PERS;
START_TRANSACTION READ_ONLY;
```

```
FOR C IN CANDIDATES
   printf("%s  %s  %s\n", C.FIRST_NAME, C.MIDDLE_INITIAL, C.LAST_NAME);
   RDB$VARYING_TO_CSTRING(C.CANDIDATE_STATUS,candidate_status);
   printf("%s\n\n", candidate_status);
END_FOR;

COMMIT;
FINISH;
}
```

Pascal Program

```
program varying_text (input,output);
DATABASE PERS = FILENAME 'PERSONNEL';

begin
READY PERS;
START_TRANSACTION READ_ONLY;

FOR C IN CANDIDATES
   writeln (C.FIRST_NAME, C.MIDDLE_INITIAL, C.LAST_NAME);
   writeln (C.CANDIDATE_STATUS);
   writeln;
END_FOR;

COMMIT;
FINISH;
end.
```

6.14 FOR Segmented String Statement

The FOR segmented string statement forms a stream of segments from a segmented string field. A single segmented string field value is made up of multiple segments. To retrieve this value you must form a record stream that first retrieves the record that contains the segmented string field, and then form a stream of segments themselves. Thus, the process of retrieving a segmented string field involves retrieving the record that contains the segmented string field with either a FOR or START_STREAM statement, then retrieving the individual segments with a FOR statement with segmented strings. The first stream (formed by the FOR or START_STREAM statement) retrieves the records that contain the segmented string. The second stream (formed by the FOR statement with segmented strings) retrieves the individual segments that compose the segmented string field.

Format
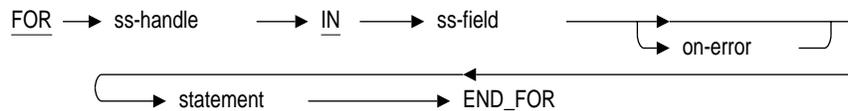
FOR ⟶ ss-handle ⟶ IN ⟶ ss-field ⟶⟶⟶ on-error ⟶⟶⟶

⟶ statement ⟶⟶ END_FOR

Arguments

ss-handle
A segmented string handle. A name that identifies the segmented string.

ss-field
A qualified field name that refers to a field defined with the SEGMENTED STRING data type. Note that this field name, like all field names in a FOR statement, must be qualified by its own context variable. This second context variable must match the variable declared in the outer FOR statement. See the Examples section.

on-error
The ON ERROR clause. Specifies host language statements or RDML statements or both to be performed if an error occurs during the FOR operation. See Section 6.17 for details.

FOR Segmented String Statement

assignment

Associates the two database variables with a value expression.

The database variables refer to the segment of a segmented string and its length. The special name for the segment can be either "VALUE" or "RDB$VALUE." The special name for the segment length can be either "LENGTH" or "RDB$LENGTH." You cannot assign any other database variables to the value expressions for segmented strings.

The assignment operator for RDML Pascal is ":=".

```
        .
        .
        .
   for linecnt := 0 to 2 do
        STORE SEG IN R.RESUME
            SEG  := document[linecnt];
            SEG.LENGTH := length(document[linecnt]);
         END_STORE;
        .
        .
        .
```

The assignment operator for RDML C is "=" or in this case the strcpy function.

```
        .
        .
        .
for (line = 0; line <= 2; line++)
             STORE LINE IN R.RESUME
                 strcpy(LINE.VALUE,document[line]);
                 LINE.LENGTH = strlen(LINE.VALUE);
             END_STORE;
        .
        .
        .
```

For more information, see the segmented string examples in this section and the value expression examples in Chapter 2.

FOR Segmented String Statement

Usage Notes

- The FOR statement with segmented strings must be embedded within a simple FOR . . . END_FOR block.

- Do not declare the host language variable to hold a segmented string field with the DECLARE_VARIABLE clause. The data type generated for a segmented string field is that of the segmented string identifier, which is the value that actually is stored in a segmented string field. For example, the following Pascal code might be used to store a RESUME field in the RESUMES relation. You should not declare the host language variable *document* with the DECLARE_VARIABLE clause.

```
FOR R IN RESUMES WITH R.EMPLOYEE_ID = '12345'
    FOR SEG IN R.RESUME
        writeln (SEG)
    END_FOR;
END_FOR;
```

- You cannot modify a subset of the strings contained in a segmented string field. You must replace the entire segmented string field. See Section 6.16, Example 3, for an example of modifying a record that contains a segmented string field.

- RDML defines a special name to refer to the segments of a segmented string. This value expression is equivalent to a field name; it names the "fields" or segments of the string. Furthermore, because segments can vary in length, RDML also defines a name for the length of a segment. You must use these value expressions to retrieve the length and value of a segment. These names are:

    - RDB$VALUE or VALUE

      The value stored in a segment of a segmented string

    - RDB$LENGTH or LENGTH

      The length in bytes of a segment

FOR Segmented String Statement

Examples

Example 1

The following programs demonstrate the use of the FOR statement to
retrieve segmented strings. Because the PERSONNEL database does not
have any segmented strings stored, the programs first store three strings
in the RESUME field of the RESUMES relation (see Section 6.26 for more
information on storing segmented strings). The programs retrieve the
segmented string using a nested FOR statement. The outer FOR statement
selects a record based on the EMPLOYEE_ID field. The inner FOR statement
prints each segmented string stored in the RESUME field for the selected
employee.

C Program

```
#include <stdio.h>
DATABASE PERS = FILENAME "PERSONNEL";

main()
{
int line;
char *document[3];

document[0] = "first line of resume  ";
document[1] = "second line of resume ";
document[2] = "last line of resume   ";

READY PERS;
START_TRANSACTION READ_WRITE;

STORE R IN RESUMES USING
   strcpy (R.EMPLOYEE_ID,"12345");
   for (line = 0; line <= 2; line++)
      STORE SEG IN R.RESUME
            strcpy(SEG.VALUE,document[line]);
            SEG.LENGTH = strlen(SEG.VALUE);
      END_STORE;
END_STORE;

FOR R IN RESUMES WITH R.EMPLOYEE_ID = "12345"
   FOR SEG IN R.RESUME
      printf("%s\n",SEG.VALUE);
   END_FOR;
END_FOR;

COMMIT;
FINISH;
}
```

FOR Segmented String Statement

Pascal Program

```
program segstr (input,output);
DATABASE PERS = FILENAME 'PERSONNEL';

type lines = varying [80] of char;
var linecnt  : integer;
 document : array [0..2] of lines;

begin

document[0] := 'first line of resume  ';
document[1] := 'second line of resume ';
document[2] := 'last line of resume   ';

READY PERS;
START_TRANSACTION READ_WRITE;

STORE R IN RESUMES USING
   R.EMPLOYEE_ID:= '12345';
   for linecnt := 0 to 2 do
      STORE SEG IN R.RESUME
         SEG := document[linecnt];
         SEG.LENGTH := length(document[linecnt]);
      END_STORE;
END_STORE;

FOR R IN RESUMES WITH R.EMPLOYEE_ID = '12345'
    FOR SEG IN R.RESUME
      writeln (SEG);
    END_FOR;
END_FOR;

COMMIT;
FINISH;
end.
```

6.15 GET Statement

The GET statement assigns values to host language variables in RDML programs.

Format

get-statement =
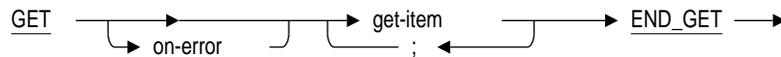
get-item =
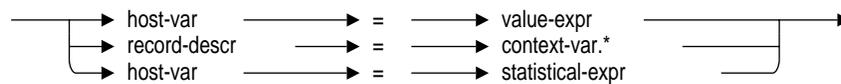
Arguments

get-item
The get-item clause includes an equal sign (=), a host language variable on the right-hand side of the equal sign, and a database value on the left-hand side of the equal sign. The database value derived from a value expression or statistical expression is assigned to the host language variable.

on-error
The ON ERROR clause. Specifies host language statements or RDML statements or both to be performed if an error occurs during the GET operation. See Section 6.17 for details.

record-descr
A valid host language record structure that contains an entry for each field in the relation. Each field of the record structure must match exactly the field names and data types of the fields in the Rdb relation referred to by the context variable. In C, the field names must be in lowercase type.

GET Statement

context-var
A context variable. A temporary name that you associate with a relation. You define a context variable in a relation clause. See Section 4.1 for more information.

host-var
A valid variable name declared in the host language program.

statistical-expr
A statistical expression. It calculates values based on a value expression for every record in the record stream.

Usage Notes

You can use the GET statement in four different ways:

- When you specify a record stream with the FOR or START_STREAM statement, you can use the GET statement to assign values from the current record in the stream to host language variables in your program. With the START_STREAM statement, you also need a FETCH statement to establish the current record in the stream.

- You can use the GET statement within a STORE operation to retrieve the values of the record currently being stored. This includes the use of GET . . . RDB$DB_KEY in a STORE . . . END_STORE block to retrieve the database key (dbkey) of a record just stored. If you use a GET statement in a STORE . . . END_STORE block, the GET statement must be the last statement before the END_STORE statement.

- You can use the GET statement alone, without a FOR, FETCH, or STORE statement, to retrieve the result of a statistical, conditional, or Boolean expression. The record stream is formed by the record selection expression within the statistical or conditional expression.

- You can use the GET * format of the GET statement to retrieve an entire record rather than just a field from a record. When you use the GET * statement you must first declare a record structure that contains all the fields in the relation. The host language record field names must match the database field names exactly. See Example 3.

Examples

Example 1

The following programs demonstrate the use of the GET statement with a statistical function. The examples store the value of the statistical function in the host language variable, *maxi*, then print this value.

C Program

```
#include <stdio.h>
DATABASE PERS = FILENAME "PERSONNEL";

DECLARE_VARIABLE maxi SAME AS PERS.CURRENT_INFO.SALARY;

main()
{
READY PERS;
START_TRANSACTION READ_ONLY;

GET
    maxi = MAX CI.SALARY OF CI IN CURRENT_INFO;
END_GET;

printf ("%f",maxi);
COMMIT;
FINISH;
}
```

Pascal Program

```
program max_function (input,output);
DATABASE PERS = FILENAME 'PERSONNEL';

DECLARE_VARIABLE maxi SAME AS PERS.CURRENT_INFO.SALARY;

begin
READY PERS;
START_TRANSACTION READ_ONLY;

GET
  maxi = MAX CI.SALARY OF CI IN CURRENT_INFO;
END_GET;

writeln (maxi:10:2);

COMMIT;
FINISH;
end.
```

GET Statement

Example 2

The following programs demonstrate the use of the GET statement with a conditional expression. The examples use the ANY conditional expression to find if any records in the SALARY_HISTORY relation have an amount greater than $50,000.00 in the SALARY_AMOUNT field. The GET statement places the result of the ANY expression in the host language variable, *who*. If a value over $50,000.00 is found, the programs display the message "Someone is not underpaid."

C Program

```
#include <stdio.h>

DATABASE PERS = FILENAME "PERSONNEL";

int who;

main()
{
READY PERS;
START_TRANSACTION READ_ONLY;

GET
 who = ANY SH IN SALARY_HISTORY WITH SH.SALARY_AMOUNT > 50000.00;
END_GET;

if (who)
    printf ("Someone is not underpaid \n");

COMMIT;
FINISH;
}
```

Pascal Program

```
program anycond (input,output);
DATABASE PERS = FILENAME 'PERSONNEL';

who : boolean;

begin
READY PERS;
START_TRANSACTION READ_WRITE;

GET
 who = ANY SH IN SALARY_HISTORY WITH SH.SALARY_AMOUNT > 50000.00;
END_GET;

if (who) then
   writeln ('Someone is not underpaid.');
```

```
COMMIT;
FINISH;
end.
```

Example 3

The following programs demonstrate the use of the GET * statement to retrieve
all the fields of a record. The examples declare a host language structure to
hold each field for the COLLEGES relation. The programs then use the FIRST
clause to find the first record in the COLLEGES relation with a college code
of HVDU. The GET * statement places the field values of this record in the
host language record structure. The programs then print the field values of the
retrieved record.

C Program

```
#include <stdio.h>
DATABASE PERS = FILENAME "PERSONNEL";

static struct
   {
   DECLARE_VARIABLE college_code SAME AS COLLEGES.COLLEGE_CODE;
   DECLARE_VARIABLE college_name SAME AS COLLEGES.COLLEGE_NAME;
   DECLARE_VARIABLE city SAME AS COLLEGES.CITY;
   DECLARE_VARIABLE state SAME AS COLLEGES.STATE;
   DECLARE_VARIABLE postal_code SAME AS COLLEGES.POSTAL_CODE;
   } colleges_record;

main()
{
READY PERS;
START_TRANSACTION READ_ONLY;

FOR FIRST 1 C IN COLLEGES
   WITH C.COLLEGE_CODE = "HVDU"
   GET
      colleges_record = C.*;
   END_GET;
END_FOR;

printf ("%s %s\n %s %s\n %s\n", colleges_record.college_code,
                                colleges_record.college_name,
                                colleges_record.city,
                                colleges_record.state,
                                colleges_record.postal_code);

COMMIT;
FINISH;
}
```

GET Statement

Pascal Program

```
program anycond (input,output);
DATABASE PERS = FILENAME 'PERSONNEL';

var
  colleges_record:
  RECORD
  DECLARE_VARIABLE college_code SAME AS COLLEGES.COLLEGE_CODE;
  DECLARE_VARIABLE college_name SAME AS COLLEGES.COLLEGE_NAME;
  DECLARE_VARIABLE city SAME AS COLLEGES.CITY;
  DECLARE_VARIABLE state SAME AS COLLEGES.STATE;
  DECLARE_VARIABLE postal_code SAME AS COLLEGES.POSTAL_CODE;
end;

begin
READY PERS;
START_TRANSACTION READ_WRITE;

FOR FIRST 1 C IN COLLEGES
    WITH C.COLLEGE_CODE = 'HVDU'
    GET
      colleges_record = C.*
    END_GET;
END_FOR;

writeln (colleges_record.college_code,' ',
         colleges_record.college_name);

writeln (colleges_record.city, ' ',
         colleges_record.state);

writeln (colleges_record.postal_code);

COMMIT;
FINISH;
end.
```

6.16 MODIFY Statement

The MODIFY statement changes the value in a field in one or more records from a relation in an open stream.

Format

MODIFY ⟶ context-var ⟶ USING
               ⟶ on-error

statement ⟶ END_MODIFY

context-var.* ⟶ = ⟶ record-descr

Arguments

context-var
A context variable. A temporary name that you associate with a relation. Define the context variable in the relation clause of the FOR or START_STREAM statement. See Section 4.1 for more information.

on-error
The ON ERROR clause. Specifies host language statements or RDML statements or both to be performed if an error occurs during the MODIFY operation. See Section 6.17 for details.

statement
Any valid RDML or host language statement to be executed within the MODIFY operation. Use a semicolon (;) at the end of each RDML, Pascal, or C statement.

context-var.*
A context variable declared in the relation clause of the FOR or START_STREAM statement. The MODIFY statement must appear after the FOR or START_STREAM statement and before the END_FOR or END_STREAM

MODIFY Statement

statement. The asterisk wildcard character (*) allows you to modify an entire record by assigning a record descriptor to the context-var* construct.

record-descr
A valid host language record descriptor that matches all the fields of the relation. Each field of the record descriptor must match exactly the field names and data types of the fields in the Rdb/VMS relation referred to by the context variable. Use a semicolon (;) at the end of the record descriptor.

Usage Notes

- Before using the MODIFY statement, you must start a read/write transaction and establish a record stream with a FOR statement or a START_STREAM statement.

- The context variable you refer to in a MODIFY statement must be the same as that defined in the FOR or START_STREAM statement.

- You can modify fields in only one record at a time.

- You can modify a record that contains a segmented string field, but you cannot not modify selected segments from the segmented string. You must use a STORE statement with segmented strings to change the segment contents. Example 3 demonstrates how to modify a record that contains a segmented string field.

- Because the effects of modifying some records in one relation and others in a second relation can be unpredictable, you should not modify records from views that refer to more than one relation.

- You can use the MODIFY * statement to modify all the fields in a relation. To use MODIFY *, you must first declare a host language record structure with field names that match the database field names exactly. Then put the field values that you want to replace into the host language record fields and modify the entire database record using the MODIFY * statement. See Example 4.

Examples

Example 1

The following programs demonstrate the use of the MODIFY statement with a host language variable. The programs:

- Declare a host language variable, *badge*, same as EMPLOYEES.EMPLOYEE_ID

- Prompt for a value for *badge*

- Prompt for a new status code

- Change the status code for the employee with the specified *badge*

The C program uses the function read_string to prompt for and receive a value for *badge*. For more information and the source code for read_string, see Appendix B.

C Program

```
#include <stdio.h>
DATABASE PERS = FILENAME "PERSONNEL";

extern void read_string();
static DECLARE_VARIABLE badge SAME AS EMPLOYEES.EMPLOYEE_ID;

main()
{
read_string ("Employee ID: ", badge, sizeof(badge));

READY PERS;
START_TRANSACTION READ_WRITE;

FOR E IN EMPLOYEES WITH E.EMPLOYEE_ID = badge
   MODIFY E USING
      strcpy(E.STATUS_CODE,"1");
   END_MODIFY;
END_FOR;

