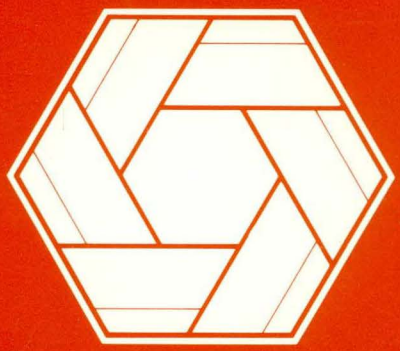
VAX Rdb/VMS

VAX Rdb/VMS
Guide to Data Manipulation

Order No. AA-N036C-TE



digital™ software

VAX Rdb/VMS
Guide to Data Manipulation

Order No. AA-N036C-TE

November 1987

This manual provides information about retrieving, modifying and erasing data in a database using the VAX Rdb/VMS interactive Relational Database Operator.

OPERATING SYSTEM:	**VMS**
	MicroVMS
SOFTWARE VERSION:	**VAX Rdb/VMS V2.3**

digital equipment corporation, maynard, massachusetts

The information in this document is subject to change without notice and should not be construed as a commitment by Digital Equipment Corporation. Digital Equipment Corporation assumes no responsibility for any errors that may appear in this document.

The software described in this document is furnished under a license and may be used or copied only in accordance with the terms of such license.

No responsibility is assumed for the use or reliability of software on equipment that is not supplied by DIGITAL or its affiliated companies.

The postage-paid READER'S COMMENTS form on the last page of this document requests your critical evaluation to assist us in preparing future documentation.

The following are trademarks of Digital Equipment Corporation:

ACMS	PDP	VAX
CDD	RALLY	VAXcluster
DATATRIEVE	Rdb/ELN	VAXinfo
DEC	Rdb/VMS	VAX Information Architecture
DECnet	ReGIS	VIDA
DECUS	TDMS	VMS
MicroVAX	TEAMDATA	VT
MicroVMS	UNIBUS	

digital™

Contents

Tables

How to Use This Manual

VAX Rdb/VMS software, referred to as Rdb/VMS in this manual, is a general purpose database management system based on the relational data model.

Purpose of This Manual

This manual introduces the syntax and semantics of Rdb/VMS data manipulation statements, which retrieve, store, and modify data in an Rdb/VMS database. This book demonstrates these statements using the Relational Database Operator (RDO) utility for Rdb/VMS, and provides examples using a demonstration database. You can use RDO for data definition, data manipulation, and database maintenance functions.

Intended Audience

To get the most out of this manual, you should be familiar with data processing procedures, basic database management concepts and terminology, and the VMS operating system.

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 in the Rdb/VMS media kit, in either the *VAX Rdb/VMS Installation Guide* or the *Before You Install* letter.

Contact your DIGITAL representative if you have questions about the compatibility of other software products with this version of Rdb/VMS. You can request the most recent copy of *VAX System Software Order Table/Optional Software Cross Reference Table*, SPD 28.98.XX, which will verify which versions of your operating system are compatible with this version of Rdb/VMS.

Structure

This manual is divided into seven chapters, two appendixes, and an index:

Chapter 1	Introduces the concepts of data organization and the relational database model. It demonstrates how to use RDO and describes how Rdb/VMS statements can be used in programs.
Chapter 2	Introduces Rdb/VMS data manipulation statements. It demonstrates how to access a database and explains how to use transactions.
Chapter 3	Describes RDO and record selection expressions (RSEs).
Chapter 4	Describes the process of retrieving data from one or more relations.
Chapter 5	Describes how to define views and how to use them in queries.
Chapter 6	Introduces the value expressions used for queries in RDO.
Chapter 7	Describes the Rdb/VMS data manipulation statements used to update databases.
Appendix A	Presents database definitions for the sample PERSONNEL database.
Appendix B	Shows how to use a text editor for program development.
Index	Provides page references for topics and commands covered in the manual.

Related Manuals

For more information on Rdb/VMS, see other manuals in this documentation set:

- *VAX Rdb/VMS Reference Manual*

  A complete description of the statements and syntax of Rdb/VMS

- *RDML Reference Manual*

  A complete description of the components of the Relational Data Manipulation Language (RDML)

- *VAX Rdb/VMS Guide to Programming*

  A complete description of how to write programs in high-level languages that use Rdb/VMS as a database access method

- *VAX Rdb/VMS Guide to Database Design and Definition*

  A tutorial that explains how to design a logical database and how to translate that design into a physical database using Rdb/VMS data definition statements

- *VAX Rdb/VMS Guide to Database Administration and Maintenance*

  A tutorial that provides guidelines for good performance and explains how to use the database maintenance utilities to perform such operations as backup, recovery, restoring journals, and analyzing the database

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.

This section explains the special symbols used in this book:

<CTRL/x>	This symbol tells you to press the CTRL (control) key and hold it down while pressing a letter key.
.	
.	
.	Vertical ellipsis in an example means that information not directly related to the example has been omitted.
Color	Color in examples shows user input.

References to Products

Rdb/VMS is a member of the VAX Information Architecture, a group of products that work with each other and with VAX languages conforming to the VAX calling standard to provide flexible solutions for information management problems.

VAX Information Architecture documentation explaining how these products interrelate is included with VAX Common Data Dictionary documentation. VAX Information Architecture documentation is also available separately. Contact your DIGITAL representative.

The Rdb/VMS documentation to which this document belongs often refers to these products by their abbreviated names:

- VAX BASIC is referred to as BASIC.
- VAX C is referred to as C.
- VAX Common Data Dictionary software is referred to as CDD.
- VAX COBOL software is referred to as COBOL.
- VAX DATATRIEVE software is referred to as DATATRIEVE.
- VAX EDT software is referred to as EDT.
- VAX Rdb/VMS software is referred to as Rdb/VMS.

Technical Changes and New Features

This section presents a list of the new Rdb/VMS Version 2.3 features that are covered in this manual. See the *VAX Rdb/VMS Release Notes* for information about all the new Version 2.3 features and for reports of current limitations or restrictions.

Version 2.3

Command Recall

Rdb/VMS now allows you to use the up and down arrow keys to recall RDO commands. You can recall up to the last 20 commands you issued while using RDO. This feature works just like the command recall feature at the DCL command level. See Chapter 1 for details.

BATCH_UPDATE No Longer Permits Rollback

BATCH_UPDATE transactions no longer allow you to issue an explicit ROLLBACK statement. Because BATCH_UPDATE works without a run-unit journal file (generally, to speed initial load operations), a ROLLBACK permanently corrupts the database file.

Instead of permitting the ROLLBACK, Rdb/VMS issues an RDB-E-NO_ROLLBACK error message and does not end the transaction. See Chapter 3 of this manual for more information.

Introduction to VAX Rdb/VMS Data Manipulation 1

This chapter introduces Rdb/VMS data manipulation concepts. It is divided into three sections:

- What Is a Relational Database (Section 1.1)

  Introduces the concepts that are the basis for relational database management systems.

- Using RDO (Section 1.2)

  Explains how to use the Relational Database Operator (RDO) utility, the interactive environment for Rdb/VMS.

- Using Rdb/VMS Statements in Programs (Section 1.3)

  Shows how to include RDO statements in high-level language programs. Programmers can read this section as an introduction to programming with Rdb/VMS.

The examples throughout this guide use the PERSONNEL database. You can create your own copy of the PERSONNEL database using the files provided in the Rdb/VMS installation kit. To do so, enter the following:

```
$ @RDM$DEMO:PERSONNEL
```

This procedure displays:

- Informational messages and prompts you for information about the Common Data Dictionary (CDD).
- A message reminding you to look in the file BUILDPERS.LOG to determine whether or not creation of the database was successful.
- Several messages while loading the database, including a display of statistics about the number of records loaded.

When the PERSONNEL database has been successfully created, you can use any of the examples presented in this manual.

1.1 What Is a Relational Database?

In a relational database, data resides in two-dimensional tables known as relations. A relation consists of rows and columns. The columns, which usually have names, divide each row into a set of fields. For a single field within a row, there is only one data item.

If you are familiar with VAX COBOL or VAX DATATRIEVE, you have probably used a COBOL File Description or a DATATRIEVE record definition. Imagine a record definition with no group fields or OCCURS clauses. Such a record definition is similar to a relation. In fact, the rows in an Rdb/VMS relation are called records, and the columns are called fields. An Rdb/VMS record, however, differs from a COBOL record in two ways:

- An Rdb/VMS relation cannot have repeating groups (lists). A maximum of one data item occupies a single named field in the record.
- An Rdb/VMS record cannot have group fields. A name within an Rdb/VMS relation refers to only one field.

Without repeating groups and group fields, the structure of the database is simplified so you may easily access each data item.

Figure 1-1 represents a typical Rdb/VMS relation that shows employee information. This relation is a subset of PERSONNEL, the sample database.

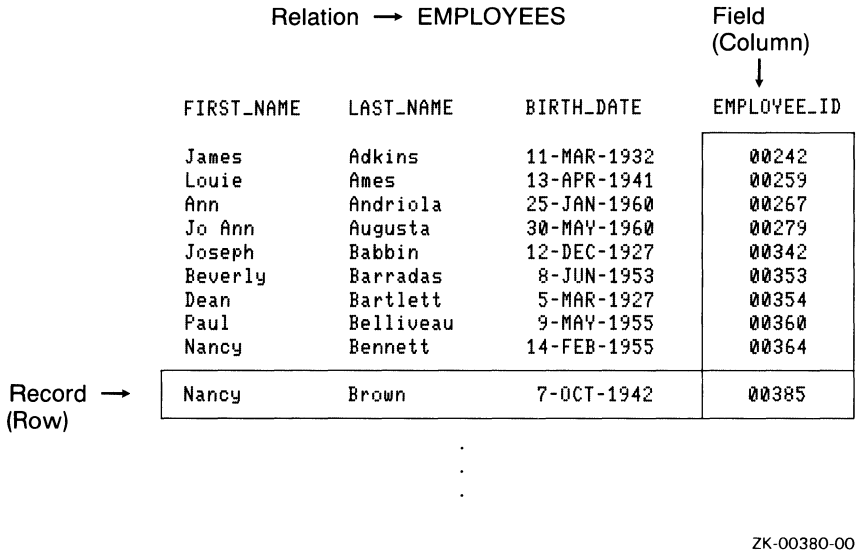


Figure 1-1: A Typical Rdb/VMS Relation in Table Form

In this relation, each field represents a particular item of data for each employee. Like relations, fields also have names. Each record represents the data on a single employee. To find the data stored in any location of the relation, you need only name the relation and specify the intersection of field and record.

For example, to find the employee ID for Nancy Brown, you need to specify the EMPLOYEES relation and the fields, LAST_NAME and FIRST_NAME.

Then Rdb/VMS:

- Finds the record in EMPLOYEES in which the LAST_NAME field is occupied by the name Brown and the FIRST_NAME field is occupied by the name Nancy
- Returns to the terminal the contents of the three fields named in the PRINT statement

The result of this query is:

```
Nancy          Brown          00385
```

1.1.1 Using Normalization to Eliminate Data Redundancy

There is no way to represent repeating groups in an Rdb/VMS relation; only one data item can occupy an intersection. Therefore, if you wanted to store information about five previous jobs for an employee, you would have to repeat the name, address, identification number, and other employee information five times. There would no longer be a one-to-one correspondence between records in the relation and employees in the company.

Storing all the information that might be relevant to employees in one relation means storing the same data in more than one place. Redundancy of data has two disadvantages:

- It wastes space in the database.
- It makes updating information difficult. For example, if you store the salary ranges for five previous jobs for an employee in the EMPLOYEES relation, you must find and change all the occurrences whenever the salary ranges change. If you miss some, the database is no longer consistent.

A process known as normalization solves these two problems. Normalization ensures that the database keeps separate concepts physically separate and eliminates data redundancy. Thus you store a data item only once, and you need to perform only one update operation to change it. When you need to bring data together from different relations (if you want an employee's job history, for instance), the database allows you to create temporary relationships by joining relations together. Rdb/VMS works best with well-designed, normalized databases.

1.2 Using RDO

You can define and access an Rdb/VMS database using RDO. When you run RDO and type statements at the RDO> prompt, Rdb/VMS executes the statements immediately. This section shows you how to start using RDO and gives a brief introduction to elements of the RDO utility.

1.2.1 Beginning an RDO Session

To invoke RDO, type at the DCL prompt:

```
$ RUN SYS$SYSTEM:RDO
```

RDO responds with the following prompt:

```
RDO>
```

Prompts help you keep track of your status during an interactive RDO session. The RDO prompts are:

RDO>	RDO command level prompt. This prompt tells you that you are typing commands to RDO and may enter any RDO statement.
cont>	The statement continuation prompt. This prompt indicates you have not yet entered a complete statement.

In addition to prompts, RDO incorporates many features to make working with Rdb/VMS easy. These features include:

- HELP

  Provides information about Rdb/VMS statements and concepts.

- SHOW

  Displays information about the database users, including the names and attributes of fields, the structure of relations, and the definitions of indexes and constraints. The SHOW command also displays information about the version of Rdb/VMS you are using.

- SET

  Determines the characteristics of the terminal session and the default directory in the Common Data Dictionary (CDD).

- Command recall

  The up and down arrow keys let you recall RDO commands. Recalls up to the last 20 commands you issued. This feature works just like the command recall feature at the DCL command level.

- Indirect command file

  Lets you store RDO statements and execute them later by using the at sign (@). The default file type is .RDO.

- A DCL invoke command

  Lets you access DIGITAL Control Language (DCL) commands from RDO by using the dollar sign ($). You do not have to leave your RDO session to answer mail or to see a directory.

- EDIT

  Calls the VAX EDT or VAXTPU editors. Type EDIT followed by an integer

to edit up to 20 of your previous commands. You also can use EDT or VAXTPU from inside RDO to insert successful RDO statements into command files and programs. Appendix B of this manual explains how to use RDO and text editors together to test and debug data manipulation statements.

- RDOINI.RDO

  A startup file that you create. When you enter RDO, the commands in your RDOINI.RDO file take effect. You may create RDOINI files in many directories, or define a logical name RDOINI to point to a central startup file.

1.2.2 Writing a Simple Record Selection Expression

The following is an example of a record selection expression:

```
FOR E IN EMPLOYEES WITH E.LAST_NAME = 'Brown' AND
  E.FIRST_NAME = 'Nancy'
    PRINT
      E.FIRST_NAME,
      E.LAST_NAME,
      E.EMPLOYEE_id
END_FOR
```

In this example, "WITH E.LAST_NAME = 'Brown' AND E.FIRST_NAME = 'Nancy'" is a **record selection expression** (RSE).

RDO statements use a group of records, called a **record stream**, that you retrieve from one or more relations in an Rdb/VMS database. You determine the records in the stream by including an RSE as part of your data manipulation statements. The clauses of the RSE determine which records from a relation are included in the stream. In this case, only records in which the employee's last name is Brown and the first name is Nancy are included in the record stream. You can include all the records of a relation, or you can restrict the record stream to a selected group of records.

Once you form the stream, you can enter statements to display, store, modify, or erase the data in the stream, one record at a time. To display the results of a record selection expression, use the PRINT statement. The PRINT statement uses values from the record stream that you identify in the RSE.

The character E in the expression E IN EMPLOYEES is a **context variable**. This variable lets you refer to the EMPLOYEES relation specifically in the record selection expressions and in the PRINT statement. Context variables are particularly important when you are working with more than one relation. If two relations have fields with the same name, a context variable helps you refer to each of them unambiguously.

1.2.3 Getting Online Help in RDO

If you need an explanation of any RDO statement or concept while using RDO, type HELP to see a list of available topics, or type HELP and the name of a topic:

```
RDO> HELP DEFINE FIELD
```

The help function contains several levels. For example, if you ask for help on DEFINE FIELD, you will see a brief description, an example, and a set of choices, including one called Format. Format shows the syntax of the DEFINE FIELD command.

Once you have located the relevant piece of information in the help files, you can exit from help by pressing RETURN until you come back to the RDO> prompt.

1.2.4 Using Multiline Statements in RDO

RDO can read enough lines in a multiline statement to detect a syntactically complete statement. If you end each line of an RSE with a keyword that belongs with the next line, RDO will wait for the entire sequence of statement lines before it executes them, as in the following example:

```
PRINT TOTAL SH.SALARY OF SH IN SALARY_HISTORY CROSS
 JH IN JOB_HISTORY OVER EMPLOYEE_ID WITH
 JH.JOB_CODE = "MENG" AND
 JH.JOB_END MISSING
```

You can also end each line of a multiline statement with the hyphen (-) continuation character to ensure that RDO reads the whole statement before execution. The continuation character must be the last character on the line to be continued. See the *VAX Rdb/VMS Reference Manual* for a full explanation of the type of input RDO accepts.

1.2.5 Exiting from RDO

You end an RDO session by entering one of the following:

- EXIT
- CTRL/Z

Entering either of these ends a session and normally returns you to the DCL prompt. For example:

```
RDO> EXIT
$
```

If you have changed the database without finishing the transaction, you cannot exit from RDO. If you try to exit, RDO responds with the following prompt:

```
RDO> EXIT
There are uncommitted changes to this database.
Would you like a chance to COMMIT these changes (No)?
```

If you type YES, RDO returns the RDO> prompt. Then you can type COMMIT, ROLLBACK, or any other RDO statement. For an explanation of COMMIT and ROLLBACK, see the section on ending a transaction in Chapter 2. If you type NO, you will leave the RDO session and return to the DCL prompt without saving any changes you may have made to the database.

Try the following statements to see how these features work. The text does not show the output:

```
RDO> HELP
RDO> HELP SET
RDO> HELP RDOINI
RDO> HELP DEFINE DATABASE
(Press return to leave the help facility)
RDO> $ DIRECTORY
RDO> $ MAIL
(Type EXIT to leave the mail facility)
RDO> EDIT *
```

1.3 Using Rdb/VMS Statements in Programs

As a programming tool, Rdb/VMS has several advantages:

- The versatility of the data manipulation statements means that the database system itself can perform many of the tasks you once needed to code in a high-level language.
- The interactive environment, RDO, lets you prototype your application before you start writing a program. With some modification, you can include the Rdb/VMS data manipulation statements in your programs.

Although the RSE you saw in the previous section was processed by interactive RDO, Rdb/VMS is intended to be used in programs.

Rdb/VMS includes a set of precompilers that let you include data manipulation statements in programs, as if they were part of the language. A precompiler translates the Rdb/VMS statements into subroutine calls and includes them in the compiled language code.

RDO provides an EDIT statement that makes developing these programs easy. Most often, you will use RDO to test queries and other data manipulation statements to make sure they produce the desired results. The EDIT command lets you modify a statement you previously entered in RDO.

Rdb/VMS saves that statement in an editing buffer, so you may use VMS text editors, such as VAX EDT and VAXTPU, to change any portion of the editing buffer. Use the EDIT statement to open the edit buffer. You may repeat this process as many times as you wish, editing up to 20 of your previous commands. When you have the desired results, save the query by issuing a WRITE or EXIT command. You may then incorporate the query into your high-level language program.

The following examples show how RDO statements can be used in a program.

The first example is a COBOL program that performs a store operation. This program reads the values for the database from the data file and stores them.

The second example is a COBOL program fragment that retrieves database values and assigns them to program variables. This example shows how to convert the RDO PRINT statement into a GET statement. The GET statement retrieves a value from the database and assigns it to a variable in the program. The GET statement uses the same kind of record selection expression as the PRINT statement. Note that the GET statement is specific to VAX BASIC. VAX COBOL and VAX FORTRAN programs use the RDBPRE precompiler. Languages supported by the Relational Data Manipulation Language (RDML) preprocessor (VAX C and VAX PASCAL) use a simple host language assignment statement instead.

Examples

Example 1

```
IDENTIFICATION DIVISION.
PROGRAM-ID.                 STORE-REC.
*
* First, identify the input data file.  The program will
* read this file and store its records in the database.
*
ENVIRONMENT DIVISION.
INPUT-OUTPUT SECTION.
FILE-CONTROL.

        SELECT EMPLOYEES-FILE  ASSIGN TO "EMP.DAT"
        ORGANIZATION IS SEQUENTIAL
        ACCESS MODE IS SEQUENTIAL.
*
* Instead of an explicit declaration, you can declare the
* record structure by copying a definition from the
* Common Data Dictionary. Then you declare the database
* file to the COBOL precompiler with the DATABASE
* statement.
*
DATA DIVISION.
FILE SECTION.
FD EMPLOYEES-FILE.

        COPY "CDD$TOP.FORESTER.PERSONNEL.RDB$RELATIONS.EMPLOYEES"
              FROM DICTONARY.
```

```
WORKING-STORAGE SECTION.

         &RDB&  INVOKE DATABASE FILENAME 'PERSONNEL'


PROCEDURE DIVISION.
*
* The PROCEDURE DIVISION consists simply of a database
* transaction in a loop.  Note the three steps:
*
* 1. Open the input file and start a transaction.
*    The RESERVING clause locks other users out of the
*    EMPLOYEES relation.
*
BEGIN.
        OPEN INPUT EMPLOYEES-FILE.
          &RDB&  START_TRANSACTION READ_WRITE
-          RESERVING EMPLOYEES FOR EXCLUSIVE WRITE.
*
* 2.  Read the file one record at a time and store
*     fields from the file into fields in the database relation:
*
READ-EMPLOYEES.
        READ EMPLOYEES-FILE AT END GO TO STORE-DONE.
         &RDB&  STORE E IN EMPLOYEES
-          USING
-            E.EMPLOYEE_ID = id;
-            E.LAST_NAME = LAST_NAME;
-            E.FIRST_NAME = FIRST_NAME;
-            E.MIDDLE_INITIAL = INITIAL;
-            E.ADDRESS = ADDRESS;
-            E.CITY = CITY;
-            E.STATE = STATE;
-            E.POSTAL_CODE = ZIP;
-       END_STORE
        GO TO READ-EMPLOYEES.
*
* 3.  Commit the transaction.  This makes the storage
*     operation complete.  The EXCLUSIVE lock on the
*     relation is released. The FINISH statement tells
*     Rdb/VMS that you are done working with the
*     database.
*
STORE-DONE.
         &RDB&  COMMIT
         &RDB&  FINISH
        CLOSE EMPLOYEES-FILE.
        STOP RUN.
```

Example 2

```
       DISPLAY "FIRST_NAME     LAST_NAME      id"

&RDB&  FOR E IN EMPLOYEES
      GET FIRST = E.FIRST_NAME;
          LAST = E.LAST_NAME;
          id = E.EMPLOYEE_id:
&RDB&  END_GET
       DISPLAY FIRST, "         ", LAST, "  ", id

&RDB&  END_FOR
```

Accessing a Database and Using Transactions 2

This chapter shows you how to use RDO to manipulate data. It describes how to use:

- The INVOKE DATABASE statement to tell RDO which database(s) you want to use
- The START_TRANSACTION statement to specify how and when transactions affect the database
- The COMMIT or ROLLBACK statement to end a transaction

2.1 Invoking a Database

Before you can access data managed by Rdb/VMS, you must name the database or databases you want to use. You use the INVOKE DATABASE statement, which identifies your process to the database(s) you name.