ROLLBACK;
FINISH;
}
```

MODIFY Statement

Pascal Program

```
program modify_with_host (input,output);
DATABASE PERS = FILENAME 'PERSONNEL';

var
    DECLARE_VARIABLE badge SAME AS EMPLOYEES.EMPLOYEE_ID;

begin
write  ('Employee ID: ');
readln (badge);

READY PERS;
START_TRANSACTION READ_WRITE;

FOR E IN EMPLOYEES WITH E.EMPLOYEE_ID = badge
    MODIFY E USING
        E.STATUS_CODE := '1';
    END_MODIFY;
END_FOR;

ROLLBACK;
FINISH;
end.
```

Example 2

The following programs demonstrate the use of the MODIFY statement to assign a new value to a field in a record stream. The programs create a record stream that consists of all the records in the JOB_HISTORY field with a department code of "MBMN". The MODIFY statement changes the value for SUPERVISOR_ID to "00167" for all the records in the record stream. Note that the C program uses the function pad_string to append trailing blanks and the null terminator to the employee ID. This ensures that the employee ID matches the length defined for the field. For more information and the source code for pad_string, see Appendix B. The writeln function in Pascal pads the employee ID for you.

C Program

```
#include <stdio.h>
DATABASE PERS = FILENAME "PERSONNEL";

extern void pad_string();

main()
{
READY PERS;
START_TRANSACTION READ_WRITE;
```

```
FOR JH IN JOB_HISTORY
   WITH JH.DEPARTMENT_CODE = "MBMN"
      MODIFY JH USING
         pad_string ("00167", JH.SUPERVISOR_ID, sizeof(JH.SUPERVISOR_ID));
      END_MODIFY;
END_FOR;

ROLLBACK;
FINISH;
}
```

Pascal Program

```
program modify_field (input,output);
DATABASE PERS = FILENAME 'PERSONNEL';

begin
READY PERS;
START_TRANSACTION READ_WRITE;

FOR JH IN JOB_HISTORY
   WITH JH.DEPARTMENT_CODE = 'MBMN'
      MODIFY JH USING
         JH.SUPERVISOR_ID := '00167';
      END_MODIFY;
END_FOR;

ROLLBACK;
FINISH;
end.
```

Example 3

The following programs demonstrate the use of the MODIFY statement to modify a record that contains a segmented string field. The programs:

- Store a resume for employee 00164.

- Print out this resume.

- Commit the transaction.

- Begin a second transaction.

- Modify the resume field by embedding a STORE statement within a MODIFY statement. This operation deletes the segmented string handle associated with the old resume and replaces it with a new segmented string handle.

- Print the new resume.

- Commit the transaction.

MODIFY Statement

C Program

```
#include <stdio.h>
DATABASE PERS = FILENAME "PERSONNEL";

main()
{
int line;
char *document[3];

document[0] = "first line of resume  ";
document[1] = "second line of resume ";
document[2] = "last line of resume   ";

READY PERS;
START_TRANSACTION READ_WRITE;

/* Store a resume for employee 00164 */

printf("Storing resume entry for employee 00164\n");

STORE R IN RESUMES USING
   strcpy (R.EMPLOYEE_ID,"00164");
   for (line = 0; line <= 2; line++)
      STORE SEG IN R.RESUME
         strcpy(SEG.VALUE,document[line]);
         SEG.LENGTH = strlen(SEG.VALUE);
      END_STORE;
END_STORE;

/* Read it back */

printf("Resume entry contains:\n");

FOR R IN RESUMES WITH R.EMPLOYEE_ID = "00164"
   FOR SEG IN R.RESUME
      printf("%s\n",SEG.VALUE);
   END_FOR;
END_FOR;

COMMIT;

/* Now modify it */

document[0] = "new first line of resume  ";
document[1] = "new second line of resume ";
document[2] = "new last line of resume   ";

START_TRANSACTION READ_WRITE;

printf("Modifying resume entry\n");
```

```
FOR R IN RESUMES WITH R.EMPLOYEE_ID = "00164"
  MODIFY R USING
    for (line = 0; line <= 2; line++)
      STORE SEG IN R.RESUME
          strcpy(SEG.VALUE,document[line]);
          SEG.LENGTH = strlen(SEG.VALUE);
      END_STORE;
   END_MODIFY;
END_FOR;

/* Read it back */

printf("Resume entry contains:\n");

FOR R IN RESUMES WITH R.EMPLOYEE_ID = "00164"
   FOR SEG IN R.RESUME
      printf("%s\n",SEG.VALUE);
   END_FOR;
END_FOR;

COMMIT;
FINISH;
}
```

Pascal Program

```
program modseg (input, output);

DATABASE FILENAME 'PERSONNEL';

const
    MAXCHARS = 80;
    MAXsegs = 3;

type
    LINERANGE = 1..MAXsegs;
    segs = varying[MAXCHARS] of char;

var
    linecnt : LINERANGE;
    document : array[LINERANGE] of segs;

begin

  document[1] := 'first line of resume';
  document[2] := 'second line of resume';
  document[3] := 'last line of resume';

  READY;

  START_TRANSACTION READ_WRITE;

  (* Store a resume for employee 00164 *)

  writeln('Storing resume entry for employee 00164');
```

MODIFY Statement

```
STORE R IN RESUMES USING
  R.EMPLOYEE_ID := '00164';  (*  Store EMPLOYEE_ID field *)
  for linecnt := 1 to MAXsegs do
   STORE LINE IN R.RESUME  (*  Store RESUME field segments *)
       LINE.VALUE := document[linecnt];
       LINE.LENGTH := length(document[linecnt]);
   END_STORE;
END_STORE;

(* Read it back *)

writeln('Resume entry contains:');

FOR R IN RESUMES WITH R.EMPLOYEE_ID = '00164'
  FOR LINE IN R.RESUME
   writeln(LINE);   (* Print resume segments *)
  END_FOR;
END_FOR;

COMMIT;

(* Now modify it *)

document[1] := 'new first line of resume';
document[2] := 'new second line of resume';
document[3] := 'new last line of resume';

START_TRANSACTION READ_WRITE;

writeln('Modifying resume entry');

FOR R IN RESUMES WITH R.EMPLOYEE_ID = '00164'
  MODIFY R USING
   for linecnt := 1 to MAXsegs do
       STORE LINE IN R.RESUME     (* Modify RESUME, erasing old segments *)
        LINE.VALUE := document[linecnt];
        LINE.LENGTH := length(document[linecnt]);
       END_STORE;
  END_MODIFY;
END_FOR;

(* Read it back *)

writeln('Resume entry contains:');

FOR R IN RESUMES WITH R.EMPLOYEE_ID = '00164'
  FOR LINE IN R.RESUME
   writeln(LINE);  (* Print new segments for RESUME *)
  END_FOR;
END_FOR;


COMMIT;

FINISH;

end.
```

MODIFY Statement

Example 4

The following programs demonstrate the use of the MODIFY * statement to modify a record in the COLLEGES relation. The programs:

- Declare a host language record structure with field names that match the relation field names

- Prompt the user for field values

- Modify the record

- Roll back the transaction

C Program

```
#include <stdio.h>
DATABASE PERS = FILENAME "PERSONNEL";

static struct
   {
   DECLARE_VARIABLE college_code SAME AS COLLEGES.COLLEGE_CODE;
   DECLARE_VARIABLE college_name SAME AS COLLEGES.COLLEGE_NAME;
   DECLARE_VARIABLE city         SAME AS COLLEGES.CITY;
   DECLARE_VARIABLE state        SAME AS COLLEGES.STATE;
   DECLARE_VARIABLE postal_code  SAME AS COLLEGES.POSTAL_CODE;
   } colleges_record;


extern void read_string();

main()
{
read_string ("Enter College Code: ", colleges_record.college_code,
                                    sizeof(colleges_record.college_code));
read_string ("Enter College Name: ", colleges_record.college_name,
                                    sizeof(colleges_record.college_name));
read_string ("Enter College City: ", colleges_record.city,
                                    sizeof(colleges_record.city));
read_string ("Enter College State: ",colleges_record.state,
                                    sizeof(colleges_record.state));
read_string ("Enter Postal Code: ",  colleges_record.postal_code,
                                    sizeof(colleges_record.postal_code));

READY PERS;
START_TRANSACTION READ_WRITE;

FOR C IN COLLEGES
 WITH C.COLLEGE_CODE = "HVDU"

 MODIFY C USING
      C.* = colleges_record;
 END_MODIFY;

END_FOR;
```

MODIFY Statement

```
ROLLBACK;
FINISH;
}
```

Pascal Program

```
program store_with_host_lang (input,output);
DATABASE PERS = FILENAME 'PERSONNEL';

var
  colleges_record:
  RECORD
  DECLARE_VARIABLE college_code SAME AS COLLEGES.COLLEGE_CODE;
  DECLARE_VARIABLE college_name SAME AS COLLEGES.COLLEGE_NAME;
  DECLARE_VARIABLE city         SAME AS COLLEGES.CITY;
  DECLARE_VARIABLE state        SAME AS COLLEGES.STATE;
  DECLARE_VARIABLE postal_code  SAME AS COLLEGES.POSTAL_CODE;
end;

begin
   writeln ('Enter College Code:');
   readln  (colleges_record.college_code);
   writeln ('Enter College Name:');
   readln  (colleges_record.college_name);
   writeln ('Enter College City:');
   readln  (colleges_record.city);
   writeln ('Enter College State:');
   readln  (colleges_record.state);
   writeln ('Enter College Postal Code:');
   readln  (colleges_record.postal_code);

   READY PERS;
   START_TRANSACTION READ_WRITE;

   FOR C IN COLLEGES
     WITH C.COLLEGE_CODE = 'HVDU'

     MODIFY C USING
       C.* = colleges_record;
     END_MODIFY;

   END_FOR;

   ROLLBACK;
   FINISH;
end.
```

Example 5

The following programs demonstrate the use of the MODIFY statement to change the date in a record selection expression. These programs:

- Assign a value to the DATE data type field using the context variable P.BIRTHDAY

- Change the date by assigning another value to the field

- Convert the date

- Perform the queries using the value assigned to E.BIRTHDAY

```
FOR E IN EMPLOYEES WITH E.BIRTHDAY = '28-feb-1990'
    printf("%s,%s\n",E.LAST_NAME,E.FIRST_NAME);
END_FOR;

FOR E IN EMPLOYEES WITH '1-JAN-1990'>E.BIRTHDAY
    printf("%s,%s\n",E.LAST_NAME,E.FIRST_NAME);
END_FOR;
```

C Program

The C program uses a format specification: E.BIRTHDAY = '27-feb-1990'

(Alternately, you could use a *memcpy* function. For an example of the *memcpy* function, see the C program example in Example 6.)

```
    #include <stdio.h>

DATABASE MINE = FILENAME "PERSONNEL.RDB";

main()
{

    READY MINE ;

    START_TRANSACTION READ_WRITE;
    STORE E IN EMPLOYEES USING
 strcpy (E.EMPLOYEE_ID , "79990");
 strcpy (E.LAST_NAME, "Smith        ");
 strcpy (E.FIRST_NAME , "michael         ");
 E.BIRTHDAY = '27-feb-1990';
 END_STORE;
     FOR P IN EMPLOYEES WITH P.EMPLOYEE_ID = '79990'
 MODIFY P USING
 P.BIRTHDAY = '28-feb-1990';
 END_MODIFY;
    END_FOR;

    FOR E IN EMPLOYEES WITH E.BIRTHDAY = '28-feb-1990'
        printf("%s,%s\n",E.LAST_NAME,E.FIRST_NAME);
    END_FOR;
```

MODIFY Statement

```
    FOR E IN EMPLOYEES WITH '1-JAN-1990'>E.BIRTHDAY
        printf("%s,%s\n",E.LAST_NAME,E.FIRST_NAME);
    END_FOR;

    COMMIT;
}
```

Pascal Program

```
program modify5p(output);

database filename 'PERSONNEL.RDB';

begin
 ready;
 start_transaction read_write;
 store e in employees
 using e.employee_id :='98116';
        e.birthday := '16-jan-1991';
        e.last_name := 'mecdiro';
        e.first_name := 'frank';
 end_store;

 for p in employees with p.employee_id = '98116'
 modify p using
  p.birthday := '26-jan-1991';
 end_modify;
 end_for;

 for e in employees with e.birthday = '26-jan-1991'
  writeln(e.employee_id, e.last_name, e.first_name);
 end_for;
 commit;
end.
```

Example 6

The following programs demonstrate how to assign a date from one field
of a record selection expression to another, and then modify the date using
an assignment statement. Both fields, in this case the START_DATE and
END_DATE fields, must be the same data type.

The following C program uses the *memcpy* function to copy the date from
one database field to another. In the *memcpy* function, "memcpy (J.JOB_
END,J.JOB_START,8);" means 8 bytes are copied. This example is in contrast
to the *strcpy* function, which is used to copy date text from one database field
to another. You can not use the *strcpy* function to copy dates between fields.

MODIFY Statement

C Program

```
#include <stdio.h>

DATABASE MINE = FILENAME "PERSONNEL.RDB";

main()
{
    READY MINE ;

    START_TRANSACTION READ_WRITE;
    STORE J IN JOB_HISTORY USING
 strcpy (J.EMPLOYEE_ID , "79990");
 J.JOB_START = '27-feb-1990';
 END_STORE;

    FOR J IN JOB_HISTORY WITH J.EMPLOYEE_ID = '79990'
 MODIFY J USING

 memcpy ( J.JOB_END,J.JOB_START,8);

 END_MODIFY;
    END_FOR;

    COMMIT;
}
```

Pascal Program

This program shows how date fields can be assigned to each other directly, as in the j.job_end := j.job_start assignment:

```
            program modify6p(output);
            database filename 'PERSONNEL.RDB';

var
 start_date: rdml$cddadt_type;

begin
 ready;
 start_transaction read_write;
 store j in job_history
 using j.employee_id :='98116';
       j.job_start := '16-jan-1991';
 end_store;

 for j in job_history with j.employee_id = '98116'
 modify j using
  j.job_end := j.job_start;
 end_modify;
 end_for;
 commit;
end.
```

6.17 ON ERROR Clause

The ON ERROR clause specifies the statements the host language performs if an error occurs during the execution of the associated RDML statement.

You can use the ON ERROR clause in all RDML statements except the DATABASE and DECLARE_STREAM statements.

Format on-error =

ON ERROR ────▶ statement ────▶ END_ERROR

Argument

statement

Any valid RDML or host language statement to be executed when an RDML error occurs. Use a semicolon (;) at the end of each RDML, Pascal, or C statement.

Usage Notes

- Error handling with RDML is accomplished through the ON ERROR clause and two program variables, RDB$STATUS and RDB$MESSAGE_VECTOR.

  Every routine returns a status value into a program variable that is declared by RDML. The status value is a longword systemwide condition value that identifies a unique message in the system message file. The returned condition value may indicate success, in which case data manipulation continues uninterrupted. Or, this value may signal an error, in which case control passes to the ON ERROR clause. RDML names this condition value RDB$STATUS and declares it to be a longword. RDB$STATUS is the second element of a twenty-longword array, RDB$MESSAGE_VECTOR, that RDML uses to pass information between the database and a C or Pascal program.

  When using C as the host language, declare each status value as a *globalvalue*.

When using Pascal as the host language, declare each status value as an [VALUE,EXTERNAL] INTEGER.

The use of these variables varies according to the Rdb/VMS or Rdb/ELN environments. See the *VAX Rdb/VMS Guide to Using RDO, RDBPRE, and RDML* or the *VAX Rdb/ELN Guide to Application Development* for more information about their use.

Examples

Example 1

The following programs demonstrate the use of the ON ERROR clause to identify I/O errors that occur during execution of the READY statement. The programs check the value of RDB$STATUS. If RDB$STATUS contains the status value "RDB$_BAD_DB_FORMAT" then the ON ERROR clause associated with the READY statement identifies this error and the programs print the informational message "I/O error at READY . . . Possibly because file not found." If the error is not an I/O error, the programs print the informational message "Unexpected Error, Application Terminating." In both cases, the program eventually terminates because a return or halt is performed.

C Program

```
#include <stdio.h>
globalvalue RDB$_BAD_DB_FORMAT;
DATABASE PERS = FILENAME "PERSONNEL";

main()
{
READY PERS
   ON ERROR
   if (RDB$STATUS == RDB$_BAD_DB_FORMAT)
      printf("I/O error at READY... Possibly because file not found\n");
   else
      {
      printf("Unexpected Error, Application Terminating\n");
      RDML$SIGNAL_ERROR(RDB$MESSAGE_VECTOR);
      }
   return;
   END_ERROR;

START_TRANSACTION READ_WRITE;

    /* perform some read/write operation */

COMMIT;
FINISH;
}
```

ON ERROR Clause

Pascal Program

```
program onerror (output);
DATABASE PERS = FILENAME 'PERSONNEL';

var
    RDB$_BAD_DB_FORMAT : [value,external] integer;

begin
READY PERS
   ON ERROR
   if (RDB$STATUS = RDB$_BAD_DB_FORMAT)
   then
      writeln ('I/O Error at READY... Possibly because file not found')
   else
      begin
      writeln ('Unexpected Error, Application Terminating');
      RDML$SIGNAL_ERROR(RDB$MESSAGE_VECTOR)
      end;
   halt;
   END_ERROR;

START_TRANSACTION READ_WRITE;

(* Perform some read/write operation *)

COMMIT;
FINISH;
end.
```

Example 2

The following programs demonstrate the use of the ON ERROR clause to identify lock errors that occur during execution of the START_TRANSACTION statement. The programs start a transaction using the NOWAIT option. This means that execution of the START_TRANSACTION statement causes a lock error if anyone else has a lock on the EMPLOYEES relation when you run the program. In this case, the program will print the message "database unavailable right now". The programs will try to access the database up to 100 more times before terminating the program.

If the error is not a lock error, the programs print the message "Unexpected Error, Application Terminating".

To illustrate this application, build it, and then run it simultaneously from two different terminals.

C Program

```
globalvalue RDB$_LOCK_CONFLICT;
globalvalue RDB$_DEADLOCK;

#include <stdio.h>
DATABASE PERS = FILENAME "PERSONNEL";

void handle_error()
{
if (RDB$STATUS == RDB$_LOCK_CONFLICT)
   printf("database unavailable right now\n");
      else
         {
         printf("Unexpected Error, Application Terminating\n");
         RDML$SIGNAL_ERROR(RDB$MESSAGE_VECTOR);
         }
   return;
}

void access_employees()
{
READY PERS
   ON ERROR
      handle_error();
      return;
   END_ERROR;

START_TRANSACTION READ_WRITE NOWAIT
   RESERVING EMPLOYEES FOR EXCLUSIVE WRITE
   ON ERROR
      handle_error();
      return;
   END_ERROR;

/* perform some read/write operation on the EMPLOYEES relation */
printf ("Accessing EMPLOYEES...\n");

COMMIT;
FINISH;
}

main()
{
int i;
for (i=0; i<=100; i++)
   access_employees();
}
```

ON ERROR Clause

Pascal Program

```
program onerror (output);
DATABASE PERS = FILENAME 'PERSONNEL';

var
    RDB$_LOCK_CONFLICT : [value,external] integer;
    i      : integer;
    error : boolean;

procedure handle_error;
begin
    if RDB$STATUS = RDB$_LOCK_CONFLICT
    then
        writeln ('database unavailable right now')
    else
        begin
        writeln ('Unexpected Error, Application Terminating');
        RDML$SIGNAL_ERROR(RDB$MESSAGE_VECTOR)
        end;
end;

begin
for i := 1 to 100 do
    begin
    error := FALSE;
    READY PERS;
    START_TRANSACTION READ_WRITE NOWAIT
        RESERVING EMPLOYEES FOR EXCLUSIVE WRITE
        ON ERROR
          handle_error;
          error := TRUE;
        END_ERROR;

    if not error then
        begin
        { perform some read/write operation on the EMPLOYEES relation }
        writeln ('Accessing EMPLOYEES...');

        COMMIT;
        FINISH;
        end;
    end;
end.
```

Example 3

The following programs demonstrate the use of the GOTO statement within the ON ERROR clause. These programs trap errors that occur during the execution of the READY statement.

C Program

```
#include <stdio.h>

DATABASE MINE = COMPILETIME FILENAME "PERSONNEL"
  RUNTIME FILENAME "NONEXISTENT.RDB"; /* does not exist */

main()
{
  READY MINE
   ON ERROR
    printf("Forced Error with READY...\n");
    goto ninetynine;
   END_ERROR;
  COMMIT;
  goto EXIT;
 ninetynine:
  printf("recovery finished\n");
EXIT:
  FINISH;
```

Pascal Program

```
program readytest1 (output);

DATABASE MINE = COMPILETIME FILENAME 'PERSONNEL'
  RUNTIME FILENAME 'NONEXISTENT.RDB'; {* does not exist *}

LABEL 99,EXIT;
begin

  READY MINE
   ON ERROR
    writeln('Forced Error with READY...');
    GOTO 99;
    END_ERROR;

    COMMIT;
    GOTO EXIT;
99:
    writeln('Error recovery finished');
EXIT:
   FINISH;

end.
```

ON ERROR Clause

Example 4

The following programs demonstrate the testing of RDB$STATUS *after* the END_ERROR statement. In order to test RDB$STATUS after the ON ERROR clause, you must define RDB$STATUS as RDB$MESSAGE_VECTOR[1].

C program

```
#include <stdio.h>
globalvalue RDB$_BAD_DB_FORMAT;
DATABASE MINE = COMPILETIME FILENAME "PERSONNEL"
  RUNTIME FILENAME "NONEXISTENT.RDB"; /* does not exist */

main()
{
  READY MINE
   ON ERROR
    if (RDB$STATUS == RDB$_BAD_DB_FORMAT)
       printf( "I/O error at READY ... Possibly because file not found\n");
    else
 {
 printf("Unexpected Error, Application Terminating\n");
      RDML$SIGNAL_ERROR(RDB$MESSAGE_VECTOR);
 }
    return;
   END_ERROR;

   RDB$STATUS = RDB$MESSAGE_VECTOR[1];
   if (RDB$STATUS == 1)
      {
 printf("There is no error in READY\n");
/*
 .
 .
*/
      }
   else
   printf("Error recovery finished\n");

   COMMIT;
   FINISH;
}
```

ON ERROR Clause

Pascal Program

```
program readytest2 (output);

DATABASE MINE = COMPILETIME FILENAME 'PERSONNEL'
  RUNTIME FILENAME 'NONEXISTENT.RDB'; {* does not exist *}
var
    RDB$_BAD_DB_FORMAT :[value,external] integer;

begin

  READY MINE
   ON ERROR
    if (RDB$STATUS = RDB$_BAD_DB_FORMAT) then
       writeln('I/O error at READY ... Possibly because file not found')
    else
       begin
 writeln('Unexpected Error, Application Terminating');
     RDML$SIGNAL_ERROR(RDB$MESSAGE_VECTOR);
       end;
   halt;
   END_ERROR;


   RDB$STATUS := RDB$MESSAGE_VECTOR[2];
   if (RDB$STATUS = 1) then
 begin
   writeln('There is no error in READY');
 end
    else
 begin
 writeln('Error recovery finished');
     RDML$SIGNAL_ERROR(RDB$MESSAGE_VECTOR);
 end;
   COMMIT;
   FINISH;
end.
```

6.18 PREPARE Statement

Use the PREPARE statement to tell Rdb/ELN that your application intends to commit a transaction. This causes Rdb/ELN to poll all concerned entities, both hardware and software, to make sure that the transaction can be committed without interruption. If it determines that no component stands in the way of committing the transaction, Rdb/ELN allows a COMMIT statement that has been issued to execute.

If you use the PREPARE statement in an Rdb/VMS environment, you will not receive an error message; the PREPARE statement has no effect in an Rdb/VMS environment.

Format
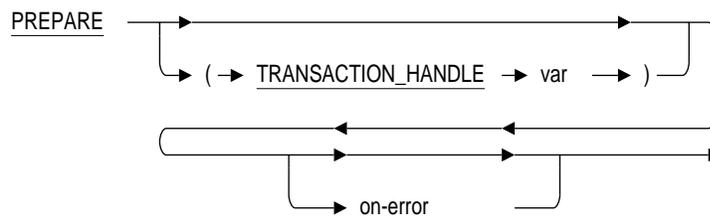
Arguments

TRANSACTION_HANDLE var
A TRANSACTION_HANDLE keyword followed by a host language variable that you associate with a transaction. If you do not supply a transaction handle explicitly, RDML supplies the default transaction handle.

on-error
The ON ERROR clause. Specifies host language statements or RDML statements or both to be performed if an error occurs during the PREPARE operation. See Section 6.17 for details.

Usage Notes

The PREPARE statement can be used in two distinct situations:

- When you have a transaction that affects multiple databases. In this case, the PREPARE statement checks that the transaction can be committed to all affected databases. If the transaction cannot be committed to all databases at once, you must roll back the transaction.

- When you need to synchronize database activity with external events before a transaction is committed. If the database activity and external events cannot be properly synchronized, you must roll back the transaction.

  Note that the PREPARE statement does not reserve database resources. It does, however, cause Rdb/ELN to poll all concerned entities, both hardware and software, to make sure that a transaction can be committed without any problems. If it determines that no component stands in the way of committing the transaction, Rdb/ELN allows a COMMIT statement that has been issued to execute.

  Your program logic should specify what to do in case the PREPARE statement fails.

Examples

Example 1

The following examples demonstrate the use of the PREPARE statement with a transaction handle. The programs:

- Are intended for an Rdb/ELN environment. A CONCURRENCY transaction and the PREPARE statement are ignored in an Rdb/VMS environment.

- Start a read/write concurrency transaction, SAL_INCREASE.

- Store a new JOBS record using the SAL_INCREASE transaction.

- Use the PREPARE statement to make sure that the transaction can be committed successfully in an Rdb/ELN environment.

Note that the C program uses the function pad_string. This function ensures that the values stored in each field have the correct number of trailing blanks to match the test size of the field. For more information and the source code for pad_string, see Appendix B.

PREPARE Statement

C Program

```
#include <stdio.h>
#define TRUE  1
#define FALSE 0

DATABASE PERS = FILENAME "PERSONNEL";

extern void pad_string();

main()
{
int SAL_INCREASE = 0;
RDML$HANDLE_TYPE success;

READY PERS;
success = TRUE;

START_TRANSACTION (TRANSACTION_HANDLE SAL_INCREASE) READ_WRITE;

STORE (TRANSACTION_HANDLE SAL_INCREASE) J IN JOBS USING
   pad_string ("TYPS", J.JOB_CODE, sizeof(J.JOB_CODE));
   pad_string ("1", J.WAGE_CLASS, sizeof(J.WAGE_CLASS));
   pad_string ("TYPIST", J.JOB_TITLE, sizeof(J.JOB_TITLE));
   J.MINIMUM_SALARY = 10000;
   J.MAXIMUM_SALARY = 17000;
END_STORE;

PREPARE (TRANSACTION_HANDLE SAL_INCREASE)
  ON ERROR
    success = FALSE;
    printf ("Sorry.  Cannot commit\n");
    printf ("Rollback of transaction about to begin ...\n");
  END_ERROR;

if (success == FALSE)
    {
    ROLLBACK (TRANSACTION_HANDLE SAL_INCREASE);
    }
else
    {
    COMMIT (TRANSACTION_HANDLE SAL_INCREASE);
    }
FINISH;
}
```

Pascal Program

```
program prepare_stmnt (input,output);
DATABASE PERS = FILENAME 'PERSONNEL';

var
  success : boolean;
  sal_increase : RDML$HANDLE_TYPE := 0;

begin
READY PERS;
success := TRUE;
```

```
START_TRANSACTION (TRANSACTION_HANDLE SAL_INCREASE) READ_WRITE CONCURRENCY;

STORE (TRANSACTION_HANDLE SAL_INCREASE) J IN JOBS USING
    J.JOB_CODE := 'TYPS';
    J.WAGE_CLASS := '1';
    J.JOB_TITLE := 'Typist';
    J.MINIMUM_SALARY := 10000;
    J.MAXIMUM_SALARY := 17000;
END_STORE;

PREPARE (TRANSACTION_HANDLE SAL_INCREASE)
  ON ERROR
    success := FALSE;
    writeln ('Sorry.  Cannot commit');
    writeln ('Rollback of transaction about to begin ...');
  END_ERROR;

if success = FALSE then
      ROLLBACK (TRANSACTION_HANDLE SAL_INCREASE)
else
      COMMIT (TRANSACTION_HANDLE SAL_INCREASE);

FINISH;
end.
```

6.19 READY Statement

The READY statement explicitly declares your intention to access one or more databases and causes an attach to the database.

Format
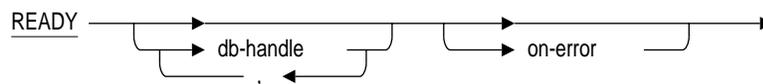
Arguments

db-handle
A database handle. A host language variable used to refer to a specific database your program uses. Specified in a DATABASE statement and declared by RDML.

on-error
The ON ERROR clause. Specifies host language statements or RDML statements or both to be performed if an error occurs during the READY operation. See Section 6.17 for details.

Usage Notes

- If you issue a READY statement without specifying a database handle, your application attaches to all databases declared in that module.

- Digital recommends that you use the /NODEFAULT_TRANSACTIONS qualifier when you preprocess your program. When you use the /NODEFAULT_TRANSACTIONS qualifier you must issue a READY statement to attach to the database. You can attach to one of many databases as you need it and then use the FINISH statement to detach from it when you are done. In this way, you do not have to allocate system resources to remain attached to all the required databases throughout the program.

- You do not have to use the READY statement to access a database. By default, a database attach occurs automatically the first time you refer to it. However, Digital recommends that you always issue a READY statement prior to accessing a database.

- You can use the READY statement to test the availability of a database. For example, you may want to check availability before your program prompts a user for input.

- When you use the READY statement, you can predict when the database attach is performed. If you do not use a READY statement, the first database access will cause an attach to occur (except when the /NODEFAULT_TRANSACTION qualifier is specified), and this may introduce a delay that is obvious to the user.

Examples

Example 1

The following program fragments demonstrate the use of the READY statement to open a database. The program fragments:

- Use the DATABASE statement to declare the PERSONNEL database

- Declare a database handle PERS for PERSONNEL

- Open the PERSONNEL database with the READY statement

- Close the database with the FINISH statement

C Program

```
#include <stdio.h>
DATABASE PERS = FILENAME "PERSONNEL";
                    .
                    .
                    .
main ()
{
READY PERS;
                    .
                    .
                    .
FINISH PERS;
}
```

READY Statement

Pascal Program

```
program empupdate;
DATABASE PERS = FILENAME 'PERSONNEL';
                      .
                      .
                      .
begin
READY PERS;
                      .
                      .
                      .

FINISH PERS;
end.
```

Example 2

The following program fragments demonstrate how to attach to two databases within the same program. The program fragments:

- Use the DATABASE statement to declare two databases, PERSONNEL and PAYROLL

- Declare database handles for both databases

- Attach to both databases

- Detach from each database

C Program

```
#include <stdio.h>
DATABASE PERS  = FILENAME "PERSONNEL";
DATABASE PAY   = FILENAME "WORK$DISK:PAYROLL";

main ()
{
                      .
                      .
                      .
READY PERS;
                      .
                      .
                      .
FINISH PERS;
                      .
                      .
                      .
```

```
READY PAY;
                 .
                 .
                 .
FINISH PAY;
                 .
                 .
                 .
READY PERS, PAY;
                 .
                 .
                 .
FINISH PERS, PAY;
}
```

Pascal Program

```
program new_employee;
DATABASE PERS  = FILENAME 'PERSONNEL';
DATABASE PAY   = FILENAME 'WORK$DISK:PAYROLL';
                 .
                 .
                 .
READY PERS;
                 .
                 .
                 .
FINISH PERS;
                 .
                 .
                 .
READY PAY;
                 .
                 .
                 .
FINISH PAY;
                 .
                 .
                 .
READY PERS, PAY;
                 .
                 .
                 .
FINISH PERS, PAY;
end.
```

6.20 REQUEST_HANDLE Clause

A request handle is a host language variable that identifies a compiled Rdb request. RDML generates request handles for statements that contain record selection expressions. In almost all cases it is unnecessary for you to explicitly specify request handles. However, if you choose to, you can specify a request handle to identify the requests that RDML generates in the following statements:

- FOR

- START_STREAM

- Boolean expressions (ANY, UNIQUE)

- STORE

- Statistical functions (AVERAGE, COUNT, MAX, MIN, TOTAL)

For the syntax diagram that shows the placement of the REQUEST_HANDLE in each of the RDML statements, see the section describing that statement.

Format

request-handle =

$$\longrightarrow (\longrightarrow \underline{REQUEST\_HANDLE} \longrightarrow host\text{-}variable \longrightarrow) \longrightarrow$$

Argument

host-variable
A valid host language variable. See Usage Notes.

Usage Notes

- Most applications do not require the use of, or benefit from, user-specified request handles. Unless you need to refer to a request handle directly (for example, you want to release a request prior to executing a FINISH statement) you probably do not need to use request handles. You may degrade performance if you use request handles unnecessarily.

REQUEST_HANDLE Clause

- Do not release a request unless it is absolutely necessary. If you release a request, yet continue to refer to that request, you force RDML to recompile the request each time you refer to it.

- RDML-supplied request handles improve the performance for an application program that repeats identical queries. A request handle serves as a pointer to the internal representation of a query. Request handles in an application cause Rdb to reuse this internal representation, reducing the run-time overhead associated with executing a query. *Note that Rdb uses request handles regardless of whether you specify handle names for the requests.*

- If you choose to explicitly declare a request handle in your program, the request handle must be:

  - Declared in the host language program as:

    - RDML$HANDLE_TYPE for Pascal

      ```
      DECLARE_VARIABLE OF name SAME AS PERS.EMPLOYEES.LAST_NAME;
      REQ1 : RDML$HANDLE_TYPE;
      ```

    - RDML$HANDLE_TYPE for C

      ```
      DECLARE_VARIABLE name SAME AS PERS.EMPLOYEES.LAST_NAME;
      extern long RDB$RELEASE_REQUEST();
      RDML$HANDLE_TYPE REQ1;
      ```

  - Initialized to zero before being used for the first time. Do not reinitialize a request prior to each time you refer to it (for example within a FOR loop). If you reinitialize a request to zero, RDML recompiles the request each time you refer to it.

  - Reinitialized to zero after a request is released, or after your program detaches from the associated database by issuing a FINISH statement.

- The value of a request handle is valid from the point when the associated query is made until the request is released, or until your program detaches from the database associated with that query by issuing a FINISH statement.

- If you are using modular programming techniques, do not issue a FINISH statement in one module and then attempt to use a request handle associated with the finished database in another module. Attempts to do so will result in the error message: BAD_REQ_HAND.

REQUEST_HANDLE Clause

- Each request has resources associated with it that are used by Rdb to store the internal representation of the request. Your program can release these resources in two ways:

  - By issuing a FINISH statement. This causes your program to detach from the database associated with the requests and releases the resources associated with all the requests for the finished database attach.

  - By issuing a call to RDB$RELEASE_REQUEST. This does not cause your program to detach from the database associated with the request.

    Before you issue a call to a RDB$RELEASE_REQUEST, you should declare it in C programs as shown in the following example:

    ```
    extern long RDB$RELEASE_REQUEST();
    ```

    You do not need to declare RDB$RELEASE_REQUEST in Pascal programs; it is declared for you in RDMLVPAS.PAS.

    To release a request in Pascal use:

    ```
    if not RDB$RELEASE_REQUEST(RDB$MESSAGE_VECTOR, request_handle)
    then RDML$SIGNAL_ERROR(RDB$MESSAGE_VECTOR);
    ```

    To release a request in C use:

    ```
    if ((RDB$RELEASE_REQUEST(RDB$MESSAGE_VECTOR, &request_handle) & 1) == 0)
        RDML$SIGNAL_ERROR(RDB$MESSAGE_VECTOR);
    ```

Examples

Example 1

The following programs demonstrate the use of the REQUEST_HANDLE clause in a FOR statement. They also show how to release a request. The programs:

- Declare the host language variable, REQ1, for a request handle and the local variable, *name*

- Initialize REQ1 to zero

- Assign a value to *name*

- Start a transaction

- Use the request handle in the FOR statement
- Release the request

Note *These programs merely show how to use and release a request. Do not, under any circumstances, routinely declare and release requests. Doing so will degrade performance.*

C Program