Rdb/VMS stores definitions of database elements in the database file itself and, optionally, in the VAX Common Data Dictionary (CDD), if it is installed. You can invoke the database by naming either its VMS file specification or its CDD path name. If you intend to retrieve or update the data itself, access the database by file name. If you intend to change data definitions, access the database by using the CDD path name. When you access the database using the CDD path name, any changes you make to database data definitions are entered in both the CDD and the physical database file.

2.1.1 Accessing the Database by File Name

Depending on where you set your current default directory, you can access an Rdb/VMS database by specifying either a partial or a full file specification. For more information, see the instructions on the next page.

- Supply a partial file specification if you use the DCL SET DEFAULT command to set your current default directory to the device and directory that contain the database. For example:

```
$ SET DEFAULT DISK1:[RDBDEMO.STAFF]
```

  Your INVOKE DATABASE statement needs only the file name:

```
RDO> INVOKE DATABASE FILENAME 'PERSONNEL'
```

- Supply the full file specification in the INVOKE DATABASE statement when your default directory is set to another directory:

```
RDO> INVOKE DATABASE FILENAME
cont>     'DISK1:[RDBDEMO.STAFF]PERSONNEL'
```

2.1.2 Accessing the Database by Path Name

Access an Rdb/VMS database by specifying a relative CDD path name or a full CDD path name. DCL and RDO provide commands to identify the correct directory in the CDD. For example:

- Supply a relative CDD path name if you use the DCL DEFINE command to identify the CDD directory where the database resides.

```
$ DEFINE CDD$DEFAULT 'CDD$TOP.RDB$DEMO.STAFF'
```

  The INVOKE DATABASE statement then requires only the relative path name of the database:

```
RDO> INVOKE DATABASE PATHNAME 'PERSONNEL'
```

- Supply a full CDD path name in the INVOKE DATABASE statement if your CDD$DEFAULT logical name is defined as another directory.

```
RDO> INVOKE DATABASE PATHNAME
cont>    'CDD$TOP.RDB$DEMO.STAFF.PERSONNEL'
```

You can also use the SET DICTIONARY command in RDO to change your CDD$DEFAULT directory. You need then supply only the relative path name of the database:

```
RDO> SET DICTIONARY CDD$TOP.RDB$DEMO.STAFF
RDO> INVOKE DATABASE PATHNAME 'PERSONNEL'
```

Note

Place quotation marks around file names and CDD path names to avoid ambiguity. The precompilers require either single or double quotes around file names and path names. See the *VAX Rdb/VMS Guide to Programming* for exact usage. RDO accepts either quoted or unquoted file specifications. However, if you do not use quotation marks, Rdb/VMS may not interpret file names or path names correctly.

2.2 Accessing the Database from a Remote Node

You can access an Rdb/VMS database from a remote node in a network using a full file specification in your INVOKE DATABASE statement. Assume you are logged on to node REM4 and the Rdb/VMS PERSONNEL database is located on the network node CENT in the DISK1:[COMPANY.STAFF] directory. The following INVOKE DATABASE statement gives you access to the PERSONNEL database:

```
RDO> INVOKE DATABASE FILENAME
cont>      'CENT::DISK1:[COMPANY.STAFF]PERSONNEL'
```

In the preceding example, CENT is the remote node name. By default, the Rdb/VMS supplied account, RDB$REMOTE, is used on the remote VAX node. For details on RDB$REMOTE, see Chapter 3 of the *VAX Rdb/VMS Installation Guide.*

To improve performance over the network, modify your login command file on the remote node to allow faster processing. For example, if you define logical names for your databases, do so at the beginning of LOGIN.COM. Then include a DCL command such as:

```
$  IF 'F$MODE()'  .EQS. "NETWORK" THEN $EXIT
```

2.2.1 Accessing Data

Once you have invoked a database, you can access the data in it. There are two types of files Rdb/VMS uses. Use the DIRECTORY command at the DCL level to look at the files created when you ran @RDM$DEMO:PERSONNEL:

```
$ DIRECTORY DISK1:[RDBDEMO.STAFF]PERSONNEL.*

PERSONNEL.RDB;1                PERSONNEL.SNP;1
```

The PERSONNEL.RDB file contains two types of information:

- Data values stored since its creation.
- Metadata that describes the fields, relations, and indexes as they are defined in the database. You can think of metadata as a set of templates that describe the format, structure, and characteristics of database elements. The field, relation, and index definitions from the PERSONNEL database you created with @RDM$DEMO:PERSONNEL are examples of metadata.

The PERSONNEL.SNP file is called a **snapshot** file. It contains copies of data that are used for READ_ONLY transactions. The section on using transactions explains the different transaction types, including the READ_ONLY transaction.

2.3 Using Transactions

Rdb/VMS allows many users access to a database at the same time, and it controls that access to avoid conflicts and data inconsistencies. Rdb/VMS, therefore, requires each user to identify a unit of database activity, called a transaction.

A **transaction** is an operation on the database that must complete as a unit or it will not complete at all. If, for example, you wanted to transfer an employee from one department to another, you would want the changes to all of that employee's records to be made at the same time. If a software error or hardware failure occurred before all operations in several transactions completed, the database might show that the employee belonged to two departments or had two salaries; thus the database would no longer be consistent. To avoid such inconsistencies, you include all such update tasks in a single transaction.

Transactions can have many characteristics, which you control with the START_TRANSACTION statement. A START_TRANSACTION statement signals the beginning of a transaction. The START_TRANSACTION statement options let you determine:

- Whether you want to work with the *snapshot* of the database or with the database itself
- Whether you intend to read or modify data in the relations
- What kind of access you allow other users to have to the database resources you are using
- When you want Rdb/VMS to consider specific conditions that must be satisfied before a record is stored or retrieved

In a START_TRANSACTION statement, you state:

- The transaction mode you need
  - READ_ONLY

    If your transaction only retrieves data values from the database, but does not change them
  - READ_WRITE

    If your transaction changes values in the database
  - BATCH_UPDATE

    For initial loads of new databases
- The names of the relations you want to access

  You can retrieve records from a single relation or from several relations joined together.
- The control you want over the access other users have to the relations you reserve for your transactions

When you access a specific record in a transaction, Rdb/VMS prevents other users from certain kinds of access to that record by *locking* the record. The kind of record locking specified in a START_TRANSACTION statement begins when you enter a query. The records identified by the record selection expression (RSE) remain locked until you terminate your transaction. The record locks are then released and other users may access those records. Refer to the section of the *VAX Rdb/VMS Guide to Database Administration and Maintenance* on performance considerations for a complete description of how Rdb/VMS uses the locking mechanism.

The following sections explain how to use the READ_ONLY and READ_WRITE transaction modes and their options, and how these affect database performance.

2.4 START_TRANSACTION Modes: READ_ONLY, READ_WRITE and BATCH_UPDATE

The START_TRANSACTION statement allows different types of access to relations in a database. You establish restrictions on other users' access and declare your work intentions by naming:

- The transaction mode you intend to use
- One or more relations you intend to use
- The type of control over the actions of other users and applications

Every statement you enter with RDO requires a specific transaction mode to determine the type of access to the database. Whether you plan to change data values, modify current database definitions, create new ones, or simply retrieve values from the database, the type of statement you enter determines how Rdb/VMS lets you do that task. If you do not enter a START_TRANSACTION statement to begin your database task, Rdb/VMS provides you with a default transaction mode depending on the first statement you issue in your interactive session.

Rdb/VMS considers all statements to be one of two types and assigns a default transaction mode to each, depending on its type. The two types of statements are:

- Data manipulation statements

  You use data manipulation statements to access data. If you do not specify a transaction mode, Rdb/VMS starts a READ_ONLY transaction. For example, if your first query after the DATABASE statement displays data from the database, Rdb/VMS allows this task to execute. If, however, the first statement modifies or updates data, Rdb/VMS returns an error because a READ_WRITE transaction is required.

- Data definition language statements

  You use data definition statements to define, change, or delete database entity definitions. For example, if you need to change the data type of a field, or to define a new relation, you need update access to the database to make these changes. By default, Rdb/VMS provides a READ_WRITE transaction after you issue a data definition statement.

Relying on the default transaction mode can present a risk and confusion in multiple database access. While testing some queries on database entity definitions for errors, you may decide to roll back some data definitions or manipulation statements and make others permanent. Unless you provide a specific transaction mode, Rdb/VMS considers all statements between the START_TRANSACTION

statement and COMMIT or ROLLBACK statements as one transaction. Even if you do not enter an explicit START_TRANSACTION statement, Rdb/VMS begins one for you and you must end this transaction with a COMMIT or ROLLBACK statement before you can begin another task. If you enter a COMMIT statement, you make all changes to the database permanent. If you enter a ROLLBACK statement, the database returns to its pre-transaction state and any uncommitted changes are lost.

Even though Rdb/VMS supplies defaults, you should always use the START_TRANSACTION statement to declare a specific transaction mode. Figure 2-1 shows three transaction modes.

RDO> START_TRANSACTION <transaction mode>

READ_ONLY	All relations are available for data retrieval. Data values are those at the moment you entered your START_TRANSACTION statement. Updates made by other users while READ_ONLY is in effect are not visible by you. You may read any record in any relation to which you have authorized access via Rdb/VMS access rights.
READ_WRITE	Rdb/VMS reserves each relation as you refer to it. You may update any record in any relation to which you have authorized access via Rdb/VMS access rights.
BATCH_UPDATE	Reduces overhead in large load operations. To speed update operations, Rdb/VMS does not write any journal files in BATCH_UPDATE. Therefore, you cannot roll back a batch update transaction; if the load fails, you must recreate the database. When you specify the BATCH_UPDATE mode in your START_TRANSACTION statement, the load or update task results in access to the entire database. It is the most efficient mode you can choose when loading the entire database for the first time, or when you batch database updates in a data file that you intend to apply to the database at one time.

ZK-00385-00

Figure 2-1: Transaction Modes of a START_TRANSACTION Statement

2.4.1 READ_ONLY Transactions

When you are updating values, you change them in the database file itself (.RDB). If you only want to read values, however, you can read them from the snapshot file (.SNP). This type of access is called a READ_ONLY transaction. When you use a READ_ONLY transaction, you read current versions of records not locked by any other user and previous versions of records that are locked. Because many transactions can share read locks that Rdb/VMS places on records in the snapshot file, your transaction does not conflict with others.

In most cases, the previous version, sometimes referred to as a **before-image** of the record, is adequate. When the update transaction finishes and you begin a new transaction, the updated version of the record is available to you. However, if your READ_ONLY transaction requires an absolutely current picture of the database, you can reserve the relations that you need to read in the database itself, rather than accessing the snapshot version. Start a READ_WRITE transaction, and use the EXCLUSIVE share mode to deny other users access to the relations that you need to read. In this way, you can ensure that no records can change during your data retrieval. The next section contains more information on READ_WRITE transactions.

Specify READ_ONLY in your START_TRANSACTION statement when:

- You do not intend to add new records or to change existing values in the database.
- You do not require absolute accuracy.
- You need to create a report. Generating a report, however, is a more extensive task than browsing for individual records. See the next section on READ_WRITE transactions for additional information.

2.4.2 READ_WRITE Transactions

Specify READ_WRITE in your START_TRANSACTION statement when you want to perform read-only tasks together with additions, deletions, or changes to the database that use the STORE, ERASE, or MODIFY statements. When you need READ_WRITE access to the database, you can use several formats.

One format of the START_TRANSACTION statement requires only the READ_WRITE transaction to allow update operations in the database:

```
RDO> START_TRANSACTION READ_WRITE
```

This format does not name a specific relation or relations for database updates. Rdb/VMS reserves the relations as you name them in your statements and, depending on the type of operation you perform in your transaction, places write locks on selected records to complete an update task.

For example, the first data manipulation statements in your transaction might retrieve data from the EMPLOYEES relation. Rdb/VMS locks the records necessary to make an update. Later in the transaction you might modify values in selected records in the EMPLOYEES relation. Rdb/VMS, using only the necessary locks to complete the transaction, would promote the level of record locking.

You can enter a START_TRANSACTION statement that names the relations you need to read and that specifies what you will allow other users to do when they access the same relations. To get the higher locking you need for certain read operations, specify READ_WRITE with the correct share mode for your transaction. See the section on reserving options for information.

2.4.3 BATCH_UPDATE Transactions

You can reduce overhead in large load operations by using the BATCH_UPDATE mode. To speed update operations, Rdb/VMS does not write to any journal files in BATCH_UPDATE mode. Therefore, you cannot roll back a batch update transaction; if the load fails, you must recreate the database. For example, if you need a large test database for development purposes, BATCH_UPDATE mode loads the database, but bypasses the journaling facilities.

When you can specify the BATCH_UPDATE mode in your START_TRANSACTION statement, the load or update task results in access to the entire database. The BATCH_UPDATE mode, for example, requires your transaction to be the *only* transaction accessing the database. It is the most efficient mode you can choose when loading the entire database for the first time, or when you batch database updates in a data file that you intend to apply to the database at one time.

In addition to requiring the fewest locks on the database, the BATCH_UPDATE mode permits updates to the database without creating a run-unit journal file. Therefore, any records changed or added during the BATCH_UPDATE transaction cannot be rolled back because Rdb/VMS does not maintain before-images of the changed records.

Before you begin a BATCH_UPDATE transaction in your programs, you should create a backup copy of the database using the VMS Backup utility.

When you start a BATCH_UPDATE transaction, you cannot roll it back; that is, issuing a ROLLBACK statement will generate an error message and your transaction will not be rolled back. The following is a sample session that shows the effects of issuing a ROLLBACK statement:

First, the user backs up the database:

```
$BACKUP/LOG PERSONNEL.* PERSBACKUP.BCK/SAVE_SET
%BACKUP-S-COPIED, copied DISK1:[CORP.DBS]PERSONNEL.RDB;1
%BACKUP-S-COPIED, copied DISK1:[CORP.DBS]PERSONNEL.SNP;1
```

Then, the user invokes RDO:

```
$RDO
RDO> INVOKE DATABASE FILENAME PERSONNEL
RDO> START_TRANSACTION BATCH_UPDATE
RDO> STORE E IN EMPLOYEES USING
cont> E.EMPLOYEE_ID = "15399";
cont> E.LAST_NAME = "NORTH";
cont> E.FIRST_NAME = "OSCAR";
cont> END_STORE
RDO> ! At this point, assume the user doesn't know that
RDO> ! rolling back a BATCH_UPDATE transaction will corrupt the
RDO> ! database.  This user now enters a rollback:

RDO> ROLLBACK
%RDB-E-NOROLLBACK, no rollback is allowed with the recovery mechanism disabl
```

At this point, the user's BATCH_UPDATE transaction is still active:

```
RDO> SHOW TRANSACTION
All Transactions in Database with filename  PERSONNEL
a read-write transaction is in progress
  - updates have been performed
  - transaction sequence number (TSN) is 152
  - snapshot space for TSNs less than 152 can be reclaimed
  - session ID number is 55
```

If you receive the RDB$NO_ROLLBACK error during a BATCH_UPDATE transaction, you have two choices:

1. Manually undo any changes you made (or fix the problem you were having) and then commit the transaction. For example, if you stored a record, erase that record and then issue a COMMIT statement:

   ```
   RDO> FOR E IN EMPLOYEES WITH E.EMPLOYEE_ID = "15231"
   cont> ERASE E
   cont> END_FOR
   RDO> COMMIT
   RDO> FINI
   RDO> EXIT
   ```

   Or, if invalid input caused a constraint to fail, enter the correct, valid data an then issue a COMMIT statement.

2. Exit the program or RDO session, which will corrupt the database.

The second option assumes that you made a backup copy of the database prior to starting the BATCH_UPDATE transaction. After restoring the database files from the .BCK backup file, you can correct the situation that led to the error and then reenter the update program or RDO session.

2.4.4 Reserving Options

Another format of the START_TRANSACTION statement lets you name more than one relation in the same START_TRANSACTION statement for different database tasks. However, if you use a single transaction to modify fields in more than one relation, you may encounter unpredictable results. See the chapter on updating databases for more information. Every lock Rdb/VMS places on a database, relation, page, or index node reduces the lock resources available to other processes on your particular system. By using more than one relation in the same START_TRANSACTION statement, you lock only those database resources necessary to complete each task. You can do this by using the RESERVING clause.

For example, you can name one relation, EMPLOYEES, from which you intend only to retrieve data, while naming other relations, COLLEGES and DEGREES, for update activities:

```
RDO> START_TRANSACTION READ_WRITE RESERVING
cont>             EMPLOYEES FOR SHARED READ,
cont>             COLLEGES FOR SHARED WRITE,
cont>             DEGREES FOR EXCLUSIVE WRITE
```

You can specify the following share modes and lock types in the RESERVING clause of your START_TRANSACTION statement for update transactions:

- SHARED READ
- SHARED WRITE
- PROTECTED READ
- PROTECTED WRITE
- EXCLUSIVE READ
- EXCLUSIVE WRITE

Some database operations in a READ_WRITE transaction may require a higher level of record locking than the shared level. In such cases, Rdb/VMS automatically promotes locking to PROTECTED READ or PROTECTED WRITE to complete the task. Although the level of locking may often be higher than that which you specified, it is never lower than the level specified in the START_TRANSACTION statement.

You can use the different modes with the multiple database access feature. For example, within a single transaction, you can access one database in READ_WRITE ... RESERVING <rel-list> PROTECTED WRITE mode and a second database in READ_WRITE ... RESERVING <rel-list> SHARED READ mode.

```
RDO> INVOKE DATABASE DB1 = FILENAME 'PERSONAL$DISK:PERSONNEL'
RDO> INVOKE DATABASE DB2 = FILENAME 'PERSONAL$DISK:BENEFITS'
RDO> START_TRANSACTION ON DB1 USING
cont> (READ_WRITE RESERVING EMPLOYEES FOR PROT WRITE) AND
cont> ON DB2 USING (READ_WRITE RESERVING JOB_INFO FOR SHARED READ)
RDO>
```

The following sections discuss these reserving options.

2.4.4.1 SHARED READ Reserving Option -- The SHARED READ option lets other users' transactions retrieve records from the same relation you have accessed. It also allows transactions to update records within the same relation, except if those transactions are in an EXCLUSIVE share mode, which is a special mode described below. However, as you retrieve individual records from the relation, those individual records become unavailable for update by other users until you terminate your transaction. You can specify more than one relation for update within the same START_TRANSACTION statement.

```
RDO> START_TRANSACTION READ_WRITE RESERVING
cont>           EMPLOYEES FOR SHARED READ,
cont>           COLLEGES FOR SHARED WRITE
```

2.4.4.2 SHARED WRITE Reserving Option -- The SHARED WRITE option lets other transactions retrieve or update records in the same relation you have accessed, but not the particular records you have locked. Updated versions of records from other transactions are not available to you during your current transaction. Both your transaction and the updating transactions must terminate with either a COMMIT or a ROLLBACK statement. Also, any updated versions of the records you change are not available to other users until you terminate your update transaction with a COMMIT or ROLLBACK statement and other users begin new transactions.

Because many users can access the same relation, many records may be locked. Such record locking can result in access conflicts that can affect the performance and the level of concurrent access to database resources. Only one transaction can update a record at one time. If another user has locked a record for update or has placed a write lock on a record for retrieval, you cannot access that record for update until the record is released by the locking transaction.

Transactions that access the snapshot file do not intend to update the database. Therefore, READ_ONLY transactions can access records in the database with no conflict.

```
RDO> START_TRANSACTION READ_WRITE RESERVING
cont>          EMPLOYEES FOR SHARED WRITE
```

2.4.4.3 PROTECTED READ Reserving Option -- The PROTECTED READ option lets you read records from the same relation that other transactions are accessing. However, PROTECTED READ ensures that no other users can write to the relation that your transaction reserves in this manner. For example, assume you retrieve a record to generate a report. Rdb/VMS must keep the record stable while the read operation is performed. PROTECTED READ, unlike SHARED READ, prevents other users from changing any records included in the report until the report is finished.

```
RDO> START_TRANSACTION READ_WRITE RESERVING
cont>          EMPLOYEES FOR PROTECTED READ
```

2.4.4.4 PROTECTED WRITE Reserving Option -- The PROTECTED WRITE option lets your transaction update the relation but prevents other transactions from updating that relation. Other users can only retrieve data from the relation. Therefore, a transaction with extensive updates may execute faster using PROTECTED WRITE rather than using SHARED WRITE because Rdb/VMS does not have to check for as many conflicting locks as in the SHARED WRITE transaction. Wherever possible, use indexed fields in your RSE to lock only those database resources required by your transaction. Otherwise, Rdb/VMS will lock the entire relation.

```
RDO> START_TRANSACTION READ_WRITE RESERVING
cont>          EMPLOYEES FOR PROTECTED WRITE
```

2.4.4.5 EXCLUSIVE READ Reserving Option -- The EXCLUSIVE READ option allows only your transaction access to the specified relation. Other users cannot read or update this relation. This reserving option uses the fewest locks because Rdb/VMS locks the resource at the relation level. Because there is no conflict with other users, the EXCLUSIVE share mode retrieves data faster than SHARED or PROTECTED share modes. Use the EXCLUSIVE share mode if your read transaction requires an absolutely current picture of the database for retrieval. Because the EXCLUSIVE READ option denies other users access to the relation, you can ensure that no records can change during your data retrieval.

```
RDO> START_TRANSACTION READ_WRITE RESERVING
cont>          EMPLOYEES FOR EXCLUSIVE READ
```

2.4.4.6 EXCLUSIVE WRITE Reserving Option -- The EXCLUSIVE WRITE option lets only your transaction have access to the relation to read or update a record. This reserving option prevents all other transactions from reading or updating any record in the relation. Include the EXCLUSIVE WRITE option in your START_TRANSACTION statement when you are doing updates to one or more relations and the transaction is fairly short.