```c
#include <stdio.h>
DATABASE PERS = FILENAME "PERSONNEL";

DECLARE_VARIABLE name SAME AS PERS.EMPLOYEES.LAST_NAME;
extern long RDB$RELEASE_REQUEST();
RDML$HANDLE_TYPE REQ1;

main()
{
REQ1 = 0;
strcpy(name,"Gray");

READY PERS;
START_TRANSACTION READ_ONLY;

FOR (REQUEST_HANDLE REQ1) E IN PERS.EMPLOYEES
     WITH E.LAST_NAME = name
     printf("%s\n",E.FIRST_NAME);
END_FOR;


if ((RDB$RELEASE_REQUEST(RDB$MESSAGE_VECTOR, &REQ1) & 1) == 0)
 RDML$SIGNAL_ERROR(RDB$MESSAGE_VECTOR);

COMMIT;
FINISH;
}
```

Pascal Program

```pascal
program request (input,output);
DATABASE PERS = FILENAME 'PERSONNEL';

DECLARE_VARIABLE OF name SAME AS PERS.EMPLOYEES.LAST_NAME;
REQ1 : RDML$HANDLE_TYPE;

begin
REQ1 := 0;
name := 'Gray';

READY PERS;
START_TRANSACTION READ_ONLY;
```

REQUEST_HANDLE Clause

```
FOR (REQUEST_HANDLE REQ1) E IN PERS.EMPLOYEES
   WITH E.LAST_NAME = name
   writeln (E.FIRST_NAME);
END_FOR;

if not RDB$RELEASE_REQUEST(RDB$MESSAGE_VECTOR, REQ1)
then  RDML$SIGNAL_ERROR(RDB$MESSAGE_VECTOR);

COMMIT;

FINISH;
end.
```

6.21 ROLLBACK Statement

The ROLLBACK statement terminates a transaction and undoes all changes made to the database since the program's most recent START_TRANSACTION statement or since the start of the specified transaction.

If an application explicitly calls the DECdtm services to start a distributed transaction, it must also explicitly call DECdtm services to complete the transaction, rather than using the ROLLBACK statement.

See the *VAX Rdb/VMS Guide to Distributed Transactions* for more information on using the DECdtm service calls in distributed transactions.

Format
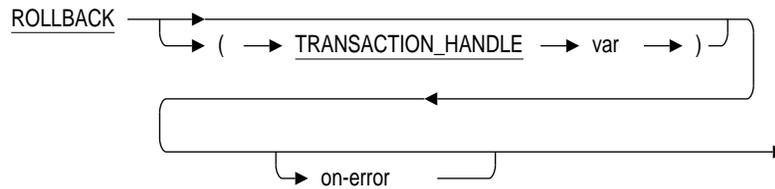
Arguments

TRANSACTION-HANDLE var
The TRANSACTION_HANDLE keyword followed by a host language variable you associate with a transaction. If you do not supply a handle name explicitly, Rdb uses the default transaction handle.

on-error
The ON ERROR clause. Specifies host language statements or RDML statements or both to be performed if an error occurs during the ROLLBACK operation. See Section 6.17 for details.

ROLLBACK Statement

Usage Notes

- The ROLLBACK statement affects all databases associated with the transaction.

- The ROLLBACK statement undoes all changes to data made with RDML ERASE, MODIFY, and STORE statements.

- The ROLLBACK statement with no argument will use the default transaction handle.

- If you start a transaction without specifying a transaction handle, you use the default transaction handle. There is one default transaction handle for the whole program. By default, when the RDML preprocessor encounters a statement without a transaction handle, it tests for the default transaction handle. If there is no default transaction, the RDML preprocessor starts one. Otherwise, the RDML preprocessor includes that statement in the existing default transaction.

  However, Digital recommends that you use the /NODEFAULT_ TRANSACTIONS qualifier when you preprocess your program. When you use the /NODEFAULT_TRANSACTIONS qualifier, RDML will not test for the default transaction handle on each statement it encounters without a transaction handle. This means that you must explicitly start and end your transaction (you do not have to specify a transaction handle). By explicitly starting and ending your transaction and using the /NODEFAULT_TRANSACTIONS qualifier, you can reduce overhead by eliminating the work RDML must do to test if a transaction has started.

- If you start a transaction and specify a transaction handle, you must use that transaction handle to roll back that transaction. The ROLLBACK statement automatically resets both user-specified and RDML-specified transaction handles to zero.

- The ROLLBACK statement also:

  - Closes open streams

  - Releases all locks in Rdb/VMS

  - Reduces all locks if you are using the CONSISTENCY option of the START_TRANSACTION statement in the Rdb/ELN environment. See the Rdb/ELN documentation set for details.

- Because the ROLLBACK statement closes open streams, you must not explicitly end the stream after a ROLLBACK statement. If you do end the stream with the END_STREAM clause of the START_STREAM statement, Rdb returns an error message.

- You cannot continue in a FOR loop after a ROLLBACK statement is issued.

Examples

Example 1

The following programs demonstrate the use of the ROLLBACK statement with a transaction handle to undo changes to the database made with the STORE statement. The programs:

- Start a read/write transaction, *SAL_INCREASE*

- Store a new JOBS record using the *SAL_INCREASE* transaction

- Use the ROLLBACK statement to undo the changes made to the database during the *SAL_INCREASE* transaction; that is, the new record is not stored in the database

Note that the C program uses the function pad_string. This function ensures that the values stored in each field have the correct number of trailing blanks to match the text size of the field. For more information and the source code for pad_string, see Appendix B.

C Program

```
#include <stdio.h>
DATABASE PERS = FILENAME "PERSONNEL";

extern void pad_string();

main()
{
int SAL_INCREASE = 0;

READY PERS;
START_TRANSACTION (TRANSACTION_HANDLE SAL_INCREASE) READ_WRITE;

STORE (TRANSACTION_HANDLE SAL_INCREASE) J IN JOBS USING
    pad_string ("TYPS", J.JOB_CODE, sizeof(J.JOB_CODE));
    pad_string ("1", J.WAGE_CLASS, sizeof(J.WAGE_CLASS));
    pad_string ("TYPIST", J.JOB_TITLE, sizeof(J.JOB_TITLE));
    J.MINIMUM_SALARY = 10000;
    J.MAXIMUM_SALARY = 17000;
END_STORE;
```

ROLLBACK Statement

```
ROLLBACK (TRANSACTION_HANDLE SAL_INCREASE);
FINISH;
}
```

Pascal Program

```
program rollback_trans (input,output);
DATABASE PERS = FILENAME 'PERSONNEL';
var sal_increase : [volatile] integer := 0;

begin
READY PERS;
START_TRANSACTION (TRANSACTION_HANDLE SAL_INCREASE) READ_WRITE;

STORE (TRANSACTION_HANDLE SAL_INCREASE) J IN JOBS USING
    J.JOB_CODE := 'TYPS';
    J.WAGE_CLASS := '1';
    J.JOB_TITLE := 'Typist';
    J.MINIMUM_SALARY := 10000;
    J.MAXIMUM_SALARY := 17000;
END_STORE;

ROLLBACK (TRANSACTION_HANDLE SAL_INCREASE);
FINISH;
end.
```

6.22 START_STREAM Statement, Declared

A declared START_STREAM statement starts a stream that was declared earlier in the module with the DECLARE_STREAM statement. A declared START_STREAM statement allows you to place the START_STREAM, FETCH, GET, and END_STREAM statements in any order within a program as long as they appear after the DECLARE_STREAM statement and are executed at run time in the order: START_STREAM, FETCH, GET, END_STREAM.

Format
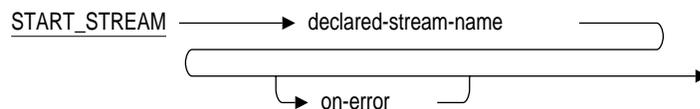
Arguments

declared-stream-name
A valid RDML name. This name must be the same name you use in the associated DECLARE_STREAM statement.

on-error
The ON ERROR clause. Specifies host language statements or RDML statements or both to be performed if an error occurs during the START_STREAM operation. See Section 6.17 for details.

Usage Notes

- Because the DECLARE_STREAM statement specifies the record selection expression and any transaction or request handles, the declared START_STREAM statement must not specify the record selection expression, a transaction handle, or a request handle.

- You can issue several declared START_STREAM statements in a module, and as long as you use the same declared stream name, they will all refer to the same stream.

START_STREAM Statement, Declared

- A stream is limited to one module.

- Once you have declared the stream (in the DECLARE_STREAM statement) and referred to this name in the START_STREAM statement, you should only use the stream name when you want to:

  - Fetch the next record with a FETCH statement.

  - Terminate the stream with the declared END_STREAM statement.

  For all other purposes you should use the context variables specified in the record selection expression of the associated DECLARE_STREAM statement. For example, if you want to modify records, you must use the context variable associated with the record in the record selection expression of the DECLARE_STREAM statement.

- Because the context variables specified in a DECLARE_STREAM statement remain visible until the end of the module, you should not reuse context variables defined in the record selection expression of the DECLARE_STREAM statement in other record selection expressions.

- Your program can use FOR statements or START_STREAM statements to establish record streams. The FOR statement is recommended. However, there are reasons for using a START_STREAM statement to create a record stream. You can use a START_STREAM statement to process multiple streams in parallel. Record streams created by the FOR statement can process nested streams, but not independent streams.

- You can process streams in the forward direction only. If you want to move the stream pointer back to a record that you already processed, you must end the stream and restart it or use database keys.

- The records in a stream are not returned in any specific order unless the record selection expression that creates the stream contains a SORTED BY clause.

Note   *Rdb retrieves the contents of any input host language variables in the record selection expression when you use the START_STREAM statement. Rdb cannot reexamine the host language variables until you end and restart the stream. Therefore, changing the value of a host language variable specified in the record selection expression that created the stream has no effect on an active stream.*

- The statements following a declared START_STREAM statement must include at least one FETCH statement before you access any record in the stream.

- Declared streams can overlap. For example:

```
START_STREAM A  . . .
        .
        .
        .
START_STREAM B  . . .
        .
        .
        .
END_STREAM A  . . .
        .
        .
        .
END_STREAM B  . . .
```

- Declared streams can be nested. For example:

```
START_STREAM A  . . .
        .
        .
        .
START_STREAM B  . . .
        .
        .
        .
END_STREAM B  . . .
        .
        .
        .
END_STREAM A  . . .
```

Examples

Example 1

The following programs demonstrate the use of the declared START_STREAM statement with the declared END_STREAM clause. The programs:

- Declare a stream *sal* with the DECLARE_STREAM statement that limits the stream to those records with a value less than $10,000.00 in the SALARY_AMOUNT field

- Start a read/write transaction

- Fetch the first record in the stream

- Modify that record so that the value in the SALARY_AMOUNT field is increased by 50 percent

START_STREAM Statement, Declared

- Fetch and modify records in the stream until all the records have been modified

- End the stream with the declared END_STREAM statement

C Program

```c
#include <stdio.h>
#define TRUE 1
#define FALSE 0

DATABASE PERS = FILENAME "PERSONNEL";

DECLARE_STREAM sal USING SH IN SALARY_HISTORY
     WITH SH.SALARY_AMOUNT LT 10000;

int end_of_stream;

main()
{
READY PERS;
START_TRANSACTION READ_WRITE;

    START_STREAM sal;

    FETCH sal
     AT END
        end_of_stream = TRUE;
    END_FETCH;

    while (! end_of_stream)
        {
        MODIFY SH USING
          SH.SALARY_AMOUNT = SH.SALARY_AMOUNT * (1.5);
        END_MODIFY;

        FETCH sal
          AT END
             end_of_stream = TRUE;
        END_FETCH;
        }

    END_STREAM sal;

    COMMIT;
    FINISH;
}
```

START_STREAM Statement, Declared

Pascal Program

```
program anycond (input,output);
DATABASE PERS = FILENAME 'PERSONNEL';

var
end_of_stream : boolean;

DECLARE_STREAM sal USING SH IN SALARY_HISTORY
     WITH SH.SALARY_AMOUNT LT 10000;

begin
    READY PERS;
    START_TRANSACTION READ_WRITE;

    START_STREAM sal;

    FETCH sal
     AT END
       end_of_stream := TRUE;
    END_FETCH;

    while not end_of_stream do
     begin
       MODIFY SH USING
         SH.SALARY_AMOUNT := SH.SALARY_AMOUNT * (1.5);
       END_MODIFY;

       FETCH sal
         AT END
           end_of_stream := TRUE;
       END_FETCH;

     end;

    END_STREAM sal;
    COMMIT;
    FINISH;
end.
```

6.23 START_STREAM Statement, Undeclared

The START_STREAM statement declares and starts a record stream. The START_STREAM statement:

- Forms a record stream from one or more relations. The record selection expression determines the records in the record stream.

- Places a pointer for that stream just before the first record in this stream.

You must then use the FETCH statement to fetch the next record in the stream and other RDML statements (for example, MODIFY and ERASE) to manipulate each record.

Note *Digital recommends that all programs use the declared START_STREAM statement (with the DECLARE_STREAM statement) in place of the undeclared START_STREAM statement. The declared START_STREAM statement provides all the features of the undeclared START_STREAM statement and provides more flexibility in programming than the undeclared START_STREAM statement.*

Format

Arguments

handle-options
A request handle, a transaction handle, or both.

REQUEST_HANDLE var
A REQUEST_HANDLE keyword followed by a host language variable. A request handle identifies a compiled Rdb/VMS request. If you do not supply a request handle explicitly, RDML generates a unique request handle for the compiled request. See Section 6.20 for more information on request handles.

TRANSACTION_HANDLE var
A TRANSACTION_HANDLE keyword followed by a host language variable. A transaction handle identifies a transaction. If you do not supply a transaction handle explicitly, RDML uses the default transaction handle.

stream-name
The stream that you create. The stream name must be a valid host language name.

rse
A record selection expression. A clause that defines specific conditions that individual records must meet before Rdb includes them in a record stream.

on-error
The ON ERROR clause. Specifies host language statements or RDML statements or both to be performed if an error occurs during the START_STREAM operation. See Section 6.17 for details.

Usage Notes

- Once you have named the stream, you should only refer to the stream-name when you want to:

  - Retrieve the next record with a FETCH statement

  - Terminate the stream with the END_STREAM statement

  For all other purposes you should use context variables. For example, if you want to modify records, you must use the context variable associated with the record selection expression of the START_STREAM statement.

START_STREAM Statement, Undeclared

- Any context variable names that you define with the START_STREAM statement are valid for the life of that stream only. Once you have defined a context variable in the record selection expression, you cannot reuse that context variable name elsewhere inside the START_STREAM . . . END_STREAM block. References to the context variable must occur between the keywords START_STREAM and END_STREAM. You can use the context variable name again outside that block.

- Your program can use FOR statements or START_STREAM statements to establish record streams. The FOR statement is recommended. However, there are reasons for using a START_STREAM statement to create a record stream. You can use a START_STREAM statement to process multiple streams in parallel. Record streams created by the FOR statement can process nested streams, but not independent streams.

- If you want to process multiple streams in parallel, you must declare transaction handles and specify the handles in the START_STREAM statement.

- You can process streams in the forward direction only. If you want to move the stream pointer back to a record that you already processed, you must end the stream and restart it (or use dbkeys).

- The records in a stream are not returned in any specific order unless the record selection expression that creates the stream contains a SORTED BY clause.

- Rdb retrieves the contents of any input host language variables in the record selection expression when you use the START_STREAM statement. Rdb cannot reexamine the host language variables until you end and restart the stream. Therefore, changing the value of a host language variable in the middle of an active stream has no effect on the records included in the record stream.

- The statements following a START_STREAM statement must include at least one FETCH statement before you access any record in the stream.

- Streams can overlap, for example:

```
START_STREAM A  . . .
        .
        .
        .
START_STREAM B  . . .
        .
        .
        .
END_STREAM A  . . .
        .
        .
        .
END_STREAM B  . . .
```

- Streams can be nested, for example:

```
START_STREAM A  . . .
        .
        .
        .
START_STREAM B  . . .
        .
        .
        .
END_STREAM B  . . .
        .
        .
        .
END_STREAM A  . . .
```

Examples

Example 1

The following programs:

- Create a record stream, CURRENT_INF_STREAM, that consists of the CURRENT_INFO record sorted by highest salary first

- Fetch the first record, thereby fetching the CURRENT_INFO record with the highest salary

- Display a message about that record

START_STREAM Statement, Undeclared

C Program

```
#include <stdio.h>
DATABASE PERS = FILENAME "PERSONNEL";

main()
{
READY PERS;
START_TRANSACTION READ_ONLY;

START_STREAM CURRENT_INF_STREAM USING
   CI IN CURRENT_INFO SORTED BY DESC CI.SALARY;
   FETCH CURRENT_INF_STREAM;
      printf ("%s makes the largest salary!\n", CI.LAST_NAME);
END_STREAM CURRENT_INF_STREAM;

COMMIT;
FINISH;
}
```

Pascal Program

```
program record_stream (input,output);
DATABASE PERS = FILENAME 'PERSONNEL';

begin
READY PERS;
START_TRANSACTION READ_ONLY;

START_STREAM CURRENT_INF_STREAM USING
   CI IN CURRENT_INFO SORTED BY DESC CI.SALARY;
   FETCH CURRENT_INF_STREAM;
      writeln (CI.LAST_NAME, ' makes the largest salary!');
END_STREAM CURRENT_INF_STREAM;

COMMIT;
FINISH;
end.
```

Example 2

The following programs demonstrate the use of the START_STREAM statement to create a record stream. The programs:

- Create a stream of all EMPLOYEES records sorted by LAST_NAME first

- Create a stream of all EMPLOYEES records sorted by FIRST_NAME first

- List the stream sorted by LAST_NAME in the left column

- List the stream sorted by FIRST_NAME in the right column

C Program

```c
#include <stdio.h>
DATABASE PERS = FILENAME 'PERSONNEL';

#define TRUE  1
#define FALSE 0
int END_OF_STREAM;

main()
{
READY PERS;
START_TRANSACTION READ_ONLY;

START_STREAM BY_LAST_NAME USING
   E1 IN EMPLOYEES SORTED BY E1.LAST_NAME, E1.FIRST_NAME;

START_STREAM BY_FIRST_NAME USING
   E2 IN EMPLOYEES SORTED BY E2.FIRST_NAME, E2.LAST_NAME;

/*The streams BY_LAST_NAME and BY_FIRST_NAME will contain the
 same number of records. It is only necessary to test
 for AT END once.*/

   END_OF_STREAM = FALSE;

   FETCH BY_LAST_NAME
   AT END
       END_OF_STREAM = TRUE;
   END_FETCH;

   if (!END_OF_STREAM)
      FETCH BY_FIRST_NAME;

   while (!END_OF_STREAM)
     {
      /*Alphabetical listing by last name down left column*/
      printf ("%s %s",E1.LAST_NAME, E1.FIRST_NAME);

      printf ("                    ");  /*skip 20 spaces*/

      /*Alphabetical listing by first name down right column*/
      printf ("%s %s\n",E2.FIRST_NAME, E2.LAST_NAME);

      FETCH BY_LAST_NAME
      AT END
         END_OF_STREAM = TRUE;
      END_FETCH;

      if (!END_OF_STREAM)
          FETCH BY_FIRST_NAME;

      }  /*while*/
```

START_STREAM Statement, Undeclared

```
END_STREAM BY_LAST_NAME;
END_STREAM BY_FIRST_NAME;

COMMIT;
FINISH;
}
```

Pascal Program

```
program two_record_streams (input,output);
DATABASE PERS = FILENAME 'PERSONNEL';

var
    end_of_stream : boolean;

begin
READY PERS;
START_TRANSACTION READ_ONLY;

START_STREAM BY_LAST_NAME USING
    E1 IN EMPLOYEES SORTED BY E1.LAST_NAME, E1.FIRST_NAME;

START_STREAM BY_FIRST_NAME USING
    E2 IN EMPLOYEES SORTED BY E2.FIRST_NAME, E2.LAST_NAME;

{* The streams BY_LAST_NAME and BY_FIRST_NAME will contain the
 exact same number of records. It is only necessary to test
 for AT END once. *}

    end_of_stream := false;

    FETCH BY_LAST_NAME
    AT END
        end_of_stream := true;
    END_FETCH;

    if not end_of_stream then
        FETCH BY_FIRST_NAME;

    while not end_of_stream do begin

        {* Alphabetical listing by last name down left column *}
        write (E1.LAST_NAME, ' ', E1.FIRST_NAME);

        write (' ' : 20);  {skip 20 spaces}

        {* Alphabetical listing by first name down right column *}
        writeln (E2.FIRST_NAME, ' ', E2.LAST_NAME);

        FETCH BY_LAST_NAME
        AT END
            end_of_stream := true;
        END_FETCH;
```

```
      if not end_of_stream then
          FETCH BY_FIRST_NAME;

  end; {* WHILE *}

END_STREAM BY_FIRST_NAME;
END_STREAM BY_LAST_NAME;

COMMIT;
FINISH;
end.
```

6.24 START_TRANSACTION Statement

The START_TRANSACTION statement initiates a transaction. A transaction is a group of statements whose changes can be made permanent or undone as a unit. Either all the statements that modify records within a transaction become permanent when the transaction is completed, or no one of them does. If you end the transaction with the COMMIT statement, all the changes within the transaction become permanent. If you end the transaction with the ROLLBACK statement, all changes made within the transaction are undone.

If an application starts a distributed transaction by explicitly calling the DECdtm service SYS$START_TRANS, it must complete the transaction by calling the SYS$END_TRANS or SYS$ABORT_TRANS system service. See the *VAX Rdb/VMS Guide to Distributed Transactions* for more information on using the SYS$END_TRANS or SYS$ABORT_TRANS system service call in distributed transactions.

Format
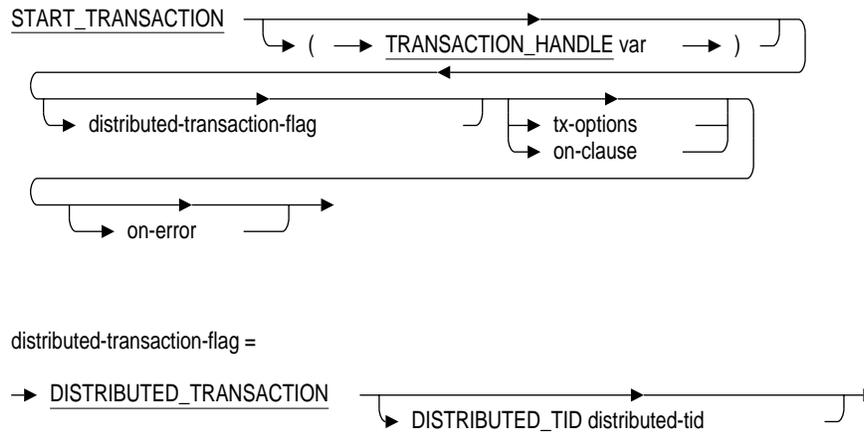
START_TRANSACTION Statement

tx-options =

```
      ┌──▶ BATCH_UPDATE ──────────────────────────────────────────┐
      │                                                            │
      │    ┌──▶ READ_ONLY ──┐   ┌──▶ WAIT ──┐   ┌──▶ CONCURRENCY ┐ │
      ├────┤                ├───┤           ├───┤                ├─┤
      │    └──▶ READ_WRITE ─┘   └──▶ NOWAIT ─┘   └──▶ CONSISTENCY ┘ │
      │                                                            │
      │         ┌──────────────────◀──────────────────┐           │
      │    ┌──▶ EVALUATING ──▶ evaluating-clause ──┬───┴───────────┤
      │    │                  ┌─────────────┐      │               │
      │    │                  └──────▶──────┘      │               │
      │    │                        ,             │                │
      │    │    ┌──────────────────◀──────────────────┐           │
      └────┴──▶ RESERVING ──▶ reserving-clause ──┬───┴────────────┘
                             ┌─────────────┐      │
                             └──────▶──────┘      │
                                   ,
```

on-clause =

```
      ┌──▶ ON ──▶ db-handle ──┬──▶ USING ──▶ ( ┬──▶ tx-options ──┬──▶ ) ──▶
      │            ┌──────┐    │                └──▶ DEFAULTS ────┘
      │            └──◀───┘    │
      │               ,        │
      │                        │
      └────────────▶ AND ◀─────┘
```

evaluating-clause =

```
      ┌──────────────────────────┐
      └──▶ db-handle ──▶ . ──▶ ──┴── constraint ──┐
                                                   │
            ┌──────────────◀──────────────────────┘
            │
            └──▶ AT ──┬──▶ VERB_TIME ───┬──▶
                      └──▶ COMMIT_TIME ─┘
```
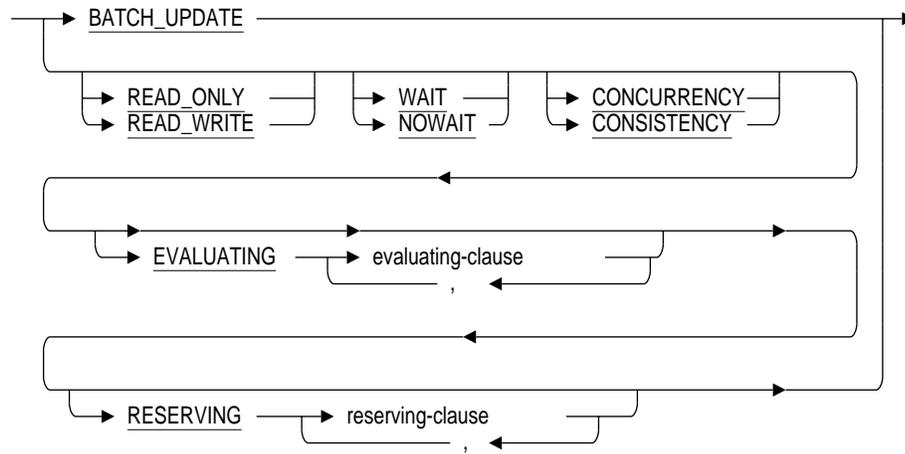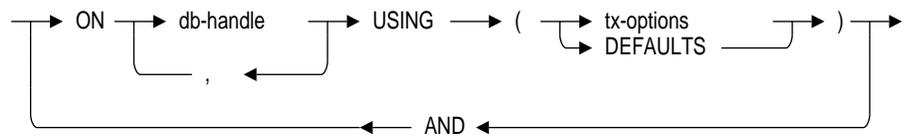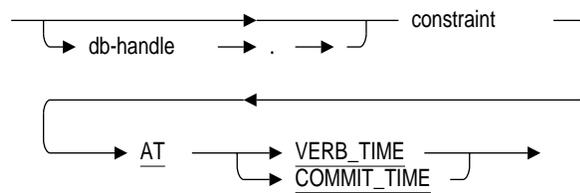
START_TRANSACTION Statement

reserving-clause =
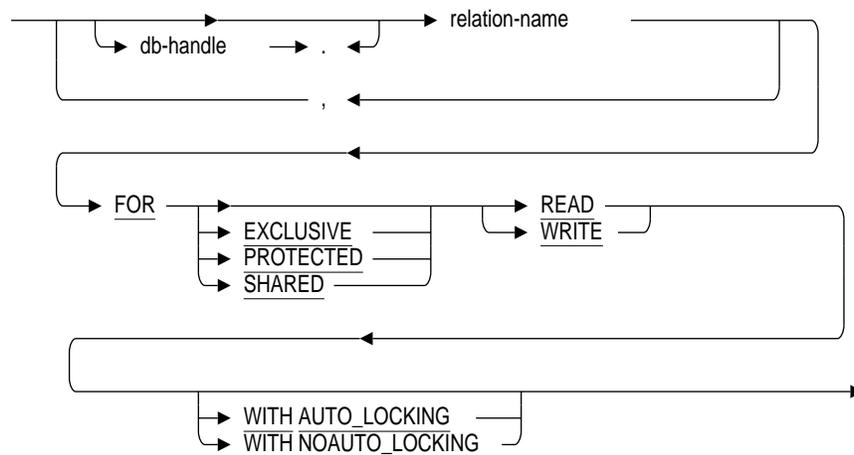
Arguments

TRANSACTION_HANDLE var

A TRANSACTION_HANDLE keyword followed by a host language variable. A transaction handle identifies a transaction. If you do not supply a transaction handle name explicitly, Rdb uses the default transaction handle.

If you specify a transaction handle in a START_TRANSACTION statement, you must also specify the same transaction handle on any operations that relate to that transaction (for example, COMMIT, FINISH, FOR, ROLLBACK, START_STREAM, and STORE statements).

DISTRIBUTED_TRANSACTION

You use this clause to specify the distributed transaction of which your transaction will be a part.

A distributed transaction is a transaction that uses more than one database handle. Examples of a distributed transaction include:

- A transaction that attaches more than once to a single Rdb/VMS database

- A transaction that attaches to two or more Rdb/VMS databases

START_TRANSACTION Statement

■ A transaction that attaches to more than one database management system (an Rdb/VMS database and a VAX DBMS database, for example)

Distributed transactions are managed by DECdtm services. DECdtm services assign a unique distributed transaction identifier (distributed TID) to each distributed transaction when the distributed transaction is started.

For more information on using distributed transactions, see the *VAX Rdb/VMS Guide to Distributed Transactions*.

DISTRIBUTED_TID distributed-tid
A keyword followed by a host language variable. You use the distributed-tid variable to hold the value of the distributed TID that DECdtm services generate and return to the application. DECdtm services use the TID to distinguish the databases involved in a particular distributed transaction.

When your application invokes the DECdtm services implicitly, you do not need to specify the DISTRIBUTED_TID keyword.

When your application explicitly calls the SYS$START_TRANS system service, you must specify the DISTRIBUTED_TID keyword. The distributed-tid variable is an octaword (16 bytes) that you should declare and initialize to zero at the beginning of your application.

tx-options
Transaction options. Allows you to specify the type of transaction you want, when you want constraints to be evaluated, and which relations you intend to access.

on-clause
Allows you to specify a particular database and the tx-options to be applied to the transaction that accesses that database attach.

BATCH_UPDATE
READ_ONLY
READ_WRITE
Declares what you intend to do with the transaction as a whole. READ_ONLY is the default. The effects of these transaction modes depend on the system you are using. Refer to the *VAX Rdb/VMS RDO and RMU Reference Manual* if you are using Rdb/VMS. Refer to the *VAX Rdb/ELN Reference Manual* if you are using Rdb/ELN.

START_TRANSACTION Statement

CONSISTENCY
CONCURRENCY
These options specify the consistency mode of the transaction:

- CONSISTENCY is the default. This mode guarantees that when all transactions complete by committing or rolling back, the effect on the database is the same as if all transactions were run sequentially. In Rdb/VMS, CONSISTENCY is the only option.

- CONCURRENCY is a high-throughput option for Rdb/ELN databases that guarantees that no transaction sees data written by another active transaction. The concurrency algorithm reduces system overhead, thereby improving overall performance while still guaranteeing a high level of data consistency (although not as high as the consistency mode).

WAIT
NOWAIT
These options specify what your transaction will do if it needs resources that are locked by another transaction:

- WAIT is the default. It causes your transaction to wait until the necessary resources are released or Rdb detects deadlock.

- With NOWAIT, Rdb will return an error if the resources you need are not immediately available, thereby forcing you to roll back your transaction.

evaluating-clause
Supported by Rdb/VMS only. Allows you to specify the point at which the named constraints are evaluated. If you specify VERB_TIME, they are evaluated when the data manipulation statement is issued. If you specify COMMIT_TIME, they are evaluated when the COMMIT statement executes. The evaluating clause is allowed syntactically, but is ignored, with read-only transactions.

constraint
The name of an Rdb/VMS constraint.

reserving-clause
Allows you to specify the relations you plan to use and attempts to lock those relations for your access.

START_TRANSACTION Statement

In general, include all the relations your transaction will access. If you use the WITH AUTO_LOCKING option (the default), constraints and triggers defined on the reserved relations will be able to access additional relations that do not appear in the list of reserved relations. The WITH AUTO_LOCKING option will not work for other relations not referenced in the reserving clause.

Note *If you use the RESERVING clause and the WITH NOAUTO_LOCKING option, you can access only those relations that you have explicitly reserved. If you access multiple databases with a single START_TRANSACTION statement and use the RESERVING clause for one or more databases, you can access only the reserved relations in a database for which you reserve relations.*

WITH AUTO_LOCKING (default)
WITH NOAUTO_LOCKING
Rdb/VMS automatically locks any relations referenced within a transaction unless you specify the optional WITH NOAUTO_LOCKING clause. The default is WITH AUTO_LOCKING.

on-error
The ON ERROR clause. Specifies host language statements or RDML statements or both to be performed if an error occurs during the START_TRANSACTION operation. See Section 6.17 for details.

db-handle
A database handle. A host language variable used to refer to a specific database your program uses. Optionally qualifies relation-name with a database handle. This option is required if you access two or more databases that share relations with the same name.

relation-name
The name of the relation to be used during the transaction.

READ (default)
WRITE
The Rdb lock type. This keyword declares what you intend to do with the relations you have reserved:

- READ reserves the specified relations for read-only access

- WRITE reserves the specified relations for read/write access

START_TRANSACTION Statement

EXCLUSIVE
PROTECTED
SHARED (default)
The Rdb/VMS share modes. The keyword you choose determines what operations you allow others to perform on the relations you are reserving. For read-only transactions, EXCLUSIVE and PROTECTED are syntactically allowed, but are ignored. The CONSISTENCY mode and the choice of read-only or read/write determines the kind of locking that is necessary.

Table 6–3 summarizes the share modes for both Rdb/ELN and Rdb/VMS.

Table 6–3   VAX Rdb/ELN and Rdb/VMS Share Modes

SHARED	Other users can work with the same relations as you. Depending on the option they choose, they can have read-only or read/write access to the database.
PROTECTED	Other users can read the relations you are using. They cannot have write access.
EXCLUSIVE	Other users cannot read or write to records from the relations included in your transaction. If another user refers to the same relation in a START_TRANSACTION statement, Rdb/VMS denies access to that user.

Usage Notes

- There are several levels of defaults for START_TRANSACTION. In general, Digital recommends that you use explicit START_TRANSACTION statements, specifying READ_WRITE or READ_ONLY, a list of relations in the RESERVING clause, and a share mode and lock type for each relation. Table 6–4 summarizes the defaults for each option and combination of options.

START_TRANSACTION Statement

Table 6–4   Defaults for the START_TRANSACTION Statement

Option	Default
Transaction Mode:	
READ_ONLY READ_WRITE BATCH_UPDATE	If you omit the START_TRANSACTION statement (or specify the START_TRANSACTION statement but do not specify a transaction mode), then RDML starts a read-only transaction (unless you have specified the RDML /NODEFAULT_TRANSACTIONS qualifier). Note that if the statement is a STORE, MODIFY, or ERASE statement, the result is an error, because you cannot update the database in a read-only transaction.
Lock Specification:	
RESERVING	If you do not specify a reserving option of a RESERVING clause, the default is SHARED_READ.
	If you specify a read/write transaction and do not include a RESERVING clause, Rdb determines the lock specification for each relation when it is first accessed with a data manipulation statement.
	If you specify a read/write transaction and include a RESERVING clause, the default share mode is SHARED.
	If you use the WITH AUTO_LOCKING option of the RESERVING clause (the default), Rdb/VMS determines the lock specification for each relation accessed within the transaction when the relation is first accessed with a data manipulation statement from a constraint or trigger.
	If you do not specify a transaction mode but do include a RESERVING clause, the default share mode is SHARED.
	If you specify a read-only transaction, the default is SHARED_READ, whether or not you specify a RESERVING clause.

(continued on next page)

START_TRANSACTION Statement

Table 6–4 (Cont.)   Defaults for the START_TRANSACTION Statement

Option	Default
Share Mode:	
SHARED PROTECTED EXCLUSIVE	The default is SHARED.
Lock Type:	
READ WRITE	If you specify a read/write transaction, the default is WRITE. If you specify a read-only transaction mode, READ is the default and only allowed lock type.
Concurrency Option:	
CONSISTENCY CONCURRENCY	CONSISTENCY is the default (and for Rdb/VMS, the only meaningful option).
Wait Mode:	
WAIT NOWAIT	WAIT is the default.
Evaluating Clause:	
VERB_TIME COMMIT_TIME	By default, Rdb/VMS evaluates each constraint at the time specified in the DEFINE CONSTRAINT definition. If the constraint definition does not specify when the constraint should be checked, the definition default is CHECK ON UPDATE (VERB_TIME).

- If you issue a data manipulation language statement (DML) without issuing a START_TRANSACTION statement first, Rdb automatically starts a read-only transaction for you. However, Digital recommends that you always explicitly start a transaction statement with the START_TRANSACTION statement. If you issue a DML statement, such as a GET or FOR statement, and then try to use the START_TRANSACTION statement, you will get an error message warning that a transaction is already in progress.

- Use of the /NODEFAULT_TRANSACTIONS qualifier requires that you issue a START_TRANSACTION statement prior to any DML statement. If you are using Rdb/VMS, see the *VAX Rdb/VMS Guide to Using RDO, RDBPRE, and RDML* for details. See the Rdb/ELN documentation set if you are using Rdb/ELN.

- You cannot specify the ROLLBACK statement as the action to be taken if an error occurs during the START_TRANSACTION operation. If an error occurs during this operation, no transaction exists; therefore, there is no transaction to roll back.

- If you choose not to use the default transaction handle, you must explicitly declare the transaction handle you use in your program. See Section 6.27 for more information on the TRANSACTION_HANDLE clause.

- Read-only consistency transactions are automatically started as read-only concurrency transactions in Rdb/ELN. Therefore it does not make sense to start a read-only transaction with CONSISTENCY. (This is not the case in Rdb/VMS, which does not provide CONCURRENCY.)

- In an Rdb/ELN environment, the choice of CONSISTENCY or CONCURRENCY affects the throughput of both your program and the programs of other users.

Examples

Example 1

The following statement starts a transaction in C and Pascal programs with the following characteristics:

- Uses the default transaction handle

- CONSISTENCY mode in both Rdb/VMS and Rdb/ELN

- WAIT option (by default)

- Read-only access (by default)