When your transaction includes a MODIFY or ERASE statement, Rdb/VMS checks to see if another user has a lock on the record or records you need. If the record has no lock, Rdb/VMS locks it by putting an EXCLUSIVE share mode on the record and executes the update statement. Your transaction holds the lock on this record until you commit the change to the database with COMMIT or undo the change with ROLLBACK.

```
RDO> START_TRANSACTION READ_WRITE RESERVING
cont>         EMPLOYEES FOR EXCLUSIVE WRITE
```

If you omit the explicit reserving options, Rdb/VMS assumes the defaults.

Figure 2-2 shows the START_TRANSACTION RESERVING syntax and explains the share modes and lock types that make up the reserving option.

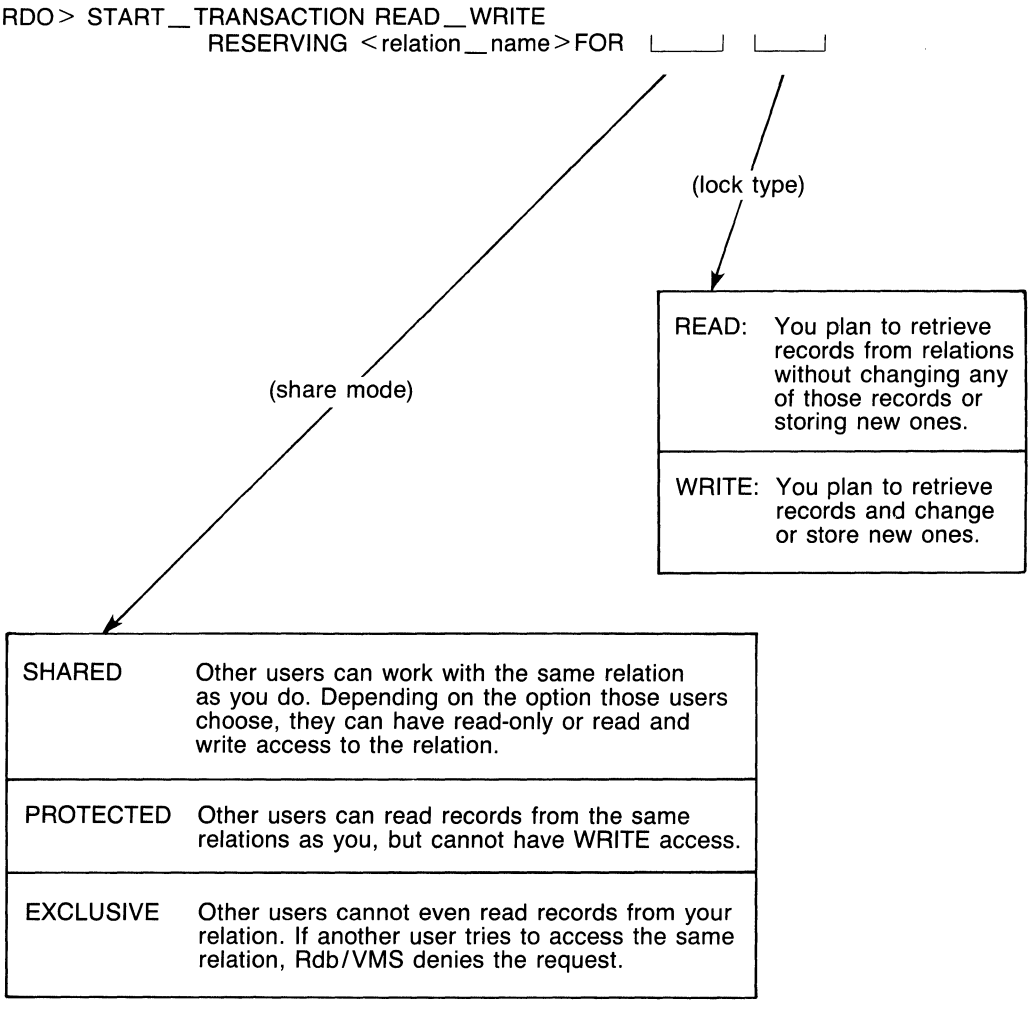


Figure 2-2: Share Modes and Lock Types for the START_TRANSACTION Statement

The reserving options you can choose in your START_TRANSACTION statement depend on the options other users currently accessing the database have already specified. Your options are more limited during multi-user database access than when you are the only user of the database.

Once Rdb/VMS grants the reserving option(s) specified in your START_TRANSACTION statement, it locks records identified by the record selection expression (RSE) according to the kind of task, or verb, your transaction executes. For example, when you retrieve a record to display the values for certain fields, Rdb/VMS places read locks on them. However, when you issue a MODIFY statement, Rdb/VMS places a more restrictive write lock on the record, or records, so that no other transaction may intervene and change the values you intend to change.

When you lock a record for a read or write operation, you affect other users. Table 2-1 shows that other users must either wait for record locks to be released (when WAIT is specified) or must terminate the active transactions and begin again.

Table 2-1: Database Access Conflicts for Relations

		SECOND USER						
		READ_ONLY (Snapshot)	READ_WRITE SHARED READ	READ_WRITE PROTECTED READ	READ_WRITE EXCLUSIVE READ	READ_WRITE SHARED WRITE	READ_WRITE PROTECTED WRITE	READ_WRITE EXCLUSIVE WRITE
FIRST USER	READ_WRITE EXCLUSIVE READ	Conflict	Conflict	Conflict	Conflict	Conflict	Conflict	Conflict
	READ_WRITE SHARED WRITE See Note 2	No conflict	No conflict	No conflict	Conflict	No conflict See Note 1	Conflict	Conflict
	READ_WRITE PROTECTED WRITE	No conflict	No conflict	Conflict	Conflict	Conflict	Conflict	Conflict
	READ_WRITE EXCLUSIVE WRITE See Note 3	Conflict	Conflict	Conflict	Conflict	Conflict	Conflict	Conflict

ZK-00376-00

```
Note 1:  If index is used.  If index is not used, may lock entire relation.
Note 2:  Updates are written to .SNP file.
Note 3:  Updates are not written to .SNP file.
```

Note

Because Rdb/VMS uses adjustable locking granularity, records may become locked. See the *VAX Rdb/VMS Guide to Database Administration and Maintenance* for information.

Note that BATCH_UPDATE works just like EXCLUSIVE UPDATE. However, because there is no .RUJ file you must be careful not to corrupt the database by issuing a ROLLBACK. BATCH_UPDATE is most useful for the initial loading of the database.

In all update cases, Rdb/VMS does not allow other transactions to read changed records until the updating transaction executes a COMMIT or a ROLLBACK. Because Rdb/VMS locks your records against access by other users, you can display the changes you have made to those records. This record locking assures the consistency and integrity of database records.

2.4.5 Other START_TRANSACTION Options

In addition to the reserving options of the START_TRANSACTION statement described in the previous section, you can specify other constraints or qualifiers that affect how Rdb/VMS handles your transactions. These options are:

- Constraints
  - EVALUATING AT VERB_TIME
  - EVALUATING AT COMMIT_TIME
- Qualifiers
  - WAIT
  - NOWAIT
  - CONSISTENCY
  - CONCURRENCY

The following sections discuss these options.

2.4.5.1 EVALUATING AT VERB_TIME Constraint -- You can define constraints to check for values you want to store in the database. You can then specify when Rdb/VMS should evaluate the constraints. By specifying VERB_TIME, you indicate that Rdb/VMS should evaluate the constraint when the STORE statement executes. By default, Rdb/VMS evaluates each user-defined constraint at the time specified in its DEFINE CONSTRAINT definition. If the constraint definition does not specify when the constraint should be checked, the definition default is CHECK ON UPDATE.

Each time Rdb/VMS executes a STORE or MODIFY statement, the record stream your RSE identifies may contain one record or many records. When the record stream contains only one record, and an error occurs, you can handle that error by displaying the offending record or writing it to an exception file. On the other hand, if the record stream identifies more than one record in a FOR ... END_FOR block containing a STORE or MODIFY statement, and an error occurs, you want to be sure which record in the record stream has violated the constraint definition. So, for each execution of the STORE or MODIFY statement in the FOR ... END_FOR block, you can specify that Rdb/VMS check the constraint by including the EVALUATING AT VERB_TIME clause.

Additionally, when you include update tasks in a host language program, you can handle errors with the ON ERROR clause. Specifying constraint evaluation at VERB_TIME causes control to pass immediately to the error handling statements. If your transaction waits until COMMIT_TIME to evaluate the constraint, Rdb/VMS may not signal the error at the verb level because the STORE or MODIFY statement will have completed. Refer to the *VAX Rdb/VMS Guide to Programming* for more details on error trapping and error handling using constraints.

If your transactions contain several update operations using both STORE and MODIFY statements, you may need to evaluate constraints at VERB_TIME to detect which operation or constraint caused the violation. Evaluating constraints at COMMIT_TIME may direct the entire transaction to roll back if error handling is not included at the verb level.

2.4.5.2 EVALUATING AT COMMIT_TIME Constraint -- If you have exclusive access to all referenced relations, you can evaluate constraints at COMMIT_TIME. You can defer constraint evaluation until you are ready to terminate your transaction. Rdb/VMS then checks each value against the defined constraint before allowing the record to be stored. When you specify constraints to be evaluated at COMMIT_TIME, you defer the expense of evaluation time until you enter the COMMIT statement. For example, assume you need to modify most of the records in a specific relation. You can specify EXCLUSIVE WRITE to avoid access conflicts with other users, reduce the use of lock resources, and complete your task efficiently. Because your transaction works with most of the

records in the relation, you do not need to evaluate one or more constraints every time a record in the record stream is updated. You can defer the cost of constraint evaluation until the transaction terminates. If the update transaction results in many constraint violations, you can roll back the transaction, correct the erroneous values, and retry the update operation.

If your tasks include batch updates to the database in a scheduled production mode, there may be very little conflict with other users accessing the database. In such cases, you can experiment with both VERB_TIME and COMMIT_TIME constraints to see which meets your needs. You can enhance performance by ensuring that fields used in the constraint definition are indexed fields. Indexes allow Rdb/VMS to locate specific records efficiently. See the *VAX Rdb/VMS Guide to Database Administration and Maintenance* for information on enhanced performance with index fields.

The following START_TRANSACTION statement specifies a constraint in the PERSONNEL database definitions called SH_EMP_ID_EXISTS to be evaluated for any new data stored in the database:

```
DEFINE CONSTRAINT SH_EMP_ID_EXISTS
   FOR SH IN SALARY_HISTORY
   REQUIRE ANY E IN EMPLOYEES WITH
      E.EMPLOYEE_ID = SH.EMPLOYEE_ID
   CHECK ON UPDATE.
```

The following constraint verifies that an EMPLOYEE_ID value exists in the EMPLOYEES relation before a SALARY_HISTORY record can be stored. The EVALUATING clause in the START_TRANSACTION statement overrides the CHECK ON UPDATE clause.

```
RDO> START_TRANSACTION READ_WRITE RESERVING
cont>          EMPLOYEES FOR EXCLUSIVE WRITE,
cont>          SALARY_HISTORY FOR EXCLUSIVE WRITE EVALUATING
cont>          SH_EMP_ID_EXISTS AT COMMIT_TIME
```

2.4.5.3 WAIT or NOWAIT Qualifiers -- You can specify how Rdb/VMS is to handle your transactions when you attempt to retrieve or update a record in a relation locked by another user. You can elect to wait for the record to be released by specifying WAIT in your START_TRANSACTION statement, or you can specify that Rdb/VMS return an error message that the record is unavailable. You can then terminate the current transaction and reenter your START_TRANSACTION statement or access another database.

```
RDO> START_TRANSACTION READ_WRITE RESERVING
cont>          EMPLOYEES FOR PROTECTED WRITE,
cont>          JOB_HISTORY FOR PROTECTED WRITE,
cont>          SALARY_HISTORY FOR SHARED READ NOWAIT
```

If you decide to include the WAIT clause in your START_TRANSACTION statement, you should consider the possibility of encountering incompatible transaction modes.

In the following series of steps, User A starts a transaction and reserves a relation for EXCLUSIVE WRITE. When Rdb/VMS grants a transaction EXCLUSIVE WRITE, other transactions started after the EXCLUSIVE WRITE transaction cannot gain snapshot access to the database resources held by the exclusive lock until the exclusive transaction is terminated. User A must first terminate the EXCLUSIVE WRITE transaction. User B's snapshot request, therefore, is not compatible with an EXCLUSIVE WRITE lock. Rdb/VMS immediately suspends User B from waiting.

1. User A reserves a relation for EXCLUSIVE WRITE, and fetches a collection of records.
2. User B then accesses the same relation for READ_ONLY and includes the WAIT clause.
3. When User B attempts to access the records in a FOR ... END_FOR block to display certain values, Rdb/VMS returns a resource lock NOWAIT error, and disregards the WAIT.
4. Although User A could terminate the current transaction, release all locks, and exit RDO and User B could begin the FOR ... END_FOR block again within the same transaction, Rdb/VMS will still return the resource lock error.

In this case, a NOWAIT error results and Rdb/VMS rejects the WAIT lock requested by User B because such a WAIT state can never be resolved for a READ_ONLY transaction against an EXCLUSIVE WRITE transaction. Therefore, Rdb/VMS overrides the WAIT qualifier and issues a NOWAIT rather than allow the READ_ONLY request to wait forever.

A transaction that specifies EXCLUSIVE WRITE does not write data to the snapshot file; it is *always* incompatible with READ_ONLY access requests. Rdb/VMS defines READ_ONLY transactions in such a way that all data committed to the database prior to its execution must be available to the transaction requesting access to the snapshot file. Because it is impossible for Rdb/VMS to determine whether the EXCLUSIVE WRITE transaction may have written data to the database, it cannot satisfy the READ_ONLY transaction's requirements.

2.4.5.4 CONSISTENCY or CONCURRENCY Qualifiers -- The CONSISTENCY and CONCURRENCY qualifiers are provided for compatibility with other DIGITAL relational database products, such as VAX Rdb/ELN. In Rdb/VMS alone, the distinction is not meaningful. Rdb/VMS always guarantees degree 3

consistency. Degree 3 consistency means that the database system guarantees that data you have read will not be changed by another user before you issue a COMMIT statement.

In other relational systems that you might access using the remote feature of Rdb/VMS, this option specifies the degree to which you want to control the consistency of the database. In such systems, the CONCURRENCY qualifier sacrifices some consistency protection for improved performance with many users.

2.4.6 Indexes

Rdb/VMS can use indexes to locate specific records using the database key for those records. A **database key** or DBKEY, is a pointer or address that indicates a specific record in the database. There is a separate index structure or B-Tree structure for each index (system or user) defined in the database. Each B-Tree structure is created by linking index nodes together in a balanced hierarchical structure. These nodes are also horizontally linked in "low to high" key value. The links between the nodes are created by using the database keys. Thus, updating records containing indexed fields means updating index nodes as well. During the database design phase, the database administrator or owner of the database should identify certain fields in each relation as primary keys and foreign keys.

A **primary key** is the key you select to be the principal identifier of each row in a relation. It is best if the field you select as a primary key is unique and stable, because the number of input/output operations necessary to update an index are high and thus costly. Therefore, you can use the primary field to locate a specific record or record, and update other, non-indexed fields, in those records. In this way, you benefit from the efficient access methods Rdb/VMS uses to locate the records you need, but you do not suffer the overhead penalty of updating the index nodes. A **foreign key** is an attribute or group of attributes in one relation that is a primary key of another relation. You can use foreign keys for joins.

You should decide which fields are important to index to reduce the number of write locks on the records in the relation. For example, you can start a transaction specifying the share mode, SHARED, and the lock type, WRITE. Another user can enter an identical START_TRANSACTION statement to read or update records in the same relation you have accessed. If no indexes are defined for the key field, Rdb/VMS must physically scan each record in the database itself, placing write locks on all of the records it touches. The other transaction attempts to select records from the same relation and conflicts with your transaction because your transaction has already placed locks on those records. Your transaction may even promote the locking to the EXCLUSIVE level, and allow no other user to access any of the records in the relation. Other users must terminate their transactions and reenter the START_TRANSACTION statement to select the records or wait until the records in that relation are available again.

If the fields you use to select records for your transaction are indexed, Rdb/VMS can refer to the index tables to locate only the records you need. Rdb/VMS will also place write locks on the index nodes that contain the database keys to those records and thus allow other users to access the remaining records in the relation. Use an index to locate records, and increase database concurrency by reducing possible deadlocks and making more resources available to other database operations. For further information on primary and foreign keys and indexes, see the *VAX Rdb/VMS Guide to Database Administration and Maintenance.*

2.4.7 Transaction Scope

Remember, a transaction is a unit of database activity you perform with one statement or many statements. The START_TRANSACTION statement that marks the start of a transaction and the COMMIT or ROLLBACK statement that terminates the transaction identify the scope of the transaction. Rdb/VMS executes either all of the statements in the scope of the transaction or none of them. Before you begin your transaction, you should determine the tasks you want to accomplish. Some of these tasks might be:

- Data retrievals from the database
- Changes you want to make to existing records in the database
- Changes to the database entity definitions

If you mix tasks in a transaction, you may want to undo some tasks and keep others. By restricting each transaction to a specific task, you can roll back certain operations and make others permanent.

```
RDO> START_TRANSACTION                ┐  (Transaction begins, transaction
RDO>         .              Scope     │   mode granted)
RDO>         .                of      │
RDO>         .          Transactions  │
RDO> COMMIT                           │  (Transaction ends, transaction
                                      ┘   mode returns to default)
```

2.4.8 Ending a Transaction

An update transaction can physically change the values in the database. In the following example, the PERSONNEL file is invoked and a START_TRANSACTION statement reserves the EMPLOYEES relation for an update transaction.

```
RDO> INVOKE DATABASE FILENAME PERSONNEL

RDO> START_TRANSACTION READ_WRITE
cont>        RESERVING EMPLOYEES FOR SHARED WRITE

RDO>
```

Before each update is flushed to disk, the original record is written to the run-unit journal file (file type .RUJ). Each user who performs an update has an .RUJ file in his or her SYS$LOGIN for the life of a transaction. After all the updated records have been flushed to disk, the EMPLOYEES relation has new records added to it. Figure 2-3 shows the effect of an update on a database.

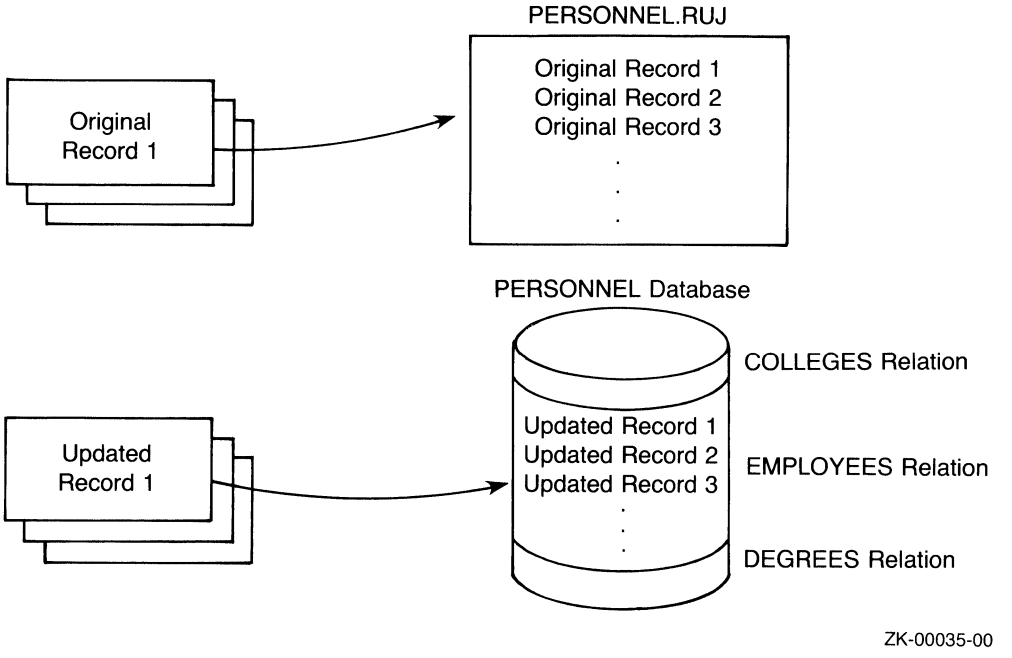


Figure 2-3: Run-Unit Journal File During an Update Transaction

You can terminate an Rdb/VMS transaction in two ways:

- COMMIT

  Use the COMMIT statement to make your changes permanent. This causes Rdb/VMS to invalidate the run-unit journal file and to make it ready for further transactions.

```
RDO> COMMIT
RDO>
```

Figure 2-4 shows the effect of a COMMIT statement on a database.

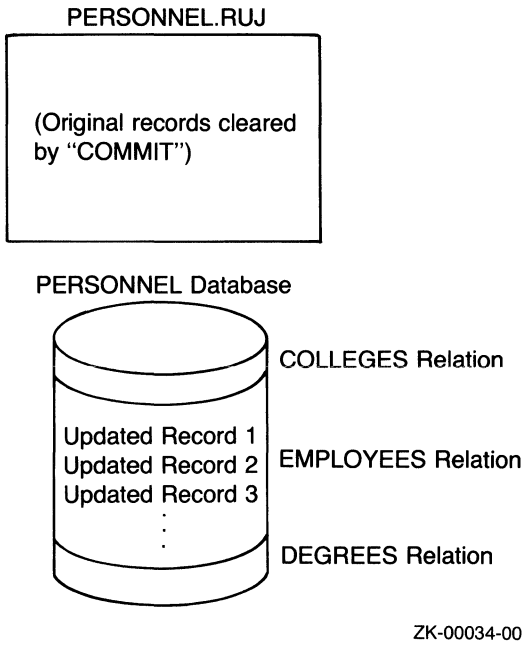


Figure 2-4: Run-Unit Journal File with COMMIT

- ROLLBACK

  Use the ROLLBACK statement to undo the changes you have made to the database within the scope of a transaction. The ROLLBACK statement uses the run-unit journal file to bring the database back to its pre-transaction state.

```
RDO> ROLLBACK
RDO>
```

Figure 2-5 shows the effect a ROLLBACK statement has on the database. Because the updates actually change the state of the database, the run-unit journal file is used to return the database to its pre-transaction state by writing the original records back to the database. When the transaction terminates, the EMPLOYEES relation is unchanged.

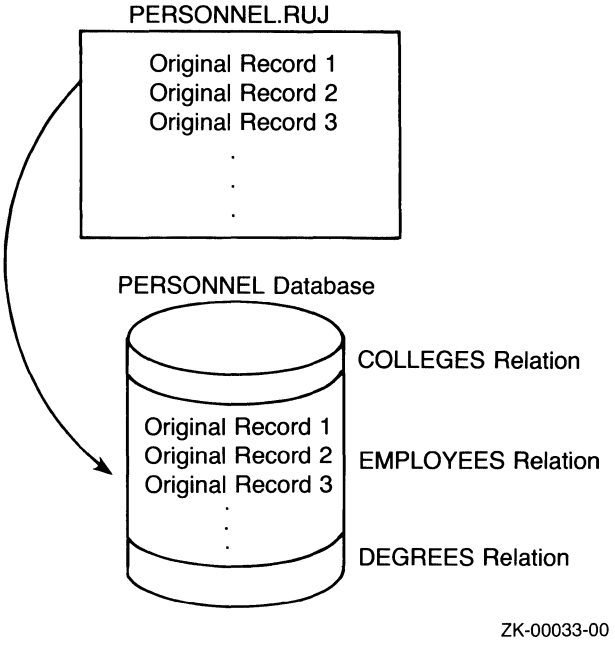


Figure 2-5: Run-Unit Journal File with ROLLBACK

2.5 The Optimizer

Because a relational database model represents the user's view of the data stored in the database, determining the best way to retrieve that data can be a very complex task. Rdb/VMS contains an optimizer that automatically analyzes every query to determine the most efficient method of access to the data. Efficiency can be measured as the number of disk accesses required to retrieve data values in the database.

The optimizer is a sophisticated component of Rdb/VMS that uses a combination of algorithms to evaluate the query and arrive at a low-cost solution to retrieve the data in the database. The order in which you specify joins and the order of the clauses in the RSE does not, in most cases, influence the order the optimizer uses to satisfy your query. You can, however, provide the optimizer with optional access methods by defining indexes for fields you use frequently in your queries. Database performance is directly affected by the ability of the system to access the record or records stored on disk through input/output (I/O) operations. The greater the number of I/O operations, the longer it takes to find and retrieve records that satisfy a query. To keep I/Os to a minimum, Rdb/VMS uses the optimizer.

To evaluate a query, the optimizer:

- Develops alternative solutions for retrieving data
- Affixes a cost factor to each solution based on estimated I/O
- Chooses the most cost-effective solution in terms of the least number of I/Os required to fetch the record

The optimizer evaluates every query in terms of an efficient access, so you do not have to be overly concerned about how to construct your queries. As a database designer, however, you can assist the optimizer by extending its access options. For example, if your query includes only those fields for which indexes are defined, you are providing the optimizer with an option to retrieve data directly from the index without scanning the relation sequentially.

However, the algorithms the optimizer uses may result in its not using the index on that field at all, if the data can be retrieved directly from the relation with fewer I/Os than by using the index(es).

The following list describes some of the tasks the optimizer performs to find the best solution for a query that contains one or more CROSS clauses.
The optimizer:

- Breaks down a query into alternative sequences of two relational joins
- Finds the best way to perform each join based on the relative, estimated costs of each access method
- Estimates cardinality (number of records to be retrieved) of each join based on a **join predicate,** which is the RSE supplied by the user, and the presence of indexes for specific fields
- Determines overall cost of each strategy
- Selects minimum cost strategy
- May use only the index if it contains all of the data necessary to satisfy the query, or if the index provides a useful ordering of records. For example, when a query names one field in the RSE and two other fields in the print list, all three fields must have indexes defined for them in order for the optimizer to choose an access method that uses only the index.

The optimizer chooses one of the following methods for retrieving data from a relation:

- Sequential retrieval

  Accesses the database pages for a relation sequentially, and searches for the field values of the records directly on the page.

- Index retrieval

  Accesses a determined index structure and retrieves the DBKEY of a record. Rdb/VMS then uses the DBKEY to directly access the data record to which it belongs.

- Index-only retrieval

  Accesses only the index data. If the desired data is located in an index key, Rdb/VMS can obtain the data without going to the relation itself.

- DBKEY retrieval

  Accesses the relation's data directly through the DBKEY (logical address) record pointer.

2.6 Sample Interactive Session Using the START_TRANSACTION Statement

The following sample command file shows the READ_ONLY and READ_WRITE versions of the basic START_TRANSACTION statement.

```
!  The statements in the scope of
!  this transaction only examine
!  the database. The transaction does not
!  change any values.
!
RDO> START_TRANSACTION READ_ONLY

!
!  Display the number of records in
!  the EMPLOYEES relation.
!
RDO> PRINT COUNT OF E IN EMPLOYEES
 101

!
!  How many employees live in Rochester?
!
RDO> PRINT COUNT OF E IN EMPLOYEES WITH E.CITY = "Rochester"
 7

!
!
!  If you attempt to change the database
!  by erasing all 'Rochester' records
```

```
!  with the ERASE statement:
!
RDO> FOR E IN EMPLOYEES WITH E.CITY  = "Rochester"
cont>     ERASE E
cont> END_FOR

!
!  You are attempting to update
!  the database.
!  RDO returns an error message:
!
%RDB-F-READ_ONLY_TRANS, attempt to update from a read_only transaction
!
!
!  Display the records
!  of employees who live in Rochester.
!
RDO> FOR E IN EMPLOYEES WITH E.CITY = "Rochester"
cont>   PRINT
cont>     E.LAST_NAME,
cont>     E.FIRST_NAME,
cont>     E.EMPLOYEE_ID,
cont>     E.CITY
cont> END_FOR
 Vormelker        Daniel       00242       Rochester
 Edwards          Keith        00254       Rochester
 Orlando          Johanna      00269       Rochester
 DuBois           Alvin        00275       Rochester
 Chase            Stan         00336       Rochester
 Boudreau         Wes          00346       Rochester
 Stornelli        Franklin     00437       Rochester

!
!  Terminate the READ_ONLY transaction scope with a
!  COMMIT or ROLLBACK statement. Then issue a new START_TRANSACTION
!  statement with update, READ_WRITE, access.
!
RDO> COMMIT

!
!  Start a new transaction allowing
!  changes to be written to the database.
!
RDO> START_TRANSACTION READ_WRITE

!
!  Display all records where the field
!  CITY contains the value 'Rochester'.
!
RDO> FOR E IN EMPLOYEES WITH E.CITY = "Rochester"
cont>   PRINT
cont>     E.EMPLOYEE_ID,
cont>     E.CITY,
cont>     E.POSTAL_CODE
cont> END_FOR
 00242   Rochester              03867
 00254   Rochester              03867
```

```
 00269   Rochester              03867
 00275   Rochester              03867
 00336   Rochester              03867
 00346   Rochester              03867
 00437   Rochester              03867


!
!  Change the value of the POSTAL_CODE field for
!  all the 'Rochester' records.
!
RDO> FOR E IN EMPLOYEES WITH E.CITY = "Rochester"
cont>   MODIFY E USING
cont>     E.POSTAL_CODE = "03801"
cont>   END_MODIFY
cont> END_FOR
```

!! Verify the change. !

```
RDO> FOR E IN EMPLOYEES WITH E.CITY = "Rochester"
cont>   PRINT
cont>     E.EMPLOYEE_ID,
cont>     E.CITY,
cont>     E.POSTAL_CODE
cont> END_FOR
 00242   Rochester              03801
 00254   Rochester              03801
 00269   Rochester              03801
 00275   Rochester              03801
 00336   Rochester              03801
 00346   Rochester              03801
 00437   Rochester              03801


!
!  Delete the 'Rochester' records.
!
RDO> FOR E IN EMPLOYEES WITH E.CITY = "Rochester"
cont>   ERASE E
cont> END_FOR


!
!  Are there any 'Rochester' records remaining?
!
RDO> PRINT COUNT OF E IN EMPLOYEES WITH E.CITY = "Rochester"
 0


!
!  Check to see that records are deleted.
!  (No records are displayed)
!
RDO> FOR E IN EMPLOYEES WITH E.CITY = "Rochester"
cont>   PRINT
cont>     E.CITY
cont> END_FOR
```

```
!
!  Add a new record.
!
RDO> STORE E IN EMPLOYEES
cont>   USING
cont>     E.EMPLOYEE_ID = "00502";
cont>     E.LAST_NAME = "Towne";
cont>     E.CITY = "Manchester";
cont>     E.POSTAL_CODE = "03103"
cont> END_STORE

!
!  Verify the store.
!
RDO> FOR E IN EMPLOYEES WITH E.LAST_NAME = "Towne"
cont>   PRINT
cont>     E.EMPLOYEE_ID,
cont>     E.LAST_NAME,
cont>     E.CITY,
cont>     E.POSTAL_CODE
cont> END_FOR
 00502   Towne            Manchester             03103

!
!  If you want to make the changes
!  to the database permanent, enter
!  the COMMIT statement.
!
!  If you do not want the changes
!  applied to the database, enter
!  the ROLLBACK statement.
!  Entering ROLLBACK causes the
!  'Rochester' records
!  to be retained in the database
!  with no changes
!  and the 'Manchester' record is not added.
!
RDO> COMMIT
RDO> FINISH (optional)
RDO> EXIT
```

Using Record Selection Expressions 3

This chapter shows you how to use record selection expressions (RSEs) to select and display values from a database. You use an RSE to select a group of records and then to manipulate the data from those records. Note that Rdb/VMS also lets you include database queries that use either embedded data manipulation statements or Callable RDO in your application programs.

Before you try to access large numbers of records, be certain that your query accurately performs the operations you request. Use RDO to make sure you are retrieving only the data you want. You can test each of the queries interactively to see examples of the output.

The next several sections describe how to:

- Restrict the number of records retrieved, using the FIRST clause
- Retrieve records that satisfy a particular set of conditions, using the WITH clause
- Sort records in a specified order, using the SORTED BY clause
- Eliminate duplicate values for fields, using the REDUCED TO clause

3.1 Forming Streams of Records

A record stream can consist of all the records in a relation, or selected records. You can form a record stream:

- With a FOR statement
- With a START_STREAM statement

In both statements, an RSE identifies the records that form the record stream.

Having chosen the records you wish to retrieve, you enter a PRINT statement in RDO to specify what fields you want displayed. The PRINT statement displays data on the terminal so you can be certain you have selected the correct records. Once you have tested your query and want to include it in a program, this display feature is no longer necessary. If you want to assign database values to variables in a BASIC, COBOL, or FORTRAN RDBPRE program, change PRINT to GET and name a host language variable. (In C and PASCAL programs, use a simple assignment statement.) Rdb/VMS assigns the database value to the variable instead of displaying the value. Then you can display or manipulate that value using the host language verbs. See the *VAX Rdb/VMS Guide to Programming* for details about converting your RDO queries to host language application programs.

3.2 Retrieving All the Records in a Relation

FOR R IN RELATION

One of the simplest operations in Rdb/VMS is selecting all the records in a relation.

The following FOR statement contains an RSE that forms a record stream consisting of all the records in the EMPLOYEES relation:

```
FOR E IN EMPLOYEES
```

The expression E IN EMPLOYEES is a record selection expression that selects records from the EMPLOYEES relation. This RSE includes every record of the EMPLOYEES relation in the record stream.

The character E in the first line of the RSE is a context variable. A **context variable** is a temporary name you assign to the record stream created by the RSE. In subsequent lines of the RSE, the context variable and a period (.) appear before each field name. By qualifying each field name with the context variable and a period, you indicate clearly to RDO to which relation each field belongs. If an RSE statement refers to more than one relation, you assign a unique context variable to each relation.

Context variables can be up to 31 characters long. Try to choose a context variable that you can easily associate with the relation. Using the first letter or two of a relation's name is a good idea.

The RDO statement FOR ... END_FOR includes the RSE and identifies a record stream. Other statements included in the FOR ... END_FOR statement operate on each record in this record stream. For example, to display data about all employees in the EMPLOYEES relation, you use the PRINT statement. The complete query is shown in the example that follows.

Examples

Example 1

```
FOR E IN EMPLOYEES
  PRINT
    E.LAST_NAME,
    E.FIRST_NAME,
    E.EMPLOYEE_ID
END_FOR
```

Example 1 displays three fields from each record in the EMPLOYEES relation. Note two important rules about using RDO PRINT statements:

- Qualify each of the field names with the context variable associated with the field's relation.
- Use commas to separate the expressions in the list.

The RSE selects records for inclusion in the record stream. The PRINT statement retrieves one record at a time and specifies fields from those records to be displayed.

You can also display data for all of the fields in the relation by using a special format of the PRINT statement. Instead of specifying each field name individually, substitute an asterisk (*) for the list of field names. RDO prints each of the field names in the relation as the first line of the display.

Example 2

```
FOR FIRST 5 E IN EMPLOYEES
   PRINT
   E.*
 END_FOR
```

When a relation such as EMPLOYEES contains many fields, RDO wraps the remainder of a long record onto the next line of your terminal. The resulting display can be difficult to read.

For a more readable display of records with many field values, create a command file with the extension .RDO; for example, REPORT1.RDO. In this file, you can include a query such as the one in example 2, and specify an output file with the SET OUTPUT <file-spec> command. Before you issue the SET OUTPUT command, use the SET NOVERIFY command. This suppresses duplicate lines of terminal output in the log file from each RDO statement you enter on the terminal, or from statements executed in an indirect command file. To close the output log file, enter SET NOOUTPUT or type SET OUTPUT without specifying a destination file name as shown on the next page.

```
SET NOVERIFY
SET OUTPUT REPORT1.LOG
FOR E IN EMPLOYEES
   PRINT
   E.*
 END_FOR
SET NOOUTPUT
```

You can execute the indirect command file by typing an at sign (@), followed by the name of the command file:

```
RDO> @REPORT1
```

You can then print the file, REPORT1.LOG, containing the results of an indirect command file, REPORT1.RDO, using the wide-line printer format instead of the 80-character limit of an interactive terminal; or you can use the SET TERMINAL/WIDTH=132. You can direct RDO to display special character strings or literal expressions by using the PRINT statement and quotation marks to enclose the string. You can combine literals with value expressions such as the statistical expression COUNT, separating each element with a comma. See Chapter 6 for a discussion of statistical expressions. The following query displays literal expressions and value expressions:

```
RDO> PRINT "Number of records in EMPLOYEES = ", COUNT OF E IN EMPLOYEES
Number of records in EMPLOYEES = 101
```

Example 3 shows you how to use a command file to format and display a simple report that shows the number of records in each relation of the PERSONNEL database.

Example 3

```
SET NOVERIFY
SET OUTPUT COUNT.LOG
PRINT "    "
PRINT "Statistics for database PERSONNEL follow: "
PRINT "  "
PRINT "Count of Employees -------> ", COUNT OF E IN EMPLOYEES
PRINT "Count of Jobs ------------> ", COUNT OF J IN JOBS
PRINT "Count of Degrees ---------> ", COUNT OF D IN DEGREES
PRINT "Count of Salary_History --> ", COUNT OF SH IN SALARY_HISTORY
PRINT "Count of Job_History -----> ", COUNT OF JH IN JOB_HISTORY
PRINT "Count of Work_Status -----> ", COUNT OF W IN WORK_STATUS
PRINT "Count of Departments -----> ", COUNT OF D IN DEPARTMENTS
PRINT "Count of Colleges --------> ", COUNT OF C IN COLLEGES
PRINT "  "
PRINT "Statistics Complete for Database: PERSONNEL"
PRINT "    "
```

3.3 Displaying Records in Sorted Order

```
FOR R IN RELATION_A SORTED BY R.FIELD_1
```

Use the SORTED BY clause of the RSE to signal RDO to arrange the records in any order you choose. The default is *ascending* order.

Assume you must arrange the EMPLOYEES records in alphabetical order by state:

Examples

Example 1

```
FOR E IN EMPLOYEES SORTED BY E.STATE
   PRINT
   E.STATE,
   E.CITY,
   E.EMPLOYEE_ID
 END_FOR
```

A field name that indicates the field on which the sort order of records is based, is called a **sort key**. In example 1, the single sort key is STATE.

Because the query in example 1 has only one sort key, it does not specify how RDO should arrange two or more records that have the same value for STATE. If you want to include cities in alphabetical order (A to Z) within the same state, use two sort keys, STATE and CITY.

Example 2

```
FOR E IN EMPLOYEES SORTED BY E.STATE, E.CITY
   PRINT
   E.STATE,
   E.CITY,
   E.EMPLOYEE_ID
 END_FOR
```

Specifying CITY as a second sort key ensures that Rdb/VMS arranges records with different values for CITY alphabetically within the same state. When you use more than one sort key, the first key is the **major sort key** and all other keys are **minor sort keys**. In the preceding example, STATE is the major sort key and CITY is a minor sort key.

Note

Rdb/VMS does not guarantee the sort order of the records unless you specify a sort key. You cannot assume that Rdb/VMS arranges records according to the values for any index key field of the relation. To control the arrangement of the records that Rdb/VMS displays, specify one or more sort keys.

3.3.1 Indicating Ascending or Descending Sort Order

```
FOR R IN RELATION_A
    SORTED BY ASCENDING R.FIELD_1,
              DESCENDING R.FIELD_2
```

When you use a sort key, RDO arranges the records in ascending order by that key according to the standard ASCII collating sequence; that is, Rdb/VMS arranges numeric values in numerical order and character fields in alphabetical order. To reverse the order, use the keyword DESCENDING. However, if you are using an indexed field as a sort key, the query may not be very efficient. This is because the optimizer must go to VMS and use the VMS Sort utility to do the job, thus resulting in increased costs in I/O operations.

You can sort a record stream by ascending values for one field and descending values for another field. To arrange the records in alphabetical order for the major sort key (STATE) and in reverse alphabetical order for the minor sort key (CITY), specify an explicit order for each field.

```
FOR E IN EMPLOYEES
 SORTED BY ASCENDING E.STATE,
           DESCENDING E.CITY
            PRINT
              E.STATE,
              E.CITY,
              E.EMPLOYEE_ID
 END_FOR
```

Now the records are in alphabetical order according to the major sort key, STATE. Therefore, if more than one record with the same STATE occurs, the records are ordered by DESCENDING CITY (Z to A). Note the minor sort key, CITY, specifies the keyword DESCENDING.

Note

Unless you explicitly specify the sort order for each minor sort key, the default sort order of any minor key is the same as the order for the last explicit or default sort key. The following table illustrates the default sort order for major and minor sort keys.

Table 3-1: Default Sort Order of Major and Minor Sort Keys

Major Key	Minor Key	Minor Key
ASCENDING (default)	ASCENDING (default)	ASCENDING (default)
DESCENDING (explicit)	DESCENDING (default)	DESCENDING (default)
ASCENDING (explicit)	ASCENDING (default)	ASCENDING (default)
ASCENDING (explicit)	DESCENDING (explicit)	DESCENDING (default)
DESCENDING (explicit)	ASCENDING (explicit)	ASCENDING (default)
DESCENDING (explicit)	DESCENDING (explicit)	DESCENDING (explicit)

Because the following RSE specifies a DESCENDING sort order for the major sort key, E.CITY, RDO sorts the other two minor sort keys, E.STATE and E.POSTAL_CODE, in descending order also:

```
FOR E IN EMPLOYEES
  SORTED BY DESCENDING E.CITY, E.STATE, E.POSTAL_CODE
```

3.3.2 Using Value Expressions as Sort Keys

```
FOR R IN RELATION_A
    SORTED BY ASCENDING (R.FIELD_1 - R.FIELD_2)
```

A sort key can also be a value expression that refers to one or more fields in a relation. Briefly, 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/VMS calculates the value associated with the expression and uses that value when executing the statement. See Chapter 6 for more information on value expressions.

The RSE in the following example finds the job codes with the greatest salary range. In this example, the sort key consists of a value expression that calculates the salary range for each job code. The query sorts the records of JOBS by range value, beginning with the smallest range.

Examples

Example 1

```
FOR J IN JOBS SORTED BY
  (J.MAXIMUM_SALARY - J.MINIMUM_SALARY)
  PRINT
  (J.MAXIMUM_SALARY - J.MINIMUM_SALARY),
  J.JOB_CODE,
  J.MAXIMUM_SALARY,
  J.MINIMUM_SALARY
END_FOR
```

To reverse the order of displayed values, include the explicit sort qualifier, DESCENDING.

Example 2

```
FOR J IN JOBS SORTED BY
  DESCENDING (J.MAXIMUM_SALARY - J.MINIMUM_SALARY)
  PRINT
  (J.MAXIMUM_SALARY - J.MINIMUM_SALARY),
  J.JOB_CODE,
  J.MAXIMUM_SALARY,
  J.MINIMUM_SALARY
END_FOR
```

3.4 Restricting the Number of Records: The FIRST Clause

```
FOR FIRST 10 R IN RELATION
```

Remember that RDO allows you to experiment with different queries to find the ones best suited to your programming needs. For example, you need not display all of the records of the database to see whether your queries work correctly. RDO has a special clause, FIRST *n*, that limits the number of records you display. The integer (*n*) in the FIRST *n* clause tells RDO how many records to retrieve.

Assume you must display the first ten records from the EMPLOYEES relation, and need to look at only three fields, STATE, CITY, and EMPLOYEE_ID, from each record. The records retrieved by RDO are not in any specific order. As you update the contents of the EMPLOYEES relation by adding, erasing, or modifying records, the order of records stored in the database changes. Therefore, unless you specify to RDO the order you want the records displayed, the FIRST 10 clause retrieves what might appear to be ten random records. See the earlier section of this chapter for details on the SORTED BY clause.

```
FOR FIRST 10 E IN EMPLOYEES
  PRINT
  E.STATE,
  E.CITY,
  E.EMPLOYEE_ID
END_FOR
```

Note

If the RSE does not specify a sort order, you cannot predict which ten records RDO will display. When you use the SORTED BY clause in the RSE, the FIRST *n* clause takes the specified number of records from the *sorted* records. RDO does the sort first, then displays the number of records specified in the FIRST *n* clause from this sorted order. Remember to use a SORTED BY clause when you begin an RSE with a FIRST *n* clause.

3.5 Specifying Conditions to Retrieve Records: Relational and Logical Operators

Assume you want to find all employees in the PERSONNEL database whose last name is "Toliver."

```
FOR E IN EMPLOYEES WITH E.LAST_NAME = "Toliver"
   PRINT
   E.FIRST_NAME,
   E.LAST_NAME,
   E.EMPLOYEE_ID
 END_FOR
```

The clause, WITH E.LAST_NAME = "Toliver", is a *conditional expression.* It is equivalent to:

```
If LAST_NAME = "Toliver"
```

The value of this conditional test for a record is either true or false, depending on whether the field value in that record satisfies the condition (true), or does not satisfy the condition (false). A conditional expression restricts the record stream to those records that satisfy the condition. If a conditional expression for a record is false, RDO will not include that record in the record stream.