```
START_TRANSACTION;
```

Example 2

The following statement starts a transaction in C and Pascal programs in the Rdb/ELN environment with the following characteristics:

- Read/write access

- CONCURRENCY mode

- WAIT option (by default)

```
START_TRANSACTION READ_WRITE CONCURRENCY;
```

START_TRANSACTION Statement

Example 3

The following statements start a transaction with these characteristics:

- Read/write access

- CONSISTENCY mode

- WRITE access for the named relations (the transaction will wait until these relations are available at this level of access)

C Statements

```
DATABASE RDBDEMO = FILENAME "RDBDEMO.RDB";
DATABASE FINANCE = FILENAME "DDP_FINANCES";
                    .
                    .
                    .
START_TRANSACTION READ_WRITE CONSISTENCY
   RESERVING RDBDEMO.EMPLOYEES,
             RDBDEMO.SALARY_HISTORY,
             FINANCE.EMPLOYEES
   FOR WRITE;
```

Pascal Statements

```
DATABASE RDBDEMO = FILENAME 'RDBDEMO.RDB';
DATABASE FINANCE = FILENAME 'DDP_FINANCES';
                    .
                    .
                    .
START_TRANSACTION READ_WRITE CONSISTENCY
   RESERVING RDBDEMO.EMPLOYEES,
             RDBDEMO.SALARY_HISTORY,
             FINANCE.EMPLOYEES
   FOR WRITE;
```

Example 4

The following statements start a transaction with these characteristics:

- Read/write access

- WITH AUTO_LOCKING

- EXCLUSIVE access for the named relations (the transaction will automatically lock the relations that the triggers and constraints associated with this relation will need to access)

START_TRANSACTION Statement

C Example

```
#include <stdio.h>
DATABASE PERS = FILENAME 'PERSONNEL';

main()
{

START_TRANSACTION READ_ONLY
RESERVING EMPLOYEES FOR EXCLUSIVE READ WITH AUTO_LOCKING;

if (ANY E IN EMPLOYEES WITH E.STATE = "MA")
    printf("Someone lives in Massachusetts and AL works exclusive in NH .\n");

ROLLBACK;

FINISH;

}
```

Pascal Example

```
program startwithal4p (input,output);
DATABASE PERS = FILENAME 'PERSONNEL';

(* Program to test autolocking qualifier*)

var
   DECLARE_VARIABLE emp_id SAME AS EMPLOYEES.EMPLOYEE_ID;

begin
write  ('Employee_ID: ');
readln (emp_id);



READY PERS;
START_TRANSACTION READ_WRITE RESERVING EMPLOYEES FOR EXCLUSIVE WRITE
WITH AUTO_LOCKING;

FOR E IN EMPLOYEES
   WITH E.EMPLOYEE_ID = emp_id
     writeln ('Employee ID =  ', E.EMPLOYEE_ID);
   ERASE E;
    writeln ('Employee ID =  ', E.EMPLOYEE_ID, 'should succeed -- autolocking');
END_FOR;

ROLLBACK;
end.
```

6.25 STORE Statement

The STORE statement inserts a record into an existing relation. You can add a record to only one relation with a single STORE statement. The statements between the keywords STORE and END_STORE form a context block. You cannot store records into views defined by any of the following record selection expression clauses:

- WITH

- CROSS

- REDUCED

- FIRST

Note *Trying to store into views that were defined with any of the preceding clauses could cause ambiguous results in your database.*

Format
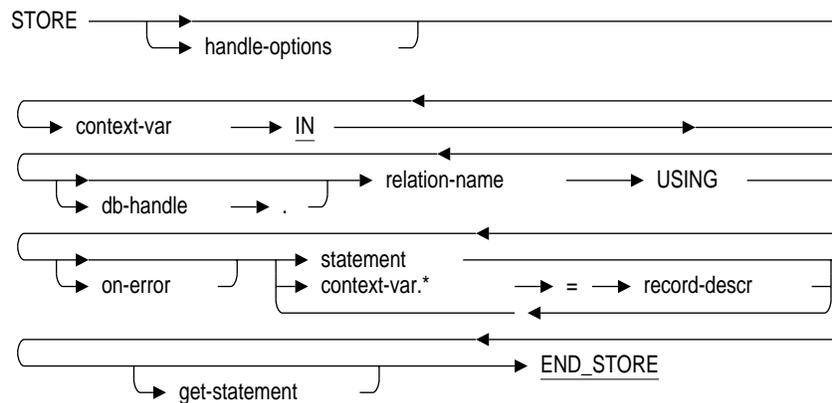
```
STORE ──────┬──────────────────┬──────────────────────────────────
            └──▶ handle-options ┘
 ┌──────────────────────────────────◀──────────────────────────┐
 └──▶ context-var ──▶ IN ─────────────────────────────────────▶
 ┌──────────────────────────◀───────────────────────────────┐
 └──┬─────────────────┬──▶ relation-name ──────▶ USING ──────▶
    └──▶ db-handle ──▶ . ┘
 ┌────────────────────────────◀───────────────────────────────┐
 └──┬──────────────┬──┬──▶ statement ────────────────────────────┐
    └──▶ on-error ─┘  └──▶ context-var.* ──┬──▶ = ──▶ record-descr ┤
                                            └────────◀───────────┘
 ┌──────────────────────────────◀──────────────────┐
 └──┬──────────────────┬──▶ END_STORE
    └──▶ get-statement ─┘
```
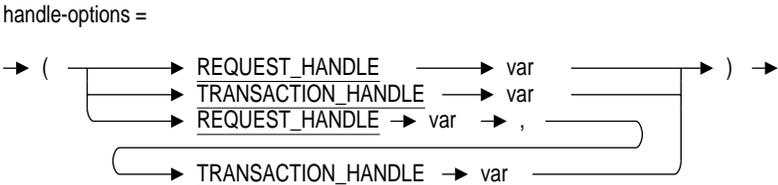
handle-options =

Arguments

handle-options
A request handle, a transaction handle, or both.

REQUEST_HANDLE var
A REQUEST_HANDLE keyword followed by a host language variable. A request handle identifies a compiled Rdb request. If you do not supply a request handle explicitly, RDML generates a unique request handle for the compiled request. See Section 6.20 for more information on request handles.

TRANSACTION_HANDLE var
A TRANSACTION_HANDLE keyword followed by a host language variable. A transaction handle identifies a transaction. If you do not supply a transaction handle explicitly, RDML uses the default transaction handle.

context-var
A context variable. A temporary name that you associate with a relation. You define a context variable in a relation clause.

db-handle
A database handle. A host language variable used to refer to a specific database your program specifies.

relation-name
The name of a relation in a database.

on-error
The ON ERROR clause. Specifies host language statements or RDML statements or both to be performed if an error occurs during the STORE operation. See Section 6.17 for details.

STORE Statement

statement
Any valid RDML or host language statement to be executed during the
STORE operation. Use a semicolon (;) at the end of each RDML, Pascal, or C
statement.

record-descr
A record descriptor matching all of the fields of the relation. Each field of the
record descriptor must match exactly the field names and data types of the
fields in the Rdb relation referred to by the context variable.

get-statement
The GET statement. If you use a GET statement within a STORE block, it
must be the last statement before the END_STORE keyword.

Usage Notes

- You can use the STORE statement to manipulate dates in international
  formats. See Section 1.1.6 for more information and Section 6.16 for
  examples.

- Do not embed host language multipath statements (such as the C switch
  statement or the Pascal case statement) in the STORE statement; this may
  lead to unpredictable results. The problem occurs when a field is referred
  to but not used at run time. This is because RDML assumes that any
  field (qualified by the appropriate context variable) mentioned within the
  STORE . . . END_STORE block is going to be updated.

  In the following example, if the program falls through to case 2 at run
  time, the FIRST_NAME field will be modified even though FIRST_NAME
  is not mentioned in case 2. Upon seeing the fields referred to in case 1,
  RDML sets up a buffer for both the FIRST_NAME and LAST_NAME fields
  to be sent to the database. Because case 2 does not supply data for the
  FIRST_NAME field, RDML sends to the database whatever happens to be
  in the buffer for the first name field.

  The following Pascal code will cause unpredictable results:

```
STORE E IN EMPLOYEES USING
   case i of
     1: E.LAST_NAME = 'Smith';
        E.FIRST_NAME = 'Andrew';

     2: E.LAST_NAME = 'Jones';
   end;
END_STORE
```

When different fields are referred to in a multipath statement, the RDML statement should be embedded in the host language multipath statement as shown in the following Pascal example:

```
case i of

  1: STORE E IN EMPLOYEES USING
       E.LAST_NAME = 'Smith';
       E.FIRST_NAME = 'Andrew';
     END_STORE;

  2: STORE E IN EMPLOYEES USING
       E.LAST_NAME = 'Jones';
     END_STORE;

end;
```

- You can use any valid format of the GET statement within the bounds of the STORE . . . END_STORE block. However, the GET statement must be the last statement before the END_STORE keyword.

- You may find it particularly useful to use the GET statement to place the database key (dbkey) of the record you just stored into a host language variable. Use the GET . . . RDB$DB_KEY construct to assign the value of the dbkey to the host language variable.

- If you do not supply a value for every field in the relation in which you are storing a record, that fields for which no values are supplied are marked as missing.

- The STORE * statement lets you manipulate database values at the record level rather than at the field level. You can store all the fields in a relation with the STORE * statement. To use STORE *, you must first declare a record structure that contains all the fields in the relation, with record field names that match the database field names exactly. See Example 4.

Note *Trying to store records into views that were defined with any of the preceding clauses could cause unexpected results.*

Examples

Example 1

The following programs demonstrate the use of the STORE statement and interactive programming to add a new record to the COLLEGES relation. The programs:

- Prompt for user input

STORE Statement

- Start a read/write transaction
- Store the user-supplied values in the relation
- Roll back the stored record from the database

Note that the C program uses the function read_string to prompt for user input and to hold these values. This function pads the input values with the necessary number of blanks to match the text size of each field. For more information on read_string, see Appendix B. The readln function in Pascal pads strings for you.

C Program

```
DATABASE PERS = FILENAME "PERSONNEL";

extern void read_string();
static DECLARE_VARIABLE coll_code SAME AS COLLEGES.COLLEGE_CODE;
static DECLARE_VARIABLE coll_name SAME AS COLLEGES.COLLEGE_NAME;
static DECLARE_VARIABLE coll_city SAME AS COLLEGES.CITY;
static DECLARE_VARIABLE coll_state SAME AS COLLEGES.STATE;
static DECLARE_VARIABLE post_code SAME AS COLLEGES.POSTAL_CODE;

main()
{
read_string ("Enter College Code: ", coll_code, sizeof(coll_code));
read_string ("Enter College Name: ", coll_name, sizeof(coll_name));
read_string ("Enter College City: ", coll_city, sizeof(coll_city));
read_string ("Enter College State: ",coll_state, sizeof(coll_state));
read_string ("Enter Postal Code: ", post_code, sizeof(post_code));

READY PERS;
START_TRANSACTION READ_WRITE;

STORE C IN COLLEGES USING
    strcpy (C.COLLEGE_CODE, coll_code);
    strcpy (C.COLLEGE_NAME, coll_name);
    strcpy (C.CITY, coll_city);
    strcpy (C.STATE, coll_state);
    strcpy (C.POSTAL_CODE, post_code);
END_STORE;