The equal sign (=) is a relational operator because it links a data value of a field or other value expression to a value. The relational operator EQUAL (=) is case sensitive. This means that RDO reads "Toliver" and "toliver" as two different character strings. In this case, if you specify a value "toliver" for an employee record stored in the database as "Toliver", RDO does not find the record you want.

In a program, the following conditional expression tests a database field value and the value of a host variable:

```
FOR E IN EMPLOYEES WITH E.LAST_NAME = LAST-NAME
```

Here, the value of the conditional expression depends on the current value of the host variable LAST_NAME. Table 3-2 summarizes most of the relational operators.

Table 3-2: RDO Relational Operators

Permitted Symbols	Relational Operation
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.
BETWEEN	True if the first value expression is equal to or between the second and third value expressions.
ANY	True if the record stream specified by the RSE includes at least one record. If you add NOT, the condition is true if there are no records in the record stream.
MATCHING	True if the second expression matches a substring of the first value expression. MATCHING uses these special characters:

	* Matches any string in that position % Matches any character in that position
MISSING	True if the value expression is null. See Chapter 7 for information on missing values.
UNIQUE	True if the record stream specified by the RSE includes only one record. If you add NOT, the condition is true if there is more than one record in the record stream or the record stream is empty.
CONTAINING	True if the string specified by the second string expression is found within the string specified by the first. Not case sensitive.
STARTING WITH	True if the first characters of the first string expression match the second string expression. Case sensitive.

Note

In all cases except the MISSING operator, if either value expression is null, the value of the condition is null.

You can combine several conditional expressions by using a logical operator to form a compound conditional expression. A **logical operator** joins two or more conditional expressions together. The logical operators are:

- AND - Evaluates to true if two or more conditions linked by the AND operator are satisfied.
- OR - Evaluates to true if at least one of several conditions linked by the OR operator is satisfied.
- NOT - Returns all other records in the record stream *except* those identified by the conditional expression following the NOT operator. Therefore, you retrieve the complement of the record stream identified by the RSE.

For information on using logical operators, see the section in this chapter on specifying compound conditions for records.

3.5.1 Using the WITH Clause as a Conditional Expression

Previous examples used the FIRST *n* clause to limit the number of records to be displayed. This is useful when you want to see a sample of all the records you intend to retrieve. But most queries ask to see only a *limited* number of the records, qualified in some way. There are several ways to restrict the record stream. One way is to use the WITH clause to test records of a relation based on field values. For example:

```
FOR E IN EMPLOYEES WITH E.EMPLOYEE_ID = "00246"
```

Here, the WITH clause tests whether or not an employee's ID is 00246. The following sections illustrate further uses of the WITH clause.

3.5.2 Retrieving Records That Satisfy a Single Condition

```
FOR R IN RELATION_A
    WITH R.FIELD_1 = "XXX"
```

To select only those records with a particular field value, specify a value for that field by including the WITH clause in the RSE. RDO retrieves only those records with the specified field value.

For example, you can request RDO to display the records of the EMPLOYEES relation that have a specific value for the CITY field. The field value is the basis of the test that determines which records RDO retrieves. RDO limits the record stream to the records that meet your test. In this case, RDO retrieves only the records of employees living in a specific city. You can include more than one conditional test in the same RSE.

3.5.3 Specifying Compound Conditions for Records

The preceding section describes an RSE that includes a single conditional expression to test records. The following sections show how to retrieve those records that satisfy a compound condition.

3.5.3.1 Retrieving Records That Satisfy Two or More Conditions

```
FOR R IN RELATION_A
    WITH R.FIELD_1 = "XXX"
    AND R.FIELD_2 > "YYY"
```

When you want each record in the stream to satisfy two or more conditions, you can combine conditional expressions together with the AND logical operator. The record stream contains only those records that satisfy all the conditions within the compound conditional expression.

For example, to retrieve all part-time employees who live in Portsmouth, New Hampshire:

```
FOR E IN EMPLOYEES
 WITH E.CITY = "Portsmouth"
  AND E.STATE = "NH"
  AND E.STATUS_CODE = "2"
   PRINT
   E.EMPLOYEE_ID,
   E.CITY,
   E.LAST_NAME,
   E.STATUS_CODE
 END_FOR
```

Table 3-3 shows how Rdb/VMS evaluates a compound conditional expression formed with the AND logical operator. A and B stand for simple conditional expressions that are components of the compound conditional expression, A AND B.

Table 3-3: AND Logical Operator

A	B	A AND B
True	False	False
True	True	True
False	False	False
True	Missing	Missing
False	Missing	False
Missing	Missing	Missing

The next sections discuss two other logical operators: OR and NOT.

3.5.3.2 Retrieving Records That Satisfy One of Several Conditions

```
FOR R IN RELATION_A
    WITH (R.FIELD_1 GT "XXX")
     OR (R.FIELD_2 GT "YYY")
```

You may want to retrieve records that meet at least one of a series of conditions. To set up such a test, form a compound conditional expression with the OR logical operator. If any one of the component conditional expressions is true for a record, Rdb/VMS includes that record in the record stream.

The following compound conditional expression that uses the OR logical operator, retrieves information about all employees who have received graduate degrees:

```
FOR D IN DEGREES WITH
  (D.DEGREE = "MA" ) OR
  (D.DEGREE = "PhD")
    PRINT
    D.DEGREE,
    D.EMPLOYEE_ID,
    D.COLLEGE_CODE,
    D.DEGREE_FIELD
END_FOR
```

To find the records for employees with either an MA degree or a PhD, specify two conditions linked by the logical operator OR. This means that RDO includes a record in the stream if either or both of the two conditions are true.

Table 3-4 illustrates how Rdb/VMS evaluates a compound conditional expression formed with the logical operator OR. A and B stand for simple conditional expressions in the compound conditional expression, A OR B.

Note

You should enclose each conditional expression in parentheses and nest them to any level necessary to make the compound expression clear. Rdb/VMS evaluates the innermost expressions first and the outermost expressions last.

Table 3-4: OR Logical Operator

A	B	A OR B
True	False	True
True	True	True
False	False	False
True	Missing	True
False	Missing	Missing
Missing	Missing	Missing

3.5.3.3 Retrieving Records That Do Not Satisfy a Condition

```
FOR R IN RELATION_A
    WITH R.FIELD_1 = "XXX"
    AND NOT (R.FIELD_2 = "YYY")
```

The third logical operator, NOT, enables you to retrieve records that are *not* identified by the conditional expression. You can include the NOT operator in a simple or compound conditional expression. RDO restricts the record stream to those records that do not satisfy the conditional expression following the NOT logical operator.

Combining the NOT logical operator with the ANY relational operator allows you to refer to the records of a second relation. Records from the first relation (EMPLOYEES) appear in the stream only when *no record* in the second relation (DEGREES) meets the condition you specify. In other words, the first WITH clause of the RSE in the FOR statement contains a second WITH clause to test and restrict the stream, as in the following example.

Examples

Example 1

```
FOR E IN EMPLOYEES
       WITH NOT ANY D IN DEGREES
       WITH D.EMPLOYEE_ID = E.EMPLOYEE_ID
    PRINT
       E.EMPLOYEE_ID,
       E.LAST_NAME,
       E.FIRST_NAME
END_FOR
```

To access data about employees without college degrees, you need to access the EMPLOYEES and DEGREES relations. But you do not want to match related records from the two relations. You select each record from the EMPLOYEES relation and check the DEGREES relation to see if that employee's ID appears. (This means that the employee has a degree from some college.) If an employee's ID does not appear even once in the DEGREES relation, you want to include the corresponding record from EMPLOYEES in the stream. In the preceding example, the first WITH clause of the RSE is:

```
WITH NOT ANY D IN DEGREES
```

The second clause is:

```
WITH D.EMPLOYEE_ID = E.EMPLOYEE_ID
```

In evaluating this query, Rdb/VMS examines a record of the EMPLOYEES relation and compares the value of EMPLOYEE_ID with every record of the DEGREES relation. An EMPLOYEES record is included in the stream only if *no record* of the DEGREES relation has a matching value for EMPLOYEE_ID.

For each record of the EMPLOYEES relation, RDO may have to check every record of DEGREES to guarantee there is no matching record on EMPLOYEE_ID. As soon as a match occurs, the specified condition fails, and the search through the DEGREES relation ends. RDO does not include that record from EMPLOYEES in the stream.

Table 3-5 illustrates how Rdb/VMS evaluates a compound conditional expression formed with the logical operator NOT. In this table, the first column represents a conditional expression A; the second column is the complement of A, or a condition identifying all other records not identified by A.

Table 3-5: NOT Logical Operator

A	NOT A
True	False
False	True
Missing	Missing

You *cannot* use the NOT operator with the following relational operators:

- EQ (=)
- NE (< >)
- GT (>)
- GE (> =)
- LT (<)
- LE (< =)

When writing a query with a conditional expression that uses one of the relational operators listed, you can express inequality as follows:

```
WITH NOT (S.SALARY_AMOUNT = 30000)
WITH S.SALARY_AMOUNT NE 30000
WITH S.SALARY_AMOUNT <> 30000
```

The converse of the NOT ANY statement is the ANY statement which tests whether another stream is not empty. The following example shows a situation in which the ANY relational operator is useful.

Assume you must identify the records in the EMPLOYEES relation for employees who have a PhD. You would use the following RSE:

Example 2

```
FOR E IN EMPLOYEES WITH ANY D IN DEGREES WITH
       D.EMPLOYEE_ID = E.EMPLOYEE_ID
         AND D.DEGREE = "PhD"
 PRINT
     E.EMPLOYEE_ID,
     E.LAST_NAME,
     E.FIRST_NAME
END_FOR
```

Once again, the query checks the DEGREES relation for each record of the EMPLOYEES relation. But this time you need a record in DEGREES with a value of PhD for the DEGREE field and a matching EMPLOYEE_ID.

In evaluating this query, Rdb/VMS examines each record of the EMPLOYEES relation, and compares the value of EMPLOYEE_ID with every record of the DEGREES relation. The EMPLOYEES record appears in the record stream only if there is a record in the DEGREES relation that matches on EMPLOYEE_ID and also has a value of PhD for the DEGREE field.

The ANY relational operator allows you to refer to the records of a second relation, in this case, the DEGREES relation. Records from the first relation (EMPLOYEES) appear in the stream only when there is at least one record in the second relation (DEGREES) that meets the condition you specify.

3.5.4 Retrieving Records That Match a Pattern

A commonly used query asks which records have field values exactly matching a specific value. This type of query would help you find all the Massachusetts employees (WITH E.STATE = "MA"), or all the employees who are part-time (WITH E.STATUS_CODE = "2").

Recall the example in which you retrieved the records of all those employees whose last name is "Toliver".

```
FOR E IN EMPLOYEES WITH E.LAST_NAME = "Toliver"
   PRINT
   E.FIRST_NAME,
   E.LAST_NAME,
   E.EMPLOYEE_ID
 END_FOR
```

Notice the example uses a WITH clause to request a specific EMPLOYEES record. In this case, you supply the value "Toliver" for the exact match. You can test for exact substring matches using the MATCHING operator.

The MATCHING operator does true pattern matching. Like the CONTAINING operator, which is discussed in the section on retrieving records by partial matches, the MATCHING operator allows you to find a substring within a source string. In addition, the MATCHING operator lets you specify the position of the substring.

The following sample log file from an RDO terminal session, shows you the results of combinations of substrings and matching characters.

```
!
! Find the names in which the letters "on" come last
! in the name. Note that the last "*" is necessary because
! the field ends in an unknown number of spaces.
!
! The statement using the MATCHING operator in the following example
! is equivalent to using the CONTAINING operator.

! Notice MATCHING uses two special characters for matching patterns:

! *       Matches any string of zero or more characters.
! %       Matches any single character.
```

```
FOR FIRST 5 E IN EMPLOYEES WITH
    E.LAST_NAME MATCHING "*on *"
    REDUCED TO E.LAST_NAME
PRINT
E.LAST_NAME
END_FOR
 Aaron
 Burton
 Clinton
 Dixon
 Ferguson

!
! Find the names in which the letters "on" come
! after the first character in the name.
!

FOR FIRST 5 E IN EMPLOYEES WITH
    E.LAST_NAME MATCHING "%on*"
    REDUCED TO E.LAST_NAME
PRINT
E.LAST_NAME
END_FOR
 Connolly
 Jones
 Lonergan

!
! MATCHING is not case sensitive.
!

FOR FIRST 5 E IN EMPLOYEES WITH
    E.LAST_NAME MATCHING "%ON*"
    REDUCED TO E.LAST_NAME
PRINT
E.LAST_NAME
END_FOR
 Connolly
 Jones
 Lonergan

!
! MATCHING also works with numeric data types.  Find
! the salaries that begin with the number 3.  This
! is another way to find all the salaries in the
! range BETWEEN 30000 AND 39999.
!


FOR FIRST 5 S IN SALARY_HISTORY WITH
  S.SALARY_AMOUNT MATCHING "3*"
    PRINT S.SALARY_AMOUNT
END_FOR
 32254.00
 30598.00
```

```
 30880.00
 32589.00
 33944.00
!
! Find the salaries where the number 87 follows the
! first digit.
!

FOR FIRST 5 S IN SALARY_HISTORY WITH
  S.SALARY_AMOUNT MATCHING "%87*"
    PRINT S.SALARY_AMOUNT
END_FOR
 48797.00

 18705.00
 18778.00
 18778.00
 18746.00
```

3.5.5 Retrieving Records That Do Not Match a Pattern

```
FOR R IN RELATION_A
    WITH R.FIELD_1 < > "XXX"
```

Sometimes you want to retrieve the records that do *not* match a pattern. Use the NOT EQUAL (NE) relational operator to include only those records with values for the field that do not match the specified value expression. You can use the symbol < > or NE to indicate that the values are not equal.

To find the employees who have earned a degree at a college other than the Massachusetts Institute of Technology, use this RSE:

```
FOR D IN DEGREES WITH D.COLLEGE_CODE <> "MIT"
   PRINT
   D.EMPLOYEE_ID,
   D.COLLEGE_CODE,
   D.DEGREE
END_FOR
```

This query retrieves all the records in the DEGREES relation that do *not* have a value of MIT for the COLLEGE_CODE field.

Note that NE (< >), like EQUAL, is case sensitive. RDO does not treat "MIT" and "mit" or "Mit" in the same way.

If COLLEGE_CODE is missing, then the value of the conditional expression is neither true nor false, but missing. The record stream does not include such a record. See Chapter 6 for an explanation of missing values.

3.5.6 Retrieving Records by Partial Matches

```
FOR R IN RELATION_A
    WITH R.FIELD_1 CONTAINING "XXX"
```

You can search for records in which a field value contains a specific sequence of characters. Two relational operators perform this type of search:

- STARTING WITH (case sensitive)
- CONTAINING (not case sensitive)

For example, assume you do not remember how to spell an employee's name, but you do know that the name begins with "Tol". To find and display the record for that employee, you could use the following query:

Examples

Example 1

```
FOR E IN EMPLOYEES WITH E.LAST_NAME STARTING WITH "Tol"
   PRINT
   E.FIRST_NAME,
   E.LAST_NAME,
   E.EMPLOYEE_ID
 END_FOR
```

Use the STARTING WITH relational operator to search for records in the EMPLOYEES relation with a last name beginning with "Tol". The STARTING WITH relational operator, like EQUAL and NE, *is* case sensitive. If you ask for employees whose last names start with "TOL", RDO does not retrieve the record because the database stores the field value as "Tol", not "TOL".

Use the CONTAINING relational operator for searches that are not case sensitive. If you substitute CONTAINING for STARTING WITH in example 1, RDO will retrieve the Toliver records.

Example 2

```
FOR E IN EMPLOYEES WITH E.LAST_NAME CONTAINING "TOL"
   PRINT
   E.FIRST_NAME,
   E.LAST_NAME,
   E.EMPLOYEE_ID
END_FOR
```

You can use the CONTAINING operator for searches on any part of a field value, not just the beginning. Example 3 shows another query that retrieves the Toliver records.

Example 3

```
FOR E IN EMPLOYEES WITH E.LAST_NAME CONTAINING "IVER"
   PRINT
   E.FIRST_NAME,
   E.LAST_NAME,
   E.EMPLOYEE_ID
END_FOR
```

The CONTAINING relational operator allows you to search through the relation, specifying only as much of the field value as you know. In most cases, the CONTAINING operator is more useful than the EQUAL operator because it ignores the uppercase or lowercase form in which the data value is stored. Only the characters themselves determine the match.

Note

Rdb/VMS does not use the index tables for indexed fields to evaluate conditional expressions that use CONTAINING. If you are searching on the initial substring of a field value, use STARTING WITH instead of CONTAINING. To get better performance, use the STARTING WITH clause or conditional expressions that contain indexed fields.

3.5.7 Retrieving Records by Range Retrieval

```
FOR R IN RELATION_A
    WITH R.FIELD_1 GT "XXX"
```

You may wish to retrieve records on the basis of a range of values in a specific field. The next example displays the supervisor ID and employee ID for all employees who started their jobs after December 10, 1982.

```
FOR JH IN JOB_HISTORY WITH JH.JOB_START GT "10-DEC-1982"
   PRINT
   JH.SUPERVISOR_ID,
   JH.EMPLOYEE_ID,
   JH.JOB_START
END_FOR
```

The JOB_HISTORY relation in the PERSONNEL database contains the required data. This query is a range retrieval because it specifies records with a range of field values for JOB_START that are greater than (after) a certain date. You need to specify a WITH clause that includes the GREATER THAN (GT) relational operator.

3.5.8 Retrieving Segmented Strings

The segmented string is a special Rdb/VMS data type designed to handle large pieces of data with a segmented internal structure. The maximum size of a string segment is 64K bytes. Except for the length of the string's segments, Rdb/VMS does not know anything about the type of data contained in a segmented string. In a segmented string, you can store large amounts of text, long strings of binary input from a data collecting device, or graphic data. A program can then retrieve the data from the database and handle it in the appropriate way.

Because Rdb/VMS does not know what kind of data is contained in a segmented string, you cannot perform many of the standard data manipulation functions on it. You cannot use relational operators, such as EQUAL and CONTAINING, to compare segmented strings. Rdb/VMS does not perform any data type conversion on data that is transferred into or out of a segmented string.

Rdb/VMS defines a special name to refer to the segments of a segmented string. This name is equivalent to a field name; it names the "fields" or segments of the string. Furthermore, because segments can vary in length, Rdb/VMS also defines a name for the length of a segment. You must use these names in the value expressions that you use to retrieve the length and value of a segment.

These names are:

- RDB$VALUE

  The value stored in a segment of a segmented string

- RDB$LENGTH

  The length in bytes of a segment

Because a single segmented string field value is made up of multiple segments, you must manipulate the segments one at a time. Therefore, segmented string operations require an internal looping mechanism, much like the record stream set up by a FOR or START_STREAM statement. The following example retrieves and prints two segmented strings:

```
FOR R IN RESUMES WITH R.EMPLOYEE_ID = '00164'
FOR S IN R.RESUME
 PRINT S.RDB$LENGTH, S.RDB$VALUE
 END_FOR
END_FOR
```

3.6 Eliminating Duplicate Values

```
FOR R IN RELATION_A
    REDUCED TO R.FIELD_1, R.FIELD_2
```

Many records in a database contain fields that hold duplicate values. For example, the field in the EMPLOYEES relation called CITY can have any number of values assigned to it (the names of all cities in Massachusetts), or every occurrence of CITY can have the same value (Boston). Some queries look for unique values for one or more fields in a record. For example, assume you want a list of the cities in which employees live. If a city occurs more than once, you want the city included only once.

The RSE in example 1 finds the cities in which all current employees live. Because many employees live in the same city, this query uses the REDUCED TO clause to restrict the final output to a unique value for the city field.

Examples

Example 1

```
FOR E IN EMPLOYEES REDUCED TO E.CITY
  PRINT E.CITY,
  PRINT E.STATE
END_FOR
```

Example 1 forms a record stream from the EMPLOYEES relation, using an RSE with a REDUCED TO clause. The field named in the REDUCED TO clause (E.CITY) is called the **reduce key**. This clause eliminates any duplicate values for the field or combination of fields specified as reduce keys. Of the records retrieved in example 1, the CITY field is the only reduce key; the STATE field is not reduced to unique name values, but it is reasonable to display the E.STATE field with E.CITY because both fields are in the same relation.

Assume you want to collect information about the educational experience of company employees. You need to display data about each college attended and the degree granted by that college. If several employees attended the same college, display the individual college and degree data only once.

You want to restrict the stream to unique *combinations of values* for the college code and the degree. Example 2 requires two reduce keys: COLLEGE_CODE and DEGREE.

Example 2

```
FOR D IN DEGREES REDUCED TO D.COLLEGE_CODE, D.DEGREE
  PRINT
  D.COLLEGE_CODE,
  D.DEGREE
END_FOR
```

A query that specifies a REDUCED TO clause restricts the record stream by excluding duplicate records. The record stream contains only unique values or combinations of values for the reduce keys.

You can include more than one field when looking for unique values. In general, limit your display to those fields specified in the REDUCED TO clause. Displaying values of fields not specified in the REDUCED TO clause may yield unpredictable results. For example, if you want to display employee identification numbers as well, the results can be misleading.

Example 3

```
FOR E IN EMPLOYEES REDUCED TO E.CITY
   PRINT
   E.CITY,
   E.EMPLOYEE_ID
END_FOR
```

Rdb/VMS lists the unique occurrences of CITY, such as Boston, but it does not know which EMPLOYEE_ID you want, 00123 or 00127. Whichever EMPLOYEE_ID RDO displays depends on how Rdb/VMS searches the records in the relation and selects a value for use. That search sequence can be different each time you execute the query.

Refer to Figure 3-1. The left side of the figure shows values for just three fields of a record as they actually occur in the EMPLOYEES relation: CITY, STATE, and EMPLOYEE_ID. The selected values on the right side are all unique occurrences for the CITY and STATE fields if Rdb/VMS searched sequentially from the beginning of the file. However, as records are added, deleted, or modified, unique values are likely to occur in a much different sequence. For this reason, attempts to associate the values in the EMPLOYEE_ID field with a corresponding unique value will almost always produce meaningless results.

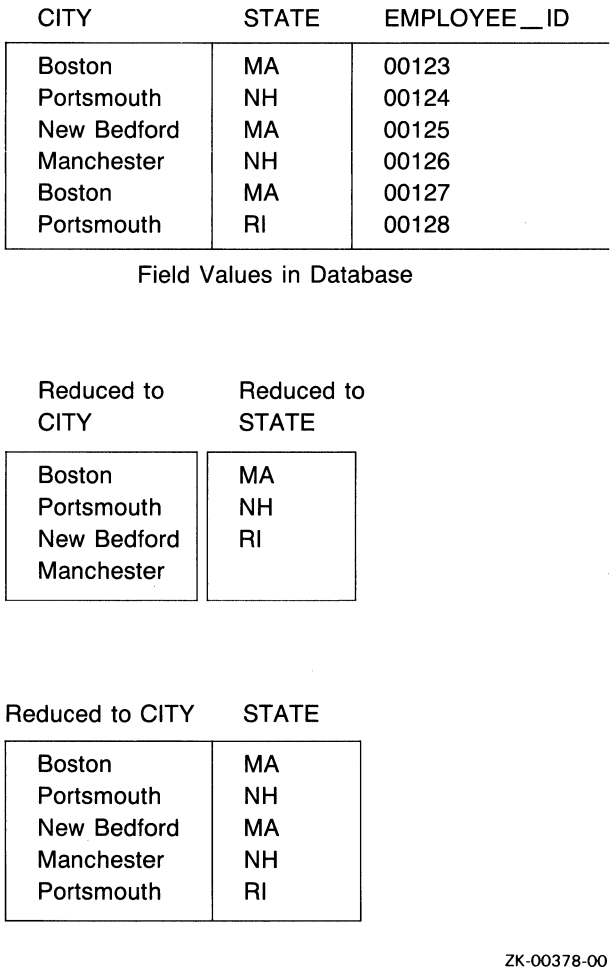


Figure 3-1: Finding Unique Values from Field Values

3.7 Testing for a Unique Record Occurrence

You can also test a relation to determine the uniqueness of a record occurrence. A record is unique if there is exactly one record meeting the record selection expression. You can find a unique record by specifying a field whose value makes that record unique. For example, if a value in the CITY field of the EMPLOYEES relation occurs only once in the database, the record containing that field value is unique. To retrieve a record based on its unique characteristics, include the UNIQUE operator as part of your RSE.

Assume the company wishes to locate any city in which only one employee lives.

Examples

Example 1

```
FOR E IN EMPLOYEES
   WITH UNIQUE EMP IN EMPLOYEES
   WITH E.CITY = EMP.CITY
     PRINT E.CITY,
      E.LAST_NAME,
      E.FIRST_NAME
END_FOR
```

A record is not unique if there is more than one record in the relation that meets the record selection expression. Use the NOT UNIQUE operator to find the records in a relation that are not unique and do not match the record selection expression.

Example 2

```
FOR E IN EMPLOYEES
   WITH NOT UNIQUE EMP IN EMPLOYEES
   WITH E.CITY = EMP.CITY
     PRINT E.CITY,
      E.LAST_NAME,
      E.FIRST_NAME
END_FOR
```

The UNIQUE operator differs from the REDUCED TO clause in one important way. When you use the REDUCED TO clause, you can display only the values of the fields named in that clause. Displaying other field values will produce unanticipated results. The UNIQUE operator locates an entire record whose field value makes the record unique and allows you to display any or all fields from the qualifying record.

The following table summarizes the effects of the ANY, NOT ANY, UNIQUE, and NOT UNIQUE operators.

Table 3-6: Testing for the Existence of Records with ANY and UNIQUE Operators

Operator	True, if
ANY RSE	At least one record found
NOT ANY RSE	No records found
UNIQUE RSE	Only one record found
NOT UNIQUE RSE	Records matching the RSE is not one

Retrieving Records and Joining Relations 4

You can use relational operators and conditional expressions to retrieve records from a single relation, or from several relations joined together. This chapter illustrates how to join relations to retrieve information contained in more than one relation.

4.1 Using the CROSS Clause to Combine Data

Although the answers to some queries come from just one relation, you must sometimes look at two or more relations to find the information you need.

When you design a relational database, you try to divide groups of data elements into separate relations. See the *VAX Rdb/VMS Guide to Database Design and Definition* for details and examples of normalization. Common fields in each relation link one relation with another. Using separate relations helps you avoid storing redundant data.

For example, you need not store information about each job an employee has held in the company with employee information. You can store employee information in an EMPLOYEES relation and job history information in a JOB_HISTORY relation. The relations share a common field: EMPLOYEE_ID. When you need to retrieve information about a worker and his or her job history, you join the two relations on the EMPLOYEE_ID field.

You can join one relation with another, or you can join one relation with itself. The following sections describe each variation.

4.1.1 Joining Records from Two Relations

The simplest type of join combines records from two relations that have a matching value for a common field. When you need to access data from two relations, you can either access each relation separately or join related records from the two relations. The following examples represent typical problems and show you how to use Relational Database Operator (RDO) queries to solve them.

The first problem is to find the job history and related job information for a specific employee.

You first look at the JOB_HISTORY relation and find that it has six fields:

- EMPLOYEE_ID (the employee's identification number)
- DEPARTMENT_CODE (the employee's department)
- JOB_CODE (an employee's job code)
- JOB_START (an employee's starting date)
- JOB_END (the date an employee ended the job)
- SUPERVISOR_ID (the supervisor's identification number)

If you wanted to know all of the jobs held by the employee whose identification number is 00164, you would need to identify the record in the JOB_HISTORY relation for EMPLOYEE_ID = "00164".

Examples

Example 1

```
FOR JH IN JOB_HISTORY WITH JH.EMPLOYEE_ID = "00164"
    PRINT
        JH.EMPLOYEE_ID,
        JH.DEPARTMENT_CODE,
        JH.JOB_START,
        JH.JOB_END,
        JH.JOB_CODE
END_FOR
```

But the data in each record of this relation does not tell you all you want to know about an employee's job. You may want to know the job title, the wage class, minimum salary, and maximum salary data from the JOBS relation. Because many employees can hold the same job, and the data about the job applies to them all, the desired information resides in a separate relation.

Consult the JOBS relation for this information using a JOB_CODE value ("MENG" for Mechanical Engineer) with the following query.

Example 2

```
FOR J IN JOBS WITH J.JOB_CODE = "MENG"
    PRINT
        J.WAGE_CLASS,
        J.JOB_TITLE,
        J.MINIMUM_SALARY,
        J.MAXIMUM_SALARY
END_FOR
```

You can combine both types of information with one query, displaying all the data as though it were one record. To do this, join a record from the JOB_HISTORY relation with a corresponding record from the JOBS relation. The CROSS clause of the record selection expression (RSE) enables you to "cross" or join records using a field common to both relations, JOB_CODE. The WITH clause specifies that only those records with a JOB_CODE value in one relation that matches a JOB_CODE value in the other relation are joined. Those records in one relation with JOB_CODE values that do not match JOB_CODE values in the other relation are excluded from the join. Because this clause joins related records from two relations, it is a **relational join.**

Example 3

```
FOR JH IN JOB_HISTORY
    CROSS J IN JOBS
    WITH JH.JOB_CODE = J.JOB_CODE
 PRINT
     JH.EMPLOYEE_ID,
     JH.DEPARTMENT_CODE,
     JH.JOB_CODE,
     J.WAGE_CLASS,
     J.JOB_TITLE,
     J.MINIMUM_SALARY,
     J.MAXIMUM_SALARY
END_FOR
```

When you join two or more relations in this way, you form an expanded output record containing data from several associated relations. Figure 4-1 illustrates the joining of two relations on a common field; in this case, JOB_CODE is the field common to both the JOBS and the JOB_HISTORY relations.

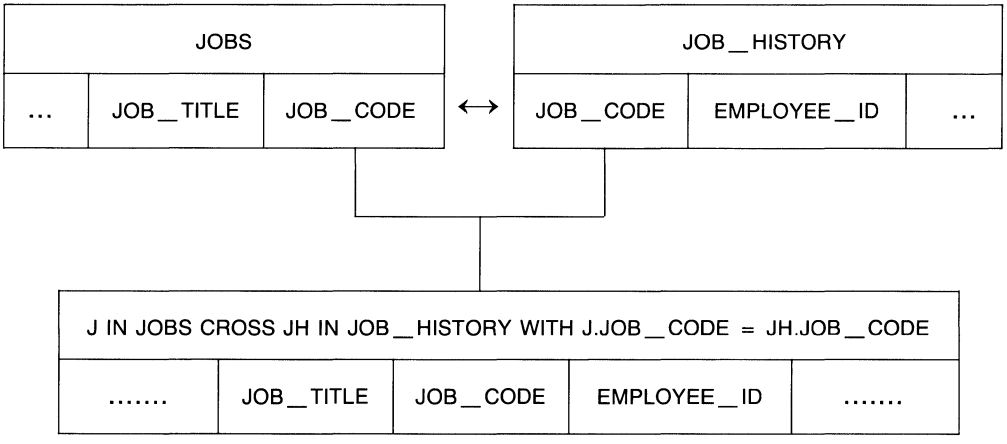


Figure 4-1: Joining Two Relations on a Common Key Field

You join two relations using a value of a field common to both relations. The field linking two relations may have the same name, or it may contain the same type of information but may have a different name. For example, the PERSONNEL database contains the fields: EMPLOYEE_ID, MANAGER_ID, and SUPERVISOR_ID. All of these fields contain employee identification numbers, but some identification numbers serve different purposes. However, any of these fields can serve as a join term, linking two relations together. As long as the join term contains logically identical data, you can use it to link two relations in a join.

You can use an OVER clause with the CROSS clause to specify the join term.

Example 4

```
FOR JH IN JOB_HISTORY
    CROSS J IN JOBS OVER JOB_CODE
 PRINT
     JH.EMPLOYEE_ID,
     JH.DEPARTMENT_CODE,
     JH.JOB_CODE,
     J.WAGE_CLASS,
     J.JOB_TITLE,
     J.MINIMUM_SALARY,
     J.MAXIMUM_SALARY
END_FOR
```

The following RSE uses CROSS and OVER clauses to join three relations. In many cases, using the OVER clause with a CROSS clause, as opposed to a clause using WITH (for example, WITH JH.JOB_CODE = J.JOB_CODE), results in reduced I/O overhead.

```
FOR E IN EMPLOYEES
   CROSS JH IN JOB_HISTORY OVER EMPLOYEE_ID
   CROSS J IN JOBS OVER JOB_CODE
```

In this RSE, Rdb/VMS joins the EMPLOYEES relation with the JOB_HISTORY relation using the EMPLOYEE_ID field to link the relations together. The second CROSS clause joins two relations, but you need to know which two relations contain the join term, JOB_CODE. Because JOB_HISTORY and JOBS both share the field, JOB_CODE, Rdb/VMS joins these two relations, not EMPLOYEES and JOBS. Each resulting virtual, or expanded, record has all the fields from JOBS and JOB_HISTORY.

You can use the OVER clause when you join two relations sharing a common field. More information about the OVER clause is contained in the section on creating queries in Chapter 5.

Using a WITH or OVER clause to qualify or limit a join lets you link related records from two relations. Although Rdb/VMS can process queries without the WITH or OVER clauses, the results are not very meaningful. You should include either clause in every join. If you do not specify a WITH or OVER clause, RDO joins each record of one relation with every record of the other relation, giving you a cross product. Such a join can be disastrous to your system's performance.

If you join the EMPLOYEES relation and the JOB_HISTORY relation without qualifying the relationship, Rdb/VMS joins every record in the EMPLOYEES relation with *every* record in the JOB_HISTORY relation. If there were 101 records in the EMPLOYEES relation and 277 records in the JOB_HISTORY relation, the cross product would contain 27,977 records.

If one relation contains unique key values for each record, you can easily join this relation with another relation that contains either similar unique key values for each record or multiple records with the same key value. Such a relationship is either one-to-one or one-to-many.

In the EMPLOYEES relation, EMPLOYEE_ID is a key field that contains a unique value for each record in the relation. The JOB_HISTORY and SALARY_HISTORY relations each contain many records belonging to an individual employee, because one employee can have many JOB_HISTORY records and many SALARY_HISTORY records. You can join the EMPLOYEES relation with the JOB_HISTORY relation to find all records belonging to a single employee.

You can also join the EMPLOYEES relation with the SALARY_HISTORY relation to retrieve all salary history records for that employee. The results of such joins are illustrated in Figure 4-2 on the next page; you assemble employee information with every JOB_HISTORY record or with every SALARY_HISTORY record.

EMPLOYEES

Employee_ID

SALARY_HISTORY

Employee_ID
Salary_Start
Salary_End
Salary_Amount

EMPLOYEES

Employee_ID

JOB_HISTORY

Employee_ID
Job_Start
Job_End
Job_Code
Dept_Code

ZK-00379-00

Figure 4-2: Joining Relations on a Key Field

However, joining the JOB_HISTORY relation with the SALARY_HISTORY relation presents a special situation. Each of these relations contains multiple records for an employee; such a relationship is a many-to-many relationship.

When you attempt a join with a many-to-many relationship, the cross product is likely to be meaningless. If you join JOB_HISTORY with SALARY_HISTORY using the join term, EMPLOYEE_ID, Rdb/VMS joins every record from JOB_HISTORY for an employee with *every* record from the SALARY_HISTORY relation. Thus every JOB_HISTORY record is associated with every SALARY_HISTORY record for a particular employee. But the relationship you need should be qualified further by checking date values in these records to link a JOB_HISTORY record with the salary history for *that particular job.*

To ensure that your join produces the correct results, include additional WITH clauses to qualify the join terms precisely. The following query contains a CROSS clause that joins the JOB_HISTORY relation with the SALARY_HISTORY relation and qualifies the join by including several WITH clauses.

Example 5

```
FOR JH IN JOB_HISTORY
  CROSS SH IN SALARY_HISTORY
   WITH JH.EMPLOYEE_ID = SH.EMPLOYEE_ID
    AND JH.JOB_END MISSING
    AND SH.SALARY_END MISSING
    AND JH.EMPLOYEE_ID = "00164"
        PRINT
                JH.EMPLOYEE_ID,
                JH.JOB_CODE,
                JH.JOB_START,
                SH.SALARY_START,
                SH.SALARY_AMOUNT,
                SH.SALARY_END
END_FOR

 00164    DMGR    21-SEP-1981  21-SEP-1981    50000.00  17-NOV-1858
```

The results of this query restrict the records retrieved from JOB_HISTORY and SALARY_HISTORY to those meeting the following qualifiers:

- EMPLOYEE_ID is 00164.
- Only records from both relations belonging to 00164 are retrieved.
- Only current records from the JOB_HISTORY and SALARY_HISTORY relations are needed; that is, the fields JOB_END and SALARY_END are missing because you are looking for employees who have not terminated their employment.

In some instances, it may be useful to add a condition that contains obvious information but is helpful to the Rdb/VMS optimizer. If you added the condition DEPT_CODE NOT MISSING to the above RSE, and DEPT_CODE were an indexed field, the optimizer would process the query more efficiently. Rather than go to the relation itself, which generates costs in I/O, the optimizer would go directly to the index table and look at cardinality. For more information about this process, see the *VAX Rdb/VMS Guide to Database Adminstration and Maintenance.*

4.1.2 Joining Records from More Than Two Relations

When you join two relations at a time, you need a CROSS clause for each pair of relations in the join. If you need to retrieve data from three relations, first join records from two relations on one field (join term). Then join one relation of the first pair with the third relation, using either the same join term or a different one.

Previously, you saw a query that joined two relations, JOB_HISTORY and JOBS, to retrieve all information about a specific job. But assume you need to find the full name of an employee, as well as his or her job history. To associate an employee with this type of information, you expand the query with another CROSS clause to access the EMPLOYEES relation and to supply an employee identification number.

```
FOR JH IN JOB_HISTORY
    CROSS J IN JOBS OVER JOB_CODE
    CROSS E IN EMPLOYEES OVER EMPLOYEE_ID
    WITH E.EMPLOYEE_ID = "00164"
PRINT
    JH.EMPLOYEE_ID,
    E.FIRST_NAME,
    E.LAST_NAME,
    J.JOB_TITLE,
    JH.DEPARTMENT_CODE,
    J.WAGE_CLASS
END_FOR
```

This query adds a second CROSS clause to get the information from the EMPLOYEES relation that you need. With this query, you access the following three relations:

- JOBS (data about each type of job)
- JOB_HISTORY (data about each job held by each employee)
- EMPLOYEES (personal data about each employee)

You join the JOB_HISTORY and JOBS relations on the JOB_CODE field to get complete job information. Then, to include the employee data associated with each set of job history records, you join the JOB_HISTORY relation with the EMPLOYEES relation. To ensure that job data corresponds to the correct employee, you perform this second join on the EMPLOYEE_ID field.

To join related records from three relations, Rdb/VMS:

- Forms a stream combining records from the JOBS and JOB_HISTORY relations that have matching values for JOB_CODE.
- Adds records from the EMPLOYEES relation that restrict values to that of EMPLOYEE_ID = "00164" for the records in the record stream of the first join. Now, the complete join operation includes only those records belonging to employee 00164.

Each resulting output record has fields from the three relations: JOBS, JOB_HISTORY, and EMPLOYEES. Each CROSS clause uses a join term common to two relations:

- JOB_CODE is the join term for the JOB_HISTORY and JOBS relations.
- EMPLOYEE_ID is the join term for the EMPLOYEES and JOB_HISTORY relations.

To improve the efficiency of complex joins, you can define an index for each frequently used join term. See the *VAX Rdb/VMS Guide to Database Design and Definition* for information on defining indexes.

When your query requires joins of two or more relations, you must include the names of the relations in your START_TRANSACTION statements. For example, the last query refers to three relations: EMPLOYEES, JOBS, and JOB_HISTORY. Your START_TRANSACTION statement might look like the following:

```
RDO> START_TRANSACTION READ_ONLY RESERVING
cont>                EMPLOYEES FOR SHARED READ,
cont>                JOBS FOR SHARED READ,
cont>                JOB_HISTORY FOR SHARED READ
```

4.1.3 Joining One Relation on Itself

Another type of query, called a **reflexive join**, allows you to join records from one relation with other records in the same relation. You treat the relation as if it were actually two relations, supplying two different context variables in the join. Perform a reflexive join when you wish to match values from fields of the same relation.

For example, you might want to list all job classifications of staff employees whose maximum salaries are greater than the minimum salaries of company executives. The JOBS relation contains information about jobs in all wage classes. Staff members are identified as wage class 2 and executives as wage class 4.

You could access the JOBS relation two separate times, first retrieving all maximum salaries for wage class 2, and then retrieving all minimum salaries for wage class 4. Finally, you compare them to find where they overlap. All of these steps are not necessary because Rdb/VMS lets you access the relation twice in the same query by means of a reflexive join.

The following query finds the salary ranges in different wage classes and locates staff salaries that are greater than the minimum executive salaries:

```
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
   PRINT
```

```
        STAFF.JOB_CODE,
        STAFF.MAXIMUM_SALARY,
        EXEC.JOB_CODE,
        EXEC.MINIMUM_SALARY
END_FOR
```

Note

When you use descriptive context variables like STAFF and EXEC, you are more likely to refer to the field names correctly. You know at a glance the stream to which you are referring.

The preceding RDO query joins the JOBS relation on itself. The query specifies two different context variables, STAFF and EXEC, for the same relation, EMPLOYEES. These statements instruct RDO to form a stream that includes records containing data on pairs of employees, STAFF and EXEC. For records of the STAFF and EXEC streams to be combined, they must meet two conditions:

- 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 the executive.

To process this query, Rdb/VMS:

- Takes the first record in STAFF (wage class = 2) and compares the maximum salary amount with the minimum salary amount of the first record in EXEC (wage class = 4).
- Compares the first record in STAFF with the next record of EXEC until all EXEC records have been compared. Rdb/VMS makes one pass through EXEC for each record in STAFF.
- Takes the second record in STAFF and compares it to the first record in EXEC.
- Compares the second record in STAFF with the second record of EXEC, and so on.
- Includes in the resulting record stream only those records that meet the specified conditions.

As Figure 4-3 illustrates, the JOBS relation appears as two relations: STAFF and EXEC. To distinguish the fields of one group from the other, this example uses STAFF and EXEC as the context variables.

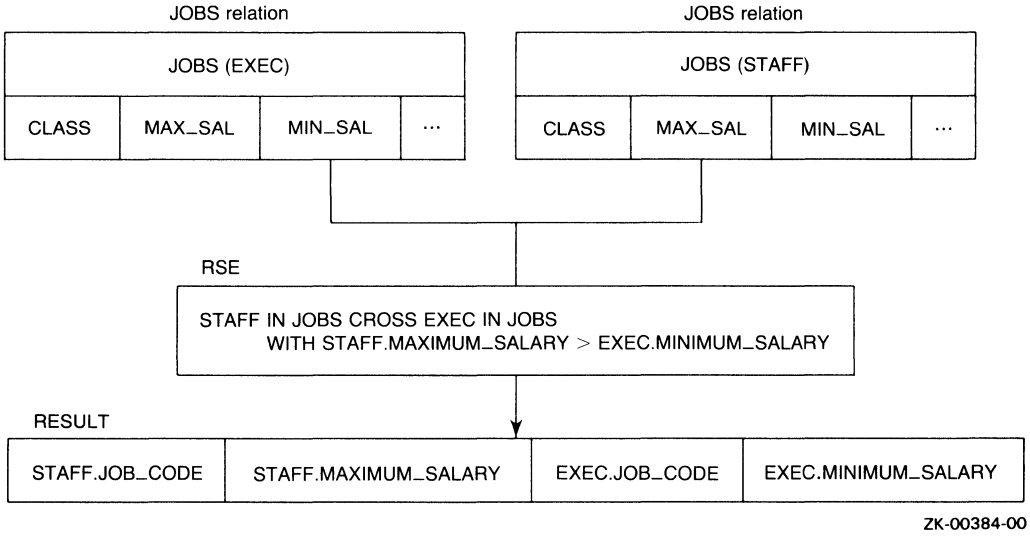


Figure 4-3: Joining a Relation on Itself (Reflexive Join)

4.2 Using Nested FOR Loops

When you use the CROSS clause to join two relations in a one-to-many relationship, Rdb/VMS links the record in the first relation to each record in the second relation. All values from the first relation are present in the resulting cross, but only the values common to both the first and second relation are included from the second relation.

Examples

Example 1

```
FOR E IN EMPLOYEES
  CROSS JH IN JOB_HISTORY
    WITH E.EMPLOYEE_ID = JH.EMPLOYEE_ID
    AND E.EMPLOYEE_ID = "00201"
      PRINT
       E.EMPLOYEE_ID,
       JH.JOB_CODE,
       JH.JOB_START,
       JH.JOB_END
END_FOR
```

```
 00201    APGM    15-APR-1979 00:00:00.00    27-MAY-1980 00:00:00.00
 00201    APGM    28-MAY-1980 00:00:00.00    17-NOV-1858 00:00:00.00
 00201    APGM     1-JUL-1975 00:00:00.00     3-JUN-1977 00:00:00.00
 00201    APGM     4-JUN-1977 00:00:00.00    14-APR-1979 00:00:00.00
```

You can suppress these repeating values by using nested FOR loops.