ROLLBACK;
FINISH;
}
```

Pascal Program

```
program store_with_host_lang (input,output);
DATABASE PERS = FILENAME 'PERSONNEL';

var
DECLARE_VARIABLE coll_code SAME AS COLLEGES.COLLEGE_CODE;
DECLARE_VARIABLE coll_name SAME AS COLLEGES.COLLEGE_NAME;
DECLARE_VARIABLE coll_city SAME AS COLLEGES.CITY;
DECLARE_VARIABLE coll_state SAME AS COLLEGES.STATE;
DECLARE_VARIABLE post_code SAME AS COLLEGES.POSTAL_CODE;

begin
writeln ('Enter College Code:');
readln  (coll_code);
writeln ('Enter College Name:');
readln  (coll_name);
writeln ('Enter College City:');
readln  (coll_city);
writeln ('Enter College State:');
readln  (coll_state);
writeln ('Enter College Postal Code:');
readln  (post_code);

READY PERS;
START_TRANSACTION READ_WRITE;

STORE C IN COLLEGES USING
   C.COLLEGE_CODE := coll_code;
   C.COLLEGE_NAME := coll_name;
   C.CITY := coll_city;
   C.STATE := coll_state;
   C.POSTAL_CODE := post_code;
END_STORE;

ROLLBACK;
FINISH;
end.
```

Example 2

The following programs demonstrate the use of the STORE statement with a record selection expression supplying the value for one of the fields. The programs:

- Start a read/write transaction

- Assign literal values to all fields in the JOBS relation except the MAXIMUM_SALARY field

- Use the FIRST FROM statement to find the first record in the JOBS relation that has a wage class of 1

STORE Statement

- Assign the maximum salary from this selected record to the MAXIMUM_ SALARY field for the record being stored

- Store these values in the relation

- Roll back the record from the database

Note that the C program uses the function pad_string to prompt for user input and to store the values in the relation. This function pads the input values with the necessary number of blanks to match the text size of each field. For more information, and the source code for pad_string, see Appendix B.

C Program

```
#include <stdio.h>
DATABASE PERS = FILENAME "PERSONNEL";

extern void pad_string();
main()
{
READY PERS;
START_TRANSACTION READ_WRITE;

STORE J1 IN JOBS USING
    pad_string ("CLNR", J1.JOB_CODE, sizeof(J1.JOB_CODE));
    pad_string ("1", J1.WAGE_CLASS, sizeof(J1.WAGE_CLASS));
    pad_string ("Cleaner",J1.JOB_TITLE, sizeof(J1.JOB_TITLE));
    J1.MINIMUM_SALARY = 8000;
    J1.MAXIMUM_SALARY = (FIRST J2.MAXIMUM_SALARY FROM J2 IN JOBS
                            WITH J2.WAGE_CLASS = "1");
END_STORE;

ROLLBACK;
FINISH;
}
```

Pascal Program

```
program store_with_assign (input,output);
DATABASE PERS = FILENAME 'PERSONNEL';

begin
READY PERS;
START_TRANSACTION READ_WRITE;
```

```
STORE J1 IN JOBS USING
   J1.JOB_CODE := 'CLNR';
   J1.WAGE_CLASS := '1';
   J1.JOB_TITLE := 'Cleaner';
   J1.MINIMUM_SALARY := 8000;
   J1.MAXIMUM_SALARY := (FIRST J2.MAXIMUM_SALARY FROM J2 IN JOBS
                              WITH J2.WAGE_CLASS = '1');
END_STORE;

ROLLBACK;
FINISH;
end.
```

Example 3

The following programs demonstrate the use of the STORE statement to store
VARYING TEXT data types.

The C program uses the function pad_string to store the values in the fields
that are defined as text data types. This function appends trailing blanks to
these values. This ensures that the values match the length defined for the
field. For more information and the source code for pad_string, see Appendix B.
The C program also uses a macro, RDB$CSTRING_TO_VARYING to store a
value in a field defined as a varying text data type. This macro is defined in
RDMLVAXC.H, which RDML automatically includes into your program. The
Pascal program does not require the use of any special functions to store either
varying text data types or fixed-length data types; in both cases, the Pascal
assignment operator is sufficient.

C Program

```
#include <stdio.h>
DATABASE PERS = FILENAME "PERSONNEL";

extern void pad_string();
main()
{
READY PERS;
START_TRANSACTION READ_WRITE;

STORE C IN CANDIDATES USING
   pad_string ("Clarkson", C.LAST_NAME, sizeof(C.LAST_NAME));
   pad_string ("Joel", C.FIRST_NAME, sizeof(C.FIRST_NAME));
   pad_string ("R", C.MIDDLE_INITIAL, sizeof(C.MIDDLE_INITIAL));
   RDB$CSTRING_TO_VARYING ("Available part time until June 15th",
                              C.CANDIDATE_STATUS);
END_STORE;
```

STORE Statement

```
ROLLBACK;
FINISH;
}
```

Pascal Program

```
program store_varying (input,output);
DATABASE PERS = FILENAME 'PERSONNEL';

begin
READY PERS;
START_TRANSACTION READ_WRITE;

STORE C IN CANDIDATES USING
   C.LAST_NAME := 'Clarkson';
   C.FIRST_NAME := 'Joel';
   C.MIDDLE_INITIAL := 'R';
   C.CANDIDATE_STATUS := 'Available part time until June 15th';
END_STORE;

ROLLBACK;
FINISH;
end.
```

Example 4

The following programs demonstrate the use of the STORE * statement to store varying text in a COLLEGES record. The programs declare a host language record structure that contains a field for each field in the COLLEGES relation. After the user specifies the field values, they are stored in the database with the STORE * statement.

The C program uses the function read_string to prompt for and to read values into host language variables. For more information and the source code for read_string, see Appendix B.

C Program

```
#include <stdio.h>
DATABASE PERS = FILENAME "PERSONNEL";

static struct
   {
   DECLARE_VARIABLE college_code SAME AS COLLEGES.COLLEGE_CODE;
   DECLARE_VARIABLE college_name SAME AS COLLEGES.COLLEGE_NAME;
   DECLARE_VARIABLE city SAME AS COLLEGES.CITY;
   DECLARE_VARIABLE state SAME AS COLLEGES.STATE;
   DECLARE_VARIABLE postal_code SAME AS COLLEGES.POSTAL_CODE;
   } colleges_record;


extern void read_string();
```

```
main()
{
read_string ("Enter College Code: \n", colleges_record.college_code,
                                sizeof(colleges_record.college_code));
read_string ("Enter College Name: \n", colleges_record.college_name,
                                sizeof(colleges_record.college_name));
read_string ("Enter College City: \n", colleges_record.city,
                                sizeof(colleges_record.city));
read_string ("Enter College State: \n",colleges_record.state,
                                sizeof(colleges_record.state));
read_string ("Enter Postal Code: \n",  colleges_record.postal_code,
                                sizeof(colleges_record.postal_code));

READY PERS;
START_TRANSACTION READ_WRITE;

STORE C IN COLLEGES USING
      C.* = colleges_record;
END_STORE;

COMMIT;
FINISH;
}
```

Pascal Program

```
program store_with_host_lang (input,output);
DATABASE PERS = FILENAME 'PERSONNEL';

var
  colleges_record:
  RECORD
  DECLARE_VARIABLE college_code SAME AS COLLEGES.COLLEGE_CODE;
  DECLARE_VARIABLE college_name SAME AS COLLEGES.COLLEGE_NAME;
  DECLARE_VARIABLE city SAME AS COLLEGES.CITY;
  DECLARE_VARIABLE state SAME AS COLLEGES.STATE;
  DECLARE_VARIABLE postal_code SAME AS COLLEGES.POSTAL_CODE;
end;

begin
   writeln ('Enter College Code:');
   readln  (colleges_record.college_code);
   writeln ('Enter College Name:');
   readln  (colleges_record.college_name);
   writeln ('Enter College City:');
   readln  (colleges_record.city);
   writeln ('Enter College State:');
   readln  (colleges_record.state);
   writeln ('Enter College Postal Code:');
   readln  (colleges_record.postal_code);

   READY PERS;
   START_TRANSACTION READ_WRITE;
```

STORE Statement

```
STORE C IN COLLEGES USING
  C.* = colleges_record;
END_STORE;

COMMIT;
FINISH;
end.
```

6.26 STORE Statement with Segmented Strings

The STORE statement with segmented strings inserts a segment into a segmented string.

Format
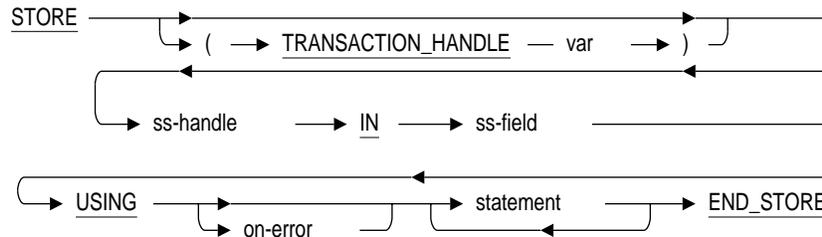
Arguments

ss-handle
A segmented string handle. A name that identifies the segmented string.

ss-field
A qualified field name that refers to a field defined with the SEGMENTED STRING data type. Note that this field name, like all field names in a FOR statement, must be qualified by its own context variable. This second context variable must match the context variable declared in the outer FOR statement. See the Examples section.

on-error
The ON ERROR clause. Specifies host language statements or RDML statements or both to be performed if an error occurs during the STORE operation. See Section 6.17 for details.

assignment
An RDML or host language statement that associates the database variables with a value expression.

STORE Statement with Segmented Strings

The database variables refer to the segment of a segmented string and its length. The special name for the segment can be either "VALUE" or "RDB$VALUE". The special name for the segment length can be either "LENGTH" or "RDB$LENGTH". You cannot assign any other database variables to the value expressions for segmented strings.

The assignment operator for RDML Pascal is ":=".

```
   .
   .
   .
   for linecnt := 0 to 2 do
      STORE SEG IN R.RESUME
           SEG  := document[linecnt];
           SEG.LENGTH := length(document[linecnt]);
      END_STORE;
   .
   .
   .
```

The assignment operator for RDML C is "=" or strcpy.

```
   .
   .
   .
for (line = 0; line <= 2; line++)
            STORE LINE IN R.RESUME
                strcpy(LINE.VALUE,document[line]);
                LINE.LENGTH = strlen(LINE.VALUE);
            END_STORE;
   .
   .
   .
```

For more information, see the segmented string examples in this section.

STORE Statement with Segmented Strings

Usage Notes

- The STORE statement with segmented strings must be embedded within a simple STORE . . . END_STORE block or within a (MODIFY . . . END_MODIFY) block.

- Do not declare the host language variable to hold a segmented string field with the DECLARE_VARIABLE clause. The data type generated for a segmented string field is that of the segmented string identifier, which is the value that actually is stored in a segmented string field. For example, the following Pascal code might be used to store a RESUME field in the RESUMES relation. You should not declare the host language variable *document* with the DECLARE_VARIABLE clause.

```
STORE R IN RESUMES USING
  R.EMPLOYEE_ID = '12345'
  for linecount := 0 to 2 do
    STORE SEG IN R.RESUME USING
        SEG.VALUE := document[lincnt];
        SEG.LENGTH := length(document[linecnt]);
    END_STORE;
END_STORE;
```

- RDML defines a special name to refer to the segments of a segmented string. This value expression is equivalent to the field name; it names the segments of the string. Furthermore, because segments can vary in length, RDML also defines a name for the length of a segment. You must use these value expressions to retrieve the length and value of a segment. These names are:

  - RDB$VALUE or VALUE

    The value stored in a segment of a segmented string

  - RDB$LENGTH or LENGTH

    The length in bytes of a segment

  Be sure to define a sufficiently large value for the RDMS$BIND_SEGMENTED_STRING_BUFFER logical name. An adequate buffer size is needed to store large segmented strings (using segmented string storage maps), in storage areas other than the default RDB$SYSTEM storage area. The minimum acceptable value for the RDMS$BIND_SEGMENTED_STRING_BUFFER logical name must be equal to the sum of the length of the segments of the segmented string. For example, if you know that the sum of the length of the segments is one megabyte, then 1,048,576 bytes is an acceptable value for this logical name.

STORE Statement with Segmented Strings

You must specify the logical name value because when the RDML precompiler stores segmented strings, Rdb/VMS does not know which table contains the string until after the entire string is stored. Rdb/VMS buffers the entire segmented string, if possible, and does not store it until the STORE statement executes.

If the segmented string remains buffered, it is stored in the appropriate storage area. If the string is not buffered (because it is larger than the defined value for the logical name or the default value of 10,000 bytes), it is stored in the default storage area and the following exception message is displayed:

```
%RDB-F-IMP_EXC, facility-specific limit exceeded
-RDMS-E-SEGSTR_AREA_INC, segmented string was stored incorrectly
```

To avoid this error, set the value of the RDMS$BIND_SEGMENTED_ STRING_BUFFER logical name to a sufficiently large value. Note that a value of up to 500 MB can be specified for this logical name. See the *VAX Rdb/VMS RDO and RMU Reference Manual* for more information on defining storage areas.

Examples

Example 1

The following programs demonstrate the use of the STORE statement to store segmented strings in a record. The programs:

- Declare an array to hold the segmented strings to be stored

- Assign values to the array

- Use a STORE operation to store the employee ID in the RESUMES relation

- Embed a second STORE operation in the first, in order to store the segmented strings in the same record in which the value for EMPLOYEE_ ID has been stored

- Store the values from the array into the RESUME field of the RESUMES relation

- Complete the STORE operation

- Retrieve the segmented strings (just stored) using a nested FOR statement

See Section 6.14 for more information on retrieving segmented strings.

STORE Statement with Segmented Strings

C Program

```c
#include <stdio.h>
DATABASE PERS = FILENAME "PERSONNEL";

main()
{
int line;
char *document[3];

document[0] = "first line of resume  ";
document[1] = "second line of resume ";
document[2] = "last line of resume   ";

READY PERS;
START_TRANSACTION READ_WRITE;

    STORE R IN RESUMES USING
        strcpy (R.EMPLOYEE_ID,"12345");
 for (line = 0; line <= 2; line++)
            STORE LINE IN R.RESUME
                strcpy(LINE.VALUE,document[line]);
                LINE.LENGTH = strlen(LINE.VALUE);
            END_STORE;
    END_STORE;

FOR R IN RESUMES WITH R.EMPLOYEE_ID = "12345"
    FOR LINE IN R.RESUME
      printf("%s\n",LINE.VALUE);
    END_FOR;
END_FOR;

COMMIT;
FINISH;
}
```

Pascal Program

```pascal
program segstr (input,output);
DATABASE PERS = FILENAME 'PERSONNEL';

type lines = varying [80] of char;
var linecnt  : integer;
 document : array [0..2] of lines;

begin

document[0] := 'first line of resume  ';
document[1] := 'second line of resume ';
document[2] := 'last line of resume   ';

READY PERS;
START_TRANSACTION READ_WRITE;
```

STORE Statement with Segmented Strings

```
STORE R IN RESUMES USING
   R.EMPLOYEE_ID:= '12345';
   for linecnt := 0 to 2 do
      STORE SEG IN R.RESUME
         SEG := document[linecnt];
         SEG.LENGTH := length(document[linecnt]);
      END_STORE;
END_STORE;

FOR R IN RESUMES WITH R.EMPLOYEE_ID = '12345'
   FOR SEG IN R.RESUME
      writeln (SEG);
   END_FOR;
END_FOR;

COMMIT;
FINISH;
end.
```

6.27 TRANSACTION_HANDLE Clause

A transaction handle is a host language variable that allows you to associate a name with a particular transaction. If you do not supply a handle name explicitly, RDML defines a default transaction handle for the transaction. You can use a transaction handle in the following RDML statements, clauses and functions:

- Boolean functions (ANY, UNIQUE)
- COMMIT
- DECLARE_STREAM
- FOR
- FOR SEGMENTED_STRINGS
- PREPARE
- ROLLBACK
- START_STREAM, Undeclared
- START_TRANSACTION
- Statistical functions (AVERAGE, COUNT, MIN, MAX, TOTAL)
- STORE
- STORE SEGMENTED_STRINGS

For the syntax diagram that shows the placement of the TRANSACTION_HANDLE in each of the preceding statements, see the section describing that statement.

Format

transaction-handle =

$$\longrightarrow (\longrightarrow \underline{TRANSACTION\_HANDLE} \longrightarrow host\text{-}var \longrightarrow) \longrightarrow$$

TRANSACTION_HANDLE Clause

Argument

host-var
A valid host language variable. See Usage Notes.

Usage Notes

A transaction handle must be:

- Declared in the host language program as:

    - Either [VOLATILE]INTEGER or RDML$HANDLE_TYPE for Pascal

    - Either Integer (int) or RDML$HANDLE_TYPE for C

- Initialized to zero (0) for C and Pascal

Note *Rdb/VMS allows each user only one active transaction per database. A user is permitted to have multiple active transactions as long as each transaction is either attached to a different database, or each transaction is a separate instance of an attach to the same database.*

Rdb/ELN allows each user to have multiple active transactions attached to the same database.

Examples

Example 1

The following programs demonstrate the use of a transaction handle. These programs declare the host language variable, emp_update. The programs use emp_update to qualify the transaction in the START_TRANSACTION statement, the record selection expression, and ROLLBACK (instead of the COMMIT statement). The record selection expression modifies the record with the specified identification number in the EMPLOYEES relation. The COMMIT statement, also qualified with the transaction handle, ensures that the modified record is stored in the database.

The C program uses the function pad_string to append trailing blanks and the null terminator to the LAST_NAME field. This ensures that the length of the last name matches the length defined for the field. For more information and the source code for pad_string, see Appendix B.

TRANSACTION_HANDLE Clause

C Program