Example 2

```
FOR E IN EMPLOYEES
  WITH E.EMPLOYEE_ID = "00201"
    PRINT E.EMPLOYEE_ID

      FOR JH IN JOB_HISTORY
        WITH JH.EMPLOYEE_ID = E.EMPLOYEE_ID

         PRINT
          JH.JOB_CODE,
          JH.JOB_START,
          JH.JOB_END
    END_FOR

END_FOR
```

```
 00201
       APGM     1-JUL-1975 00:00:00.00     3-JUN-1977 00:00:00.00
       APGM    28-MAY-1980 00:00:00.00    17-NOV-1858 00:00:00.00
       APGM    15-APR-1979 00:00:00.00    27-MAY-1980 00:00:00.00
       APGM     4-JUN-1977 00:00:00.00    14-APR-1979 00:00:00.00
```

The process works as follows:

- RDO retrieves the first record in the stream formed by the outer loop and displays any expressions listed in the PRINT statement.
- RDO then processes the inner loop for each record specified by the inner loop's RSE.
- Control returns to the outer loop, and the cycle continues until there are no more records in the outer loop's stream.

Nesting FOR loops means entering one FOR statement (the outer loop) followed by a second FOR statement (the inner loop). The inner loop is part of the main FOR statement that controls it. Each loop has an RSE to bring the two record streams together in the same statement.

One date value that appears in the display, 17-NOV-1858, is the VMS base date used here to indicate a missing value for fields with a DATE data type.

With a nested FOR format, you can suppress the repeating values for EMPLOYEE_ID and display only the values from the JOB_HISTORY relation that change. In example 2, EMPLOYEE_ID 00201 appears only once, while the values from the records in the JOB_HISTORY relation are displayed for each separate record belonging to that employee. The results look like control breaks in routine reports. The nested FOR loop makes the resulting report more readable than reports that include duplicate employee identification numbers. Nested FOR loops are convenient for this type of display because you can show a one-to-many relationship: one EMPLOYEES record to many JOB_HISTORY records.

You can also use nested FOR loops to establish relationships for outer joins. See Chapter 7 for more information.

Using Views and View Definitions 5

This chapter introduces views and view definitions. A view is a query that you have named and stored. With a view, you can take a set of fields from a relation or a combination of fields from different relations that you use often and give them a name. Thus, you can treat those combinations as you would a relation, and use them in other record selection expressions (RSEs).

5.1 Using Views for Queries

After some experience using your database, you may discover that you often enter the same query and display the same field. Or you may enter the same record selection expression, but each time, you use different fields from that relation. You can take advantage of a feature in Rdb/VMS that allows you to make those queries permanent by using a view.

You may discover that you often use some fields in the EMPLOYEES relation more than you use others. You can define a view to create a more restricted version of the EMPLOYEES relation using fields from that relation. For example, you may find a view helpful if you frequently enter queries that use RDO to get the last name, first name, and employee identification number, as in example 1:

Examples

Example 1

```
FOR E IN EMPLOYEES
 PRINT
  E.LAST_NAME,
  E.FIRST_NAME,
  E.EMPLOYEE_ID,
END_FOR
```

Turn this query into a view by using the following view definition. Because you are defining a new database entity definition, you need to include write access to the database in your START_TRANSACTION statement.

Example 2

```
START_TRANSACTION READ_WRITE

DEFINE VIEW EMP_ID OF E IN EMPLOYEES.
  E.LAST_NAME.
  E.FIRST_NAME.
  E.EMPLOYEE_ID.
END VIEW.

COMMIT
```

Now you can refer to the view just as you refer to a relation, using the same field names as in the EMPLOYEES relation.

Example 3

```
FOR E IN EMP_ID
 PRINT E.*
END_FOR
```

You can also include an RSE when you refer to a view to restrict the records you display.

Example 4

```
FOR E IN EMP_ID WITH E.EMPLOYEE_ID = "00164"
 PRINT E.*
END_FOR
```

Example 4 displays four fields for the employee with EMPLOYEE_ID ="00164". If you forget which fields are included in the view definition, use the SHOW RELATIONS command. RDO displays all relations and indicates which definitions are views. For information on other parameters you can use with the SHOW statement, see the reference manual for Rdb/VMS.

```
RDO> SHOW RELATIONS

User Relations in Database with filename  personnel
     CANDIDATES
     COLLEGES
     CURRENT_INFO                        A view.
     CURRENT_JOB                         A view.
     CURRENT_SALARY                      A view.
     DEGREES
     DEPARTMENTS
     EMPLOYEES
     JOBS
     JOB_HISTORY
     RESUMES
     SALARY_HISTORY
     WORK_STATUS
```

Use the SHOW FIELDS FOR CURRENT_INFO to display the names of the fields in this view definition:

```
RDO> SHOW FIELDS FOR CURRENT_INFO

 Fields for relation  CURRENT_INFO
     LAST_NAME                          text size is  14
     FIRST_NAME                         text size is  10
     ID                                 text size is  5
       based on global field  ID_NUMBER
     DEPARTMENT                         text size is  30
       based on global field  DEPARTMENT_NAME
     JOB                                text size is  20
       based on global field  JOB_TITLE
     JSTART                             Date
       based on global field  STANDARD_DATE
     SSTART                             Date
       based on global field  STANDARD_DATE
     SALARY                             signed longword scale  -2
```

You can use views that contain selected records from a larger relation like the following view definition.

Example 5

```
DEFINE VIEW ACTIVE_EMP OF E IN EMPLOYEES
  WITH E.STATUS-CODE = "1".
   E.EMPLOYEE_ID.
   E.LAST_NAME.
   E.FIRST_NAME.
   E.MIDDLE_INITIAL.
   E.ADDRESS_DATA_1.
   E.ADDRESS_DATA_2.
   E.CITY.
   E.STATE.
   E.POSTAL_CODE.
   E.SEX.
   E.BIRTHDAY.
   E.STATUS_CODE.
END VIEW.
```

You can use the view in example 5 to refer to all full-time employees. The view contains every field defined for the EMPLOYEES relation.

You can also define a view that selects a few fields from all records in a relation. For example, to list only employee names and addresses for a mailing list, define the following view:

Example 6

```
DEFINE VIEW EMP_MAIL OF E IN EMPLOYEES.
   E.LAST_NAME.
   E.FIRST_NAME.
   E.MIDDLE_INITIAL.
   E.ADDRESS_DATA_1.
   E.ADDRESS_DATA_2.
   E.CITY.
   E.STATE.
   E.POSTAL_CODE.
END VIEW.
```

To see a list of the values in each field without naming every field, use the following PRINT format:

Example 7

```
FOR M IN EMP_MAIL
  SORTED BY M.LAST_NAME
    PRINT
     M.*
END_FOR
```

5.2 Creating a View Definition for a Join

The design of the PERSONNEL database includes ten relations, each shows a different aspect of the employees in a company. Other applications may require another view of the database, that combines fields from many relations. Rdb/VMS lets you create new relationships from the nine basic relations in the database, forming virtual relations. You do this by defining views using a join statement. Views have the following advantages:

- View definitions can prevent unauthorized users from accessing sensitive data by not including them in the view definition, while still allowing users to access the data they need. This is done by omitting the fields that contain sensitive information from the view definition.
- Queries that use complex selection criteria can be formalized in a view definition to make access easy.
- Updates can be made to views that are defined by using a single relation.
- Views let you assemble groups of fields from the original database relations.

As the previous examples show, joining relations sometimes can be quite complex. If you must often form the same RSE to retrieve records from several relations, you might consider creating a view definition. A view brings together fields from several relations based on an RSE specified in the view definition. A user can refer to the view definition as if it were a single relation and request RDO to display field values. Thus, a user who may not understand the syntax for a complex join can still access data from such a join when it is contained in a view.

To create a view definition that refers to a restricted record stream from three relations:

- Use the DEFINE VIEW statement that includes an RSE for specifying the record stream you wish to establish.
- Indicate which fields you want included from the record stream.

You can use the following view definition in place of a query. Use this view definiton just as you would a new relation that contains the fields you need.

Examples

Example 1

```
DEFINE VIEW EMP_HISTORY
   OF JH IN JOB_HISTORY
       CROSS J IN JOBS
       CROSS E IN EMPLOYEES
       WITH J.JOB_CODE = JH.JOB_CODE AND

       JH.EMPLOYEE_ID = E.EMPLOYEE_ID.
          JH.EMPLOYEE_ID.
          E.FIRST_NAME.
          E.LAST_NAME.
          J.JOB_TITLE.
          JH.DEPARTMENT_CODE.
          J.WAGE_CLASS.
END VIEW.
```

You can now use the view definition EMP_HISTORY with a record selection expression (RSE) to retrieve employee history data for a particular employee. However, you must include the name of the view in a START_TRANSACTION statement. Rdb/VMS implicitly reserves the referenced relations according to the reserving option you specify.

Example 2

```
START_TRANSACTION READ_ONLY RESERVING EMP_HISTORY FOR SHARED READ

FOR E IN EMP_HISTORY WITH E.EMPLOYEE_ID = "00164"
    PRINT E.*
END_FOR
```

To form a query that joins four relations, consider the following

You need to compile a report for each employee in the EMPLOYEES relation. The report should include:

- Each employee's identification number
- Each employee's first and last name
- The title of the job he or she currently holds
- The code of the department where the employee works
- Current salary information
- The wage class of the job the employee holds

The SALARY_HISTORY relation contains current salary information as well as the salary start date and salary amount for a particular job. The JOB_HISTORY relation holds data about each job an employee has held, including the department and job code. The JOBS relation contains information about each job in the company. The EMPLOYEES relation describes each employee in the company. Each of these relations supplies some data for the report. To get the necessary fields from each, you must join four relations.

Example 3

```
FOR SH IN SALARY_HISTORY                                    ⎫
        CROSS J IN JOBS                                     ⎬ ①
        CROSS E IN EMPLOYEES                                ⎭
        CROSS JH IN JOB_HISTORY
                WITH SH.SALARY_END MISSING                          ⎫
                AND SH.EMPLOYEE_ID = E.EMPLOYEE_ID                  ⎪
                AND JH.EMPLOYEE_ID = E.EMPLOYEE_ID                  ⎬ ②
                AND JH.JOB_END MISSING                              ⎪
                AND JH.JOB_CODE = J.JOB_CODE                        ⎭
        PRINT
                SH.EMPLOYEE_ID,                             ⎫
                E.FIRST_NAME,                               ⎪
                E.LAST_NAME,                                ⎪
                J.JOB_TITLE,                                ⎬ ③
                JH.DEPARTMENT_CODE,                         ⎪
                J.WAGE_CLASS,                               ⎪
                SH.SALARY_AMOUNT,                           ⎪
                SH.SALARY_START                             ⎭
END_FOR
```

To access the salary history data and the associated job and employee data, join the following four relations:

- JOB_HISTORY (each job ever held by each employee)
- JOBS (each type of job an employee can hold)
- EMPLOYEES (personal information on each employee)
- SALARY_HISTORY (each salary level held by an employee for each job held by each employee)

To join the four relations in the preceding query, use the CROSS clause three times. The following list refers to the numbered items in example 3:

1. Identify the four relations with enough CROSS clauses to specify those relations containing the fields you want to compare or display.
2. Add the WITH clauses to link each pair of relations sharing a common field or meeting a specific condition.

   One relation can be associated with more than one other relation:

   - JOB_HISTORY links with SALARY_HISTORY in the JOB_START field.
   - Records with JOB_END MISSING are current jobs. JOB_HISTORY also links with JOBS on the JOB_CODE field.
   - SALARY_HISTORY links with the EMPLOYEES relation on EMPLOYEE_ID but only for those records in SALARY_HISTORY whose SALARY_END date field contains the value MISSING for the job held currently.

     You can use any field in one relation that is common to any other relations. Each WITH clause links a field in one relation with a field in another and further restricts the records included in your record stream.

3. Display only those fields you need. Qualify each field with its context variable.

Using Value, Arithmetic and Statistical Expressions 6

A value expression is a string of symbols that specifies a value Rdb/VMS can use when executing statements. This chapter describes:

- Literals as value expressions
- Arithmetic expressions, such as (100 * SH.SALARY)
- Statistical expressions, such as AVERAGE SH.SALARY OF SH IN SALARY_HISTORY

6.1 Literals

A literal is either a character string or a numeric literal. You can use a literal as a value expression. For example:

```
FOR E IN EMPLOYEES WITH E.LAST_NAME = "Toliver"
   PRINT E.EMPLOYEE_ID
END_FOR
                                    character string literal


FOR S IN SALARY_HISTORY WITH S.SALARY_AMOUNT > 40000
   PRINT S.EMPLOYEE_ID
END_FOR
                                                numeric literal
```

6.1.1 Character String Literals

A character string literal is a string of printable characters that must be enclosed in quotation marks. The maximum length of a character string is 65,536 characters. The printable characters consist of:

- Uppercase alphabetic characters:

  A - Z

- Lowercase alphabetic characters:

  a - z

- Numerals:

  0 - 9

- Special characters:

  ! @ # $ % ^ & * ( ) - _ = + ` ~

  [ ] { } ; : ' " \ | / ? > < . ,

Use a pair of single or double quotation marks to enclose a character string literal. When using quotation marks, follow these rules:

- Begin and end a character string literal with the same type of quotation mark.
- To include a quotation mark of one type in a character string literal, enclose the literal in quotation marks of the other type. For example, to include double quotation marks in a character string literal, enclose the character string in single quotation marks.
- If a quotation mark appears in a character string literal enclosed by quotation marks of the same type, use two consecutive quotation marks for every one you want to include in the literal. This technique is necessary if you want to include quotation marks of both types in a single quoted string.

Table 6-1 shows how to use quotation marks in character string literals.

Table 6-1: Embedding Quotation Marks in Literal Expressions

Character String Value Expression	Value
"JONES"	JONES
'JONES'	JONES
"JONES'	[invalid]
""""""	''''
""""""	[invalid]
'My name is "Lefty".'	My name is "Lefty".
'My ''handle'' is "Lefty".'	My 'handle' is "Lefty".

Rdb/VMS usually treats uppercase and lowercase forms of the same letter as the same character. However, Rdb/VMS preserves the case distinction when comparing character strings. That is, NAME = "JONES" and NAME = "Jones" yield different results. See the previous chapter on specifying conditions for an explanation of case-sensitive expressions. Note that these rules apply only to RDO. For language specific rules, refer to the *VAX Rdb/VMS Guide to Programming*, or your programming language manual.

6.1.2 Numeric Literals

A numeric literal is a string of digits that Rdb/VMS interprets as a decimal number. A numeric literal may be:

- A decimal string consisting of digits and an optional decimal point. The maximum length, not counting the decimal point, is 19 digits.
- A decimal number in scientific notation (E-format), consisting of a decimal string mantissa and a signed integer exponent, separated by the letter E, or D for G_FLOATING.

Rdb/VMS allows great flexibility in numeric expressions. You can use unary plus and minus, any form of decimal notation, and embedded spaces in E notation. The following are valid numeric strings:

```
123
34.9
-123
.25
123.
0.33889909
6.03  E+23
6.03  E -23
```

If you use a numeric literal to assign a value to a field or a variable, the data type of the field or variable determines the maximum size value you can assign.

See Table 5-1 in the *VAX Rdb/VMS Reference Manual* for the maximum size allowed for a numeric literal assigned to each Rdb/VMS data type.

Because a period at the end of a data definition statement line terminates the line, do not use a decimal point to terminate a number if you want to include more data definition clauses in the statement. The period terminates the line in the following data definition statement:

```
COMPUTED BY X*2.
```

If you want to include more data definition clauses, include a zero after the decimal point, or place the value expression in parentheses:

```
COMPUTED BY X*2.0
COMPUTED BY (X*2.)
```

6.2 Arithmetic Expressions

An **arithmetic expression** combines value expressions and arithmetic operators. When you use an arithmetic expression in a statement, Rdb/VMS calculates the value associated with the expression and uses that value when executing the statement. Therefore, an arithmetic expression must be reducible to a value. However, Rdb/VMS does not permit arithmetic operations on fields defined with the DATE or text data type.

If either operand of an arithmetic expression is a null value, the resulting value is also null.

The arithmetic operators and their functions are:

```
+ Add
- Subtract
* Multiply
/ Divide
```

You do not have to use spaces to separate arithmetic operators from value expressions, except in one case: if an Rdb/VMS name precedes a minus sign, you must separate the name and the minus sign by a space.

You can use parentheses to control the order in which Rdb/VMS performs arithmetic operations. Rdb/VMS follows the normal rules of precedence. That is, it evaluates arithmetic expressions in the following order:

1. Value expressions in parentheses
2. Multiplication and division, from left to right
3. Addition and subtraction, from left to right

An arithmetic expression can be used in a data definition statement:

```
DEFINE VIEW DEDUCT OF
   E IN EMPLOYEES CROSS
   S IN SALARY_HISTORY OVER EMPLOYEE_ID WITH
   S.SALARY_
END MISSING.
      E.LAST_NAME.
      E.FIRST-NAME.
      AMOUNT COMPUTED BY
         ( ( S.SALARY_AMOUNT / 52 ) * 0.05 ).
END DEDUCT VIEW.

FOR FIRST 5 D IN DEDUCT PRINT D.* END_FOR
 LAST_NAME       FIRST_NAME    AMOUNT
 Toliver         Alvin         4.972307692307692E+001
 Smith           Terry         1.122692307692308E+001
 Dietrich        Rick          1.778557692307692E+001
 Kilpatrick      Janet         1.683653846153846E+001
 Nash            Norman        3.101346153846154E+001
```

This example defines a view that calculates a payroll deduction for health insurance:

- The record selection expression limits the records in the view to current salary records.
- The view fields include the employee's name and a weekly deduction field, calculated using an arithmetic expression from the annual salary for each employee (5% of the weekly salary).

In addition to using literal expressions, you can combine other value expressions in PRINT statements and definition statements to create new entities. You do this by creating value expressions and linking them with the concatenate operator (|) to form a concatenated expression.

A concatenated expression is a value expression that combines two other value expressions by joining the second to the end of the first. You can use the concatenate operator to concatenate expressions and combine value expressions of any kind, including numeric expressions, string expressions, and literals.

You can use concatenated expressions to simulate group fields and edit strings. The following examples show you how to use the concatenate operator.

You can use a concatenated expression as a field in a view:

```
DEFINE VIEW MAIL OF
 E IN EMPLOYEES.
  NAME COMPUTED BY
    E.FIRST_NAME | ' ' | E.MIDDLE_INITIAL | ' ' | E.LAST_NAME.
  E.ADDRESS_DATA_2.
  E.CITY.
  E.POSTAL_CODE.
END MAIL VIEW.
```

This statement creates a field for the view by combining the three name fields from EMPLOYEES.

A concatenated expression can simulate an edit string:

```
DEFINE VIEW DOLLARS OF S IN SALARY_HISTORY.
   S.EMPLOYEE_ID.
   S.SALARY_START.
   S.SALARY_END.
   SAL COMPUTED BY "$"|S.SALARY_AMOUNT.
END.
```

When you display the fields of DOLLARS, a dollar sign appears in front of the salary amount.

6.3 Statistical Expressions

Statistical expressions are a kind of value expression that calculate a value for all the records in a record stream. Statistical expressions are sometimes called aggregate expressions because they calculate a single value for a collection of records. When you use a statistical expression (with the exception of COUNT), you specify a value expression and an RSE. Rdb/VMS first evaluates the value expression for each record in the record stream formed by the RSE. Then it calculates a single value based on the results of the first step.

The following list describes queries and shows examples using statistical expressions in RDO statements:

- AVERAGE

  Print average salary from SALARY_HISTORY relation:

  ```
  PRINT AVERAGE SH.SALARY_AMOUNT OF SH IN SALARY_HISTORY
  ```

- COUNT

  Print number of employees from EMPLOYEES relation:

  ```
  PRINT COUNT OF E IN EMPLOYEES
  ```

- MAXIMUM

  Print highest value of salary amount from SALARY_HISTORY relation:

  ```
  PRINT MAX SH.SALARY_AMOUNT OF SH IN SALARY_HISTORY
  ```

- MINIMUM

  Print lowest value in the MINIMUM_SALARY field in the JOBS relation:

```
PRINT MIN J.MINIMUM_SALARY OF J IN JOBS
```

- TOTAL

  Display a total of all salaries for current mechanical engineers, where the JOB_CODE value equals "MENG":

```
PRINT TOTAL SH.SALARY_AMOUNT OF SH IN SALARY_HISTORY CROSS
    JH IN JOB_HISTORY OVER EMPLOYEE_ID WITH
    JH.JOB_CODE = "MENG" AND
    JH.JOB_END MISSING
```

While the COUNT expression includes missing values in the number of records, the TOTAL expression does not. Missing values are null fields and therefore cannot be included in a sum.

6.3.1 Statistical Expressions with Groups of Records

A statistical expression can operate on a group of records within a record stream. This operation is often called a global aggregate function because you can group records by a value in any relation in the database. For example, you can use the DEPARTMENT_CODE field in the DEPARTMENTS relation to group records in another relation, SALARY_HISTORY, in order to get the average salary for each department.

If you need to find the average salary for all employees in a corporation, you could use the RDO statements in the following example:

Examples

Example 1

```
PRINT AVERAGE SH.SALARY_AMOUNT OF SH IN SALARY_HISTORY WITH
    SH.SALARY_END MISSING
```

Here, the AVERAGE function operates on all the records in the record stream formed by the RSE. However, a statistical function also can operate on groups of records within the record stream.

To find out how many employees currently work in each department:

Example 2

```
FOR D IN DEPARTMENTS
   PRINT D.DEPARTMENT_NAME,
         COUNT OF JH IN JOB_HISTORY WITH
           JH.DEPARTMENT_CODE = D.DEPARTMENT_CODE AND
           JH.JOB_END MISSING
END_FOR
```

Here, instead of returning a value for each record in the record stream, a global aggregate function returns only a single value for the whole group.

Figure 6-1 illustrates how the global aggregate function works, using a simplified subset of the two relations.