```
#include <stdio.h>
DATABASE PERS = FILENAME "PERSONNEL";

extern void pad_string();

main()
{
int EMP_UPDATE = 0;

READY PERS;
START_TRANSACTION (TRANSACTION_HANDLE EMP_UPDATE) READ_WRITE;

FOR (TRANSACTION_HANDLE EMP_UPDATE) E IN EMPLOYEES
   WITH E.EMPLOYEE_ID = "00178"
      MODIFY E USING
         pad_string("Brannon", E.LAST_NAME, sizeof(E.LAST_NAME));
      END_MODIFY;
END_FOR;

ROLLBACK (TRANSACTION_HANDLE EMP_UPDATE);
FINISH;
}
```

Pascal Program

```
program trhand (input,output);
DATABASE PERS = FILENAME 'PERSONNEL';

var EMP_UPDATE : [volatile] integer := 0;

begin

READY PERS;
START_TRANSACTION (TRANSACTION_HANDLE EMP_UPDATE) READ_WRITE;

FOR (TRANSACTION_HANDLE EMP_UPDATE) E IN EMPLOYEES
   WITH E.EMPLOYEE_ID = '00178'
      MODIFY E USING
         E.LAST_NAME := 'Brannon';
      END_MODIFY;
END_FOR;

ROLLBACK (TRANSACTION_HANDLE EMP_UPDATE);
FINISH;
end.
```

TRANSACTION_HANDLE Clause

Example 2

The following programs demonstrate the use of a transaction handle with a ROLLBACK statement to undo changes to the database made with the STORE statement. The programs:

- Start a read/write transaction, SAL_INCREASE

- Store a new JOBS record using the SAL_INCREASE transaction

- Use the ROLLBACK statement to undo the changes made to the database during the SAL_INCREASE transaction; that is, the new record is not stored in the database

Note that the C program uses the function pad_string. This function ensures that the values stored in each field have the correct number of trailing blanks to match the text size of the field. For more information and the source code for pad_string, see Appendix B.

C Program

```
#include <stdio.h>
DATABASE PERS = FILENAME "PERSONNEL";

extern void pad_string();

main()
{
int SAL_INCREASE = 0;

READY PERS;
START_TRANSACTION (TRANSACTION_HANDLE SAL_INCREASE) READ_WRITE;

STORE (TRANSACTION_HANDLE SAL_INCREASE) J IN JOBS USING
   pad_string ("TYPS", J.JOB_CODE, sizeof(J.JOB_CODE));
   pad_string ("1", J.WAGE_CLASS, sizeof(J.WAGE_CLASS));
   pad_string ("TYPIST", J.JOB_TITLE, sizeof(J.JOB_TITLE));
   J.MINIMUM_SALARY = 10000;
   J.MAXIMUM_SALARY = 17000;
END_STORE;

ROLLBACK (TRANSACTION_HANDLE SAL_INCREASE);
FINISH;
}
```

TRANSACTION_HANDLE Clause

Pascal Program

```
program rollback_trans (input,output);
DATABASE PERS = FILENAME 'PERSONNEL';
var sal_increase : [volatile] integer := 0;

begin
READY PERS;
START_TRANSACTION (TRANSACTION_HANDLE SAL_INCREASE) READ_WRITE;

STORE (TRANSACTION_HANDLE SAL_INCREASE) J IN JOBS USING
    J.JOB_CODE := 'TYPS';
    J.WAGE_CLASS := '1';
    J.JOB_TITLE := 'Typist';
    J.MINIMUM_SALARY := 10000;
    J.MAXIMUM_SALARY := 17000;
END_STORE;

ROLLBACK (TRANSACTION_HANDLE SAL_INCREASE);
FINISH;
end.
```

A

RDML-Generated Data Types

The tables in this appendix list the VAX C, VAX Pascal, and VAXELN Pascal
data types that RDML generates for each data type permitted in an Rdb
database.

In some cases, the data type that RDML generates depends on the scale factor.
For example, the following entry is from the VAX C table:

Rdb Database Data Type	VAX C Data Type Generated by RDML
SIGNED WORD SCALE n	int (n=1,2,3,4) char [8] (n>4)

This table entry indicates that the value of n determines whether an "int"
or "char [8]" data type is defined for a database field whose type is SIGNED
WORD SCALE n. If n equals 1, 2, 3, or 4, RDML will declare that field as an
int. If n is greater than 4, RDML will declare that field as a char [8].

See Table A–1 for RDML-generated data types for VAX C.

Table A–1   RDML-Generated Data Types for VAX C

Rdb Database Data Type	VAX C Data Type Generated by RDML
SIGNED BYTE (no scale)	char
SIGNED BYTE SCALE n	short (n=1,2)
SIGNED BYTE SCALE n	int (n=3,4,5,6,7)
SIGNED BYTE SCALE n	char [8] (n>8)
SIGNED BYTE SCALE –n	double

(continued on next page)

Table A–1 (Cont.)   RDML-Generated Data Types for VAX C

Rdb Database Data Type	VAX C Data Type Generated by RDML
SIGNED WORD	short
SIGNED WORD SCALE n	int (n=1,2,3,4)
SIGNED WORD SCALE n	char [8] (n>4)
SIGNED WORD SCALE –n	float
SIGNED LONGWORD	int
SIGNED LONGWORD SCALE n	char [8]
SIGNED LONGWORD SCALE –n	double
SIGNED QUADWORD	char [8]
SIGNED QUADWORD SCALE n	char [8]
SIGNED QUADWORD SCALE –n	double
F_FLOATING	float
G_FLOATING	double
DATE	char [8]
TEXT n	char [n+1]
VARYING STRING n	unsupported
SEGMENTED STRING ID	char [8]

The VARYING STRING data type is not supported in C. However, you can still use VARYING STRINGS in RDML/C. See Section 6.13, Example 4, and Section 6.25, Example 3.

See Table A–2 for RDML-generated data types for VAX Pascal and Table A–3 for VAXELN Pascal.

Table A–2   RDML-Generated Data Types for VAX Pascal

Rdb Database Data Type	VAX Pascal Data Type Generated by RDML
SIGNED BYTE (no scale)	[BYTE] –128 . . . 127;
SIGNED BYTE SCALE n	WORD (n=1,2)
SIGNED BYTE SCALE n	INTEGER (n=3,4,5,6,7)
SIGNED BYTE SCALE n	RECORD L0:UNSIGNED;L1:INTEGER;END (n>8)

Table A–2 (Cont.)   RDML-Generated Data Types for VAX Pascal

Rdb Database Data Type	VAX Pascal Data Type Generated by RDML
SIGNED BYTE SCALE –n	DOUBLE (n<0)
SIGNED WORD	[WORD] –32768..32767
SIGNED WORD SCALE n	INTEGER (n=1,2,3,4) RECORD L0:UNSIGNED;L1:INTEGER;END (n>4)
SIGNED WORD SCALE –n	REAL
SIGNED LONGWORD	INTEGER
SIGNED LONGWORD SCALE n	RECORD L0:UNSIGNED;L1:INTEGER;END
SIGNED LONGWORD SCALE –n	DOUBLE
SIGNED QUADWORD	QUADWORD RECORD L0:UNSIGNED; L1:INTEGER;END
SIGNED QUADWORD SCALE n	RECORD L0:UNSIGNED;L1:INTEGER;END
SIGNED QUADWORD SCALE –n	DOUBLE
F_FLOATING	REAL
G_FLOATING	DOUBLE
DATE	[BYTE(8)] RECORD END
TEXT n	CHAR (n=1) PACKED ARRAY [1..n] OF CHAR (n>1)
VARYING STRING n	VARYING [n] OF CHAR
SEGMENTED STRING ID	RECORD L0:UNSIGNED;L1:INTEGER;END

Table A–3   RDML-Generated Data Types for VAXELN Pascal

Rdb Database Data Type	VAXELN Pascal Data Type Generated by RDML
SIGNED WORD	–32768..32767
SIGNED WORD SCALE n	INTEGER (n=1,2,3,4) LARGE_INTEGER (n>4)
SIGNED WORD SCALE –n	REAL
SIGNED LONGWORD	INTEGER
SIGNED LONGWORD SCALE n	LARGE_INTEGER
SIGNED LONGWORD SCALE –n	DOUBLE
SIGNED QUADWORD	LARGE_INTEGER

Table A–3 (Cont.)   RDML-Generated Data Types for VAXELN Pascal

Rdb Database Data Type	VAXELN Pascal Data Type Generated by RDML
SIGNED QUADWORD SCALE n	LARGE_INTEGER
SIGNED QUADWORD SCALE –n	DOUBLE
F_FLOATING	REAL
G_FLOATING	DOUBLE
DATE	LARGE_INTEGER
TEXT n	CHAR (n=1) STRING(n)
VARYING STRING n	VARYING_STRING(n)
SEGMENTED STRING ID	LARGE_INTEGER

B

VAX C Language Functions for I/O Operations

The VAX C functions described in this appendix are used to simplify the code in examples and to allow you to concentrate on the RDML statements rather than C input/output (I/O) issues. When you design your application programs you should carefully consider the I/O operations and determine the best method for handling these operations in your application. Most likely, the simple methods shown here are not sufficient. Read the *Guide to VAX C* for information on handling I/O tasks in C programs.

The C functions described in this appendix are:

- pad_string
- read_float
- read_int
- read_string

The source code for the functions appears at the end of this appendix.

Table B–1   Summary of VAX C Input/Output Functions

pad_string	Truncates or appends blanks to text strings. This function is used in the examples with STORE and MODIFY statements to ensure that the size of the string to be stored matches the size of the field into which it is being stored.
read_int	Prompts for integer input from the keyboard and stores this input in a C variable. This function also performs error testing for valid input.
read_float	Prompts for floating-point input from the keyboard and stores this input in a C variable. This function also performs error testing for valid input.
read_string	Prompts for text string input from the keyboard and stores this input in a C variable. This function also truncates or appends blanks to the input text strings as appropriate to fill out the field to the correct size. When used in conjunction with STORE and MODIFY statements, read_string ensures that the size of the string to be stored matches the size of the field into which it is being stored.

Usage Notes

To use these functions with the sample programs, you must:

1 Create a file that contains the functions listed in the following source code.

2 Name this file using the appropriate file extension (for example, C_FUNCTIONS.C).

3 Compile this file using the CC/G_FLOATING command.

4 Declare within your module those functions that you want to call.

5 Link the object file that contains these functions to the object file for the module in which you want to use them. For example:

```
LINK myfile.obj,c_functions.obj,options_file/OPT
```

Source Code:

```
/***********************************************************
 *                                                         *
 *  Code is provided for the following four functions.     *
 *                                                         *
 *  'pad_string' copies a null terminated string to a      *
 *  specified target, with space padding or truncation.    *
 *  It is used by the 'read_string' function.              *
 *                                                         *
 *  'read_int' reads decimal integer from standard input.  *
 *                                                         *
 *  'read_float' reads a floating-point (real) number from *
 *  standard input.                                        *
 *                                                         *
 *  'read_string' reads a string from standard input, and  *
 *  returns a padded or truncated result.                  *
 *                                                         *
 ***********************************************************/

#include <stdio.h>

#ifndef TRUE
#define TRUE    (1==1)
#define FALSE   (1!=1)
#endif

#ifndef EOS
#define EOS     '\0'
#endif
/***********************************************************
 *                                                         *
 *  pad_string (source, target, size)                      *
 *                                                         *
 *      Function to take a null terminated string (source) *
 *      and copy it to target, padding with spaces, or     *
 *      truncating, as appropriate, to the specified size. *
 *                                                         *
 *      Note that the resulting string will not be null-   *
 *      terminated.                                        *
 *                                                         *
 ***********************************************************/
void pad_string(source, target, size)
char *source,
     *target;
int  size;
{
char *sptr = source,                    /* Temporary source ptr */
     *tptr = target;                    /* Temporary target ptr */

    /* Copy no more than 'size' chars to the target string.    */
```

```
        while (size > 0)
            {
            if (!(*tptr = *sptr))
                break;
            sptr++;
            tptr++;
            size--;
            }

    /* Pad the end with spaces, if necessary. */

    while (size-- > 0)
        *tptr++ = ' ';
}
/***********************************************************
 *                                                         *
 * read_int(prompt)                                        *
 *                                                         *
 *      Function to read a decimal integer from stdin.     *
 *      Keeps prompting until valid integer is input.      *
 *                                                         *
 ***********************************************************/
int read_int(prompt)
char *prompt;
{
int len = 0;                    /* Temporary length   */
int i;                          /* Temporary integer  */
int matches;                    /* Match count        */

    if (prompt != NULL)
        len = strlen(prompt);   /* Extract the length once */

    while (TRUE)
        {
        /* If prompt specified, output it */

        if (len != 0)
            fputs(prompt, stdout);

        /* Get a decimal integer from stdin, converted   */
        /* (Note that any white space will terminate it) */

        matches = scanf("%d", &i);

        /* Flush extraneous input */

        while (getchar() != '\n')
            ;

        if (matches == 1)       /* If we matched on the scanf, we got one */
            break;

        /* Invalid, so print error message and do it again */

        fprintf(stderr, "Invalid input -- try again\n");
        }
    return i;
}
```

```
/**********************************************************
 *                                                        *
 * read_float(prompt)                                     *
 *                                                        *
 *      Function to read a floating-point (real) number   *
 *      from stdin.                                       *
 *      Keeps prompting until valid float is input.       *
 *                                                        *
 **********************************************************/
float read_float(prompt)
/* prompt is the phrase you want to output to the terminal to
   prompt the user for a real number. */
char *prompt;
{
int   len = 0;                   /* Temporary length     */
float f;                         /* Temporary float      */
int   matches;                   /* Match count          */

    if (prompt != NULL)
        len = strlen(prompt);    /* Extract the length once */

    while (TRUE)
        {
        /* If prompt specified, output it */

        if (len != 0)
            fputs(prompt, stdout);

        /* Get a real number from stdin, converted       */
        /* (Note that any white space will terminate it) */

        matches = scanf("%f", &f);

        /* Flush extraneous input */

        while (getchar() != '\n')
            ;

        if (matches == 1)        /* If we matched on the scanf, we got one */
            break;

        /* Invalid, so print error message and do it again */

        fprintf(stderr, "Invalid input -- try again\n");
        }
    return f;
}
```

```
/************************************************************
 *                                                          *
 * read_string(prompt, target, size)                        *
 *                                                          *
 *      Function to read a string from stdin.               *
 *      Truncates or pads the string to size.               *
 *                                                          *
 *      The returned string will not be null-terminated.    *
 *                                                          *
 *      If valid string input, returns 0.                   *
 *      If EOF, returns EOF.                                 *
 *                                                          *
 ************************************************************/
int read_string(prompt, target, size)
char *prompt;
char *target;
int size;
{
static char buffer[132];             /* Input buffer     */
int        return_value;             /* Value to be returned */

    /* If prompt specified, output it */

    if (prompt != NULL)
        if (strlen(prompt) != 0)
            fputs(prompt, stdout);  /* Output prompt without newline */

    /* Get an input line */

    if (gets(buffer) != NULL)        /* Get a line of input */
        {
        pad_string(buffer, target, size); /* Pad or truncate it */
        return_value = 0;                 /* Return success */
        }
    else
        return_value = EOF;

    return return_value;
}
```

Index

A

Absent values
  MISSING conditional expression,
    3–27
Access control
  START_TRANSACTION statement,
    6–128
Accessing multiple databases
  FINISH statement, 6–55
  using database handles, 6–21
  using the BASED ON clause, 6–4
  using the READY statement, 6–102
Accessing records
  database key, 2–26
  START_TRANSACTION statement,
    6–128
Access modes
  EXCLUSIVE, 6–134, 6–137
  PROTECTED, 6–134, 6–137
  SHARED, 6–134, 6–137
Adding a record to a relation
  STORE statement, 6–140
Addition
  arithmetic operator, 2–5
Advancing in a stream
  FETCH statement, 6–49
  FOR statement, 6–58
Aggregate expressions
  See Statistical functions

Alphabetic characters
  sort order of, 4–45
AND logical operator, 3–4
  described, 3–7e to 3–8e
ANY conditional expression
  described, 3–9, 3–10e, 3–11e to
    3–12e
  testing for presence of record, 3–9
Arithmetic expression, 2–6e to 2–8e
  order of evaluation, 2–5
Arithmetic operator, 2–4t
  addition, 2–5
  division, 2–5
  in value expression, 2–4
  multiplication, 2–5, 2–6
  subtraction, 2–5, 2–7
  unary operator, 2–5
Arithmetic value expressions, 2–2, 2–4
ASCII
  sorting order, 4–45
ASCTIM routine
  using to convert DATE data types,
    4–5, 4–11, 5–19
Assigning values to host language
    variable
  using the GET statement, 6–71, 6–72
Assignment operator
  in STORE statement with segmented
    strings, 6–152

R

S