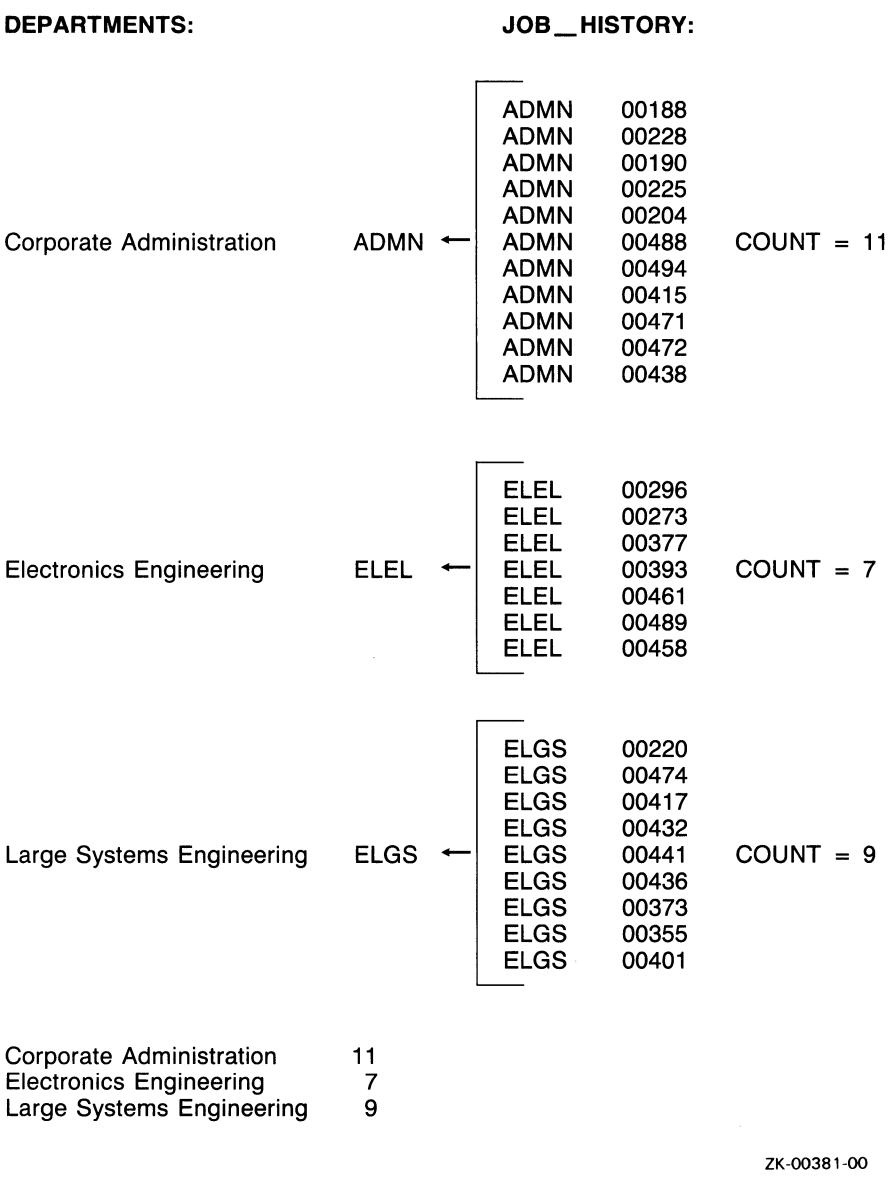


Figure 6-1: Using a Statistical Expression to Group Records

In example 2:

- FOR D IN DEPARTMENTS sets up an outer loop. The query performs a PRINT statement for each record in the DEPARTMENTS relation.
- The COUNT statistical expression is the object of the PRINT statement. The statistical expression includes its own looping function: it loops through the records specified by the RSE and counts them.
- The RSE links the inner loop, formed by the COUNT function, to the outer loop, formed by the FOR statement. The following steps take place:
  - Rdb/VMS finds the first record in DEPARTMENTS and reads the DEPARTMENT_CODE field. Assume that this value is ADMN.
  - Rdb/VMS finds the records in JOB_HISTORY where the DEPARMENT_CODE is ADMN. For each record in JOB_HISTORY where DEPARTMENT_CODE is ADMN, it adds one to the count.
  - When Rdb/VMS has found all the matches in JOB_HISTORY for ADMN, it displays the DEPARTMENT_NAME value that goes with ADMN and the count of JOB_HISTORY records.
  - Then Rdb/VMS returns to the outer loop of the query and finds the next DEPARTMENT_CODE value.

This type of query always has the following three features:

- The outer loop establishes the value by which to group records.
- The statistical expression creates the inner loop.
- The WITH clause binds the two together.

Note

The WITH clause in the inner loop must refer to the context variable established in the outer loop.

You can also use a conditional expression to place a restriction on the outer loop.

Example 3

The next example counts the number of employees in each job code group:

```
FOR J IN JOBS
PRINT J.JOB_TITLE,
      COUNT OF JH IN JOB_HISTORY WITH
         J.JOB_CODE = JH.JOB_CODE AND
         JH.JOB_END MISSING
END_FOR
```

Example 3 works in a similiar fashion to the previous example:

- FOR J IN JOBS forms the outer loop.
- The COUNT statistical expression forms the inner loop.
- The WITH clause links the inner loop with the outer loop.

Example 4

To find the total payroll for each department:

```
FOR D IN DEPARTMENTS
    PRINT D.DEPARTMENT_NAME,
        TOTAL SH.SALARY_AMOUNT OF
            SH IN SALARY_HISTORY CROSS
            JH IN JOB_HISTORY OVER EMPLOYEE_ID WITH
                JH.DEPARTMENT_CODE = D.DEPARTMENT_CODE AND
                JH.JOB_END MISSING AND
                SH.SALARY_END MISSING
END_FOR
```

Example 4 is complicated by having no direct link between the department information in DEPARTMENTS and the salary information in SALARY_HISTORY. The query uses JOB_HISTORY as the link because it contains the common fields, DEPARTMENT_CODE and EMPLOYEE_ID, and is interpreted as follows:

- FOR D IN DEPARTMENTS sets up the outer loop.
- The TOTAL statistical function starts the inner loop.
- In this case, the query requires an extra join to link the salary information in SALARY_HISTORY through JOB_HISTORY to DEPARTMENTS. SALARY_HISTORY and JOB_HISTORY share only one field: EMPLOYEE_ID.
- JH.DEPARTMENT_CODE = D.DEPARTMENT_CODE links the inner loop to the outer loop.

- The final two elements in the WITH clause limit the record stream to only those records that correspond to current SALARY_HISTORY and JOB_HISTORY records. *They are both necessary.* If you simply limit the record stream to current SALARY_HISTORY records, there is one current salary record for each JOB_HISTORY record with a matching EMPLOYEE_ID. The result looks correct but is actually inflated.

Figure 6-2 shows how the preceding query works with a subset of the data.

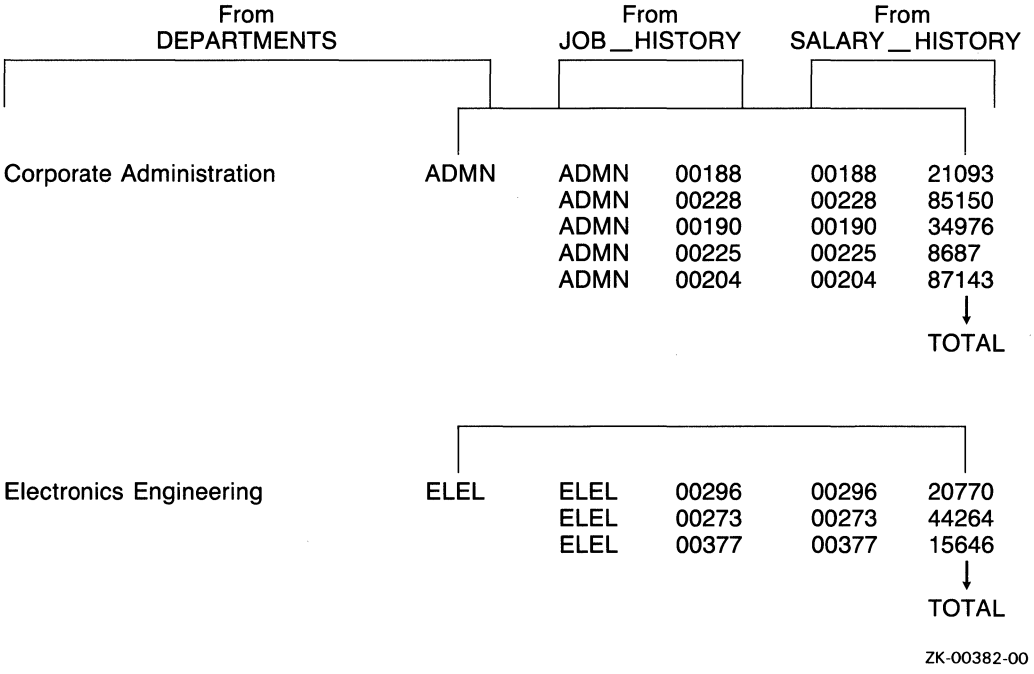


Figure 6-2: A Statistical Expression Across Three Relations

Example 5

In the next example, the PRINT statement includes two value expressions:

- The field D.DEPARTMENT_NAME
- The arithmetic expression, which includes two statistical expressions

Otherwise, this query is identical to previous examples in this chapter.

```
FOR D IN DEPARTMENTS
      PRINT D.DEPARTMENT_NAME, ((TOTAL SH.SALARY_AMOUNT OF
        SH IN SALARY_HISTORY CROSS
        JH IN JOB_HISTORY OVER EMPLOYEE_ID WITH
          JH.DEPARTMENT_CODE = D.DEPARTMENT_CODE AND
          JH.JOB_END MISSING AND
          SH.SALARY_END MISSING) / TOTAL SH.SALARY_AMOUNT OF
        SH IN SALARY_HISTORY WITH
          SH.SALARY_END MISSING) * 100
END_FOR
```

The preceding example is limited by the fact that SALARY_AMOUNT is defined as a longword integer. This means that the precision of the arithmetic expression is limited to whole numbers. If you include this query in a program, you can remove the calculation of the percentage from the Rdb/VMS query and use your program's host language calculation instead. For example, you can retrieve SALARY_AMOUNT, insert it into your program, and convert it to F_FLOATING before you calculate the percentage.

6.3.2 Arithmetic and Statistical Expressions Based on Field Values

```
AVERAGE   R.FIELD_1 OF R IN RELATION_A
COUNT               OF R IN RELATION_A
MAXIMUM   R.FIELD_1 OF R IN RELATION_A
MINIMUM   R.FIELD_1 OF R IN RELATION_A
TOTAL     R.FIELD_1 OF R IN RELATION_A
```

You can use arithmetic or statistical expressions with field names in the PRINT statement. For example, to find the average minimum salary of all jobs in a particular wage class, use the following query:

```
PRINT AVERAGE J.MINIMUM_SALARY OF J IN JOBS WITH
        J.WAGE_CLASS = "3"
```

In this example the statistical expression is:

```
AVERAGE J.MINIMUM_SALARY OF J IN JOBS
```

RDO checks all the records identified by the record selection expression, sums the minimum salaries, and divides by the number of records that satisfy the conditional expression WITH J.WAGE_CLASS = "3" to give the average value of MINIMUM_SALARY. The definition of the MINIMUM_SALARY field allows the value to be missing. That is, the field does not have to contain a value. When Rdb/VMS calculates the average minimum salary, it ignores records where MINIMUM_SALARY is missing.

Updating Databases 7

This chapter shows you how to use RDO to store, modify, and erase data in an Rdb/VMS database. After you become familiar with the statements that perform these operations, you can include them in your host language programs. See Appendix B for some examples of this procedure using the EDT editor in RDO. Other examples can be found in the *VAX Rdb/VMS Guide to Programming.*

7.1 Storing Data in an Rdb/VMS Database

After you define your database and the entity definitions, you can store data in each relation using one of the following methods:

- Use the RDO STORE statement.
- Embed your store operations in a host language program to load data into your database interactively or from disk files.
- Use VAX DATATRIEVE to load data from VAX DATATRIEVE files or a VAX DBMS database.

7.1.1 Storing Values in One Relation

Entering and maintaining data in a database is an ongoing task. For example, every time a new employee joins a particular company, the Personnel department adds a record to a relation like the EMPLOYEES relation described throughout this guide.

Adding a new record to a relation in the database does not present a record-level conflict to other users of the database because they cannot access a record that does not yet exist. The START_TRANSACTION statement that precedes the STORE statement should specify SHARED share mode and WRITE lock type:

```
RDO> START_TRANSACTION READ_WRITE RESERVING
cont>         EMPLOYEES FOR SHARED WRITE
```

Setting the share mode to SHARED ensures that you are able to store several records in a session while other users access the same relation. However, other users can specify transaction modes in their START_TRANSACTION statements that conflict with your intentions. When such conflicts occur, RDO might not allow other users access to the relation until you terminate your transaction. Refer to Chapter 2 for details on access conflicts.

When an employee joins the company, you should have enough information to store values for each field in a record of the EMPLOYEES relation. In Rdb/VMS, you insert a record into a relation with the STORE statement. This statement usually includes a series of assignments that specify the values for each field of the record.

```
START_TRANSACTION READ_WRITE RESERVING EMPLOYEES FOR SHARED WRITE

STORE E IN EMPLOYEES USING
      E.EMPLOYEE_ID = "00502";
      E.FIRST_NAME = "Paul";
      E.LAST_NAME = "Chris";
      E.CITY = "Boston"
END_STORE

COMMIT
```

7.1.2 Storing Values in Multiple Relations

You can store values in more than one relation within the same FOR statement in RDO. However, you need a STORE statement for each relation. For example, after you have entered all the records for new employees in the EMPLOYEES relation, you may want to add corresponding employee records in the JOB_HISTORY and SALARY_HISTORY relations. No field in the EMPLOYEES relation distinguishes a new employee record from existing employee records. You can, however, use the EMPLOYEES relation to compare existing EMPLOYEE_ID values with those already stored in the other two relations. The JOB_HISTORY and SALARY_HISTORY relations should have at least one corresponding record for every employee currently working in the company. Therefore, those records in the EMPLOYEES relation with no records in the JOB_HISTORY or SALARY_HISTORY relations must be newly hired employees.

You can perform a join across these three relations to check for the existence of corresponding records. If there are no matching records in the history relations, you can store new records in the history relations using values from the EMPLOYEES relation.

Before you can store new values in the database, you must specify the necessary relations in your START_TRANSACTION statement and indicate the correct share mode and lock type. Notice you need write access for JOB_HISTORY and SALARY_HISTORY, but only read access to the EMPLOYEES relation.

```
START_TRANSACTION READ_WRITE RESERVING
        EMPLOYEES FOR SHARED READ,
        JOB_HISTORY FOR SHARED WRITE,
        SALARY_HISTORY FOR SHARED WRITE
```

Using the NOT ANY operator here allows you to check that no corresponding records exist in the history relations. RDO executes the STORE statement only when there is no such correspondence. Example 1 uses a nested FOR construction to check for correspondence and to store data in the JOB_HISTORY and SALARY_HISTORY relations. It also prints each EMPLOYEE_ID value that represents each new employee.

Examples

Example 1

```
FOR E IN EMPLOYEES
 WITH NOT ANY JX IN JOB_HISTORY
 WITH JX.EMPLOYEE_ID = E.EMPLOYEE_ID
  PRINT E.EMPLOYEE_ID

    FOR EX IN EMPLOYEES
     WITH NOT ANY SX IN SALARY_HISTORY
     WITH SX.EMPLOYEE_ID = EX.EMPLOYEE_ID

       STORE JH IN JOB_HISTORY
        USING
          JH.EMPLOYEE_ID = EX.EMPLOYEE_ID;
          JH.JOB_START = "19-APR-1985"
       END_STORE

       STORE SH IN SALARY_HISTORY
        USING
          SH.EMPLOYEE_ID = EX.EMPLOYEE_ID;
          SH.SALARY_START = "19-APR-1985"
       END_STORE

    END_FOR
END_FOR

COMMIT
```

Example 2 performs the same store operation as example 1, using one FOR statement and a more complex WITH clause:

Example 2

```
FOR E IN EMPLOYEES
  WITH
    (NOT ANY JX IN JOB_HISTORY
      WITH JX.EMPLOYEE_ID = E.EMPLOYEE_ID)
  AND
    (NOT ANY SX IN SALARY_HISTORY
      WITH SX.EMPLOYEE_ID = E.EMPLOYEE_ID)
```

```
        STORE JH IN JOB_HISTORY USING
          JH.EMPLOYEE_ID = E.EMPLOYEE_ID;
          JH.JOB_START = "19-APR-1985"
        END-STORE

        STORE SH IN SALARY_HISTORY USING
          SH.EMPLOYEE_ID = E.EMPLOYEE_ID;
          SH.SALARY_START = "19-APR-1985"
        END-STORE

END_FOR
```

Since you can include STORE statements like these in a single transaction, it is good practice to keep related updates together in one transaction.

7.2 Using RDB$MISSING for Missing Values

Sometimes you may not have information for every field when you are storing information for a record. When a field in a record is left blank, Rdb/VMS automatically marks the field as missing. You have the option to define a missing value for a field in its field definition. If you do not define a missing value and the field is left blank, Rdb/VMS supplies default missing values to the field in the form of zeros for numeric fields and spaces for text fields.

When a field is defined as missing, Rdb/VMS marks the field and returns the defined missing value when you include the field in display statements. You can think of a missing field as empty. Retrieve missing fields by using the MISSING relational operator, or by using the PRINT statement with an asterisk (such as with PRINT E.*) to print all the field values including the missing value.

To define a missing value for the field MIDDLE_INITIAL, use the following definition:

```
DEFINE FIELD  MIDDLE_INITIAL
        DESCRIPTION IS /* Employee's middle initial */
        DATATYPE IS TEXT SIZE IS 1
        MISSING_VALUE IS " ".
```

7.2.1 Retrieving Records with a Missing Field Value

```
FOR R IN RELATION_A WITH R.FIELD_1 MISSING
```

When you do not explicitly store a value in a field, the value for that field is considered missing. You can retrieve records in which a field value is missing using the MISSING relational operator. You can also define a MISSING VALUE for any field. When you retrieve a missing field, Rdb/VMS returns the missing value as though it were actually stored in the field.

To display any employee records whose birth date is missing, use the following:

```
FOR E IN EMPLOYEES WITH E.BIRTHDAY MISSING
  PRINT
    E.EMPLOYEE_ID,
    E.LAST_NAME,
    E.FIRST_NAME,
    E.STATUS_CODE
END_FOR
```

Rdb/VMS finds all the records containing fields where no birth date is assigned and prints a space for BIRTHDAY in those records.

Note

You can use the MISSING operator to retrieve a segmented string. When relational operators are used, Rdb/VMS does not include fields whose values are missing in the record stream. Therefore, when using relational operators to make comparisons of fields in the database, be sure to write the RSE clearly to include records whose values are missing. For example, if MIDDLE_INITIAL is missing in some records, you could use one of the following RSEs to find those records:

```
FOR E IN EMPLOYEES WITH E.MIDDLE_INITIAL > 'M'
```

or

```
FOR E IN EMPLOYEES WITH E.MIDDLE_INITIAL MISSING
```

If a field has a specific missing value defined for it, the missing value is included in the field values displayed. Missing values are assigned the highest value in the ASCII collating sequence. Records sorted by a field with a defined missing value appear first when the sort order is DESCENDING and last when the sort order is ASCENDING.

If the field has no missing value in its definition, and the value is blank, Rdb/VMS assigns either spaces to a text field or zeros to a numeric field or segmented string. Moreover, if the MIDDLE_INITIAL field in the EMPLOYEES relation, for example, has no missing value defined for it, you cannot locate records using the MISSING relational operator for this field. The following example displays all of the stored values for MIDDLE_INITIAL using the GE (greater than or equal to) relational operator.

```
FOR E IN EMPLOYEES
  WITH E.MIDDLE_INITIAL GE " "
   SORTED BY ASCENDING E.MIDDLE_INITIAL
   PRINT
    E.MIDDLE_INITIAL
END_FOR
```

Because spaces precede alphabetic characters in the ASCII collating sequence, all records with spaces in the MIDDLE_INITIAL field are displayed first, followed by valid middle initials from A to Z.

7.2.1.1 Using Nested FOR Loops, Outer Joins and the MISSING Clause

You can use nested FOR loops to establish relationships for outer joins. In a common type of join, such as an equijoin, Rdb/VMS matches certain values in a field from one relation with a corresponding field 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.

Note

To allow Rdb/VMS to optimize queries, use nested FOR loops only when you want to reference more than one database or to perform outer joins.

To accomplish an outer join using Rdb/VMS, you must include an RDB$MISSING clause in the RSE so the unmatched values are added at the end of the join. The RDB$MISSING clause denotes the value of a field has been defined as absent. The following example shows how you use the RDB$MISSING clause in a nested FOR loop to find all employees and show what degrees they have. The employees' last names are sorted in alphabetic order.

```
FOR E IN EMPLOYEES SORTED BY E.LAST_NAME
    FOR D IN DEGREES WITH D.EMPLOYEE_ID = E.EMPLOYEE_ID
        PRINT
           E.LAST_NAME,
           E.FIRST_NAME,
           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

         PRINT
          E.LAST_NAME,
          E.FIRST_NAME,
          RDB$MISSING(D.DEGREE),
          RDB$MISSING(D.DEGREE_FIELD)

    END_FOR

END_FOR
```

This query prints information for all employees. If they have degrees, it prints each degree they have. If an employee has no degrees, the missing value for the degree field is printed (in this case, the value is the word UNKNOWN). Because the outer FOR loop sorts the employees by last name, all employees without degrees are included along with the employees that have degrees.

7.2.2 Storing Missing Values

When a field has a defined missing value, you can store a missing value with the STORE statement or you can store a new record and not supply a value for the field you want to have a missing value. The following example shows how to use the STORE statement to store a missing value in the E.MIDDLE_INITIAL field:

```
STORE E IN EMPLOYEES
  USING
     E.MIDDLE_INITIAL = RDB$MISSING(E.MIDDLE_INITIAL)
END_STORE
```

7.2.3 Storing Segmented Strings

You can use the Rdb/VMS segmented string data type to store large blocks of data in a database. Using the segmented string data type, you can store unstructured data such as text, graphics, voice, telemetry, or bit streams. Any data type can be stored in and retrieved from a segmented string. The data is stored in unstructured bytes; you could store character data in a segmented string and then retrieve it as hexadecimal data.

The segmented string is stored as a field in a relation. In fact, what you actually store is a pointer to the segmented string, called a segmented string handle, in the field that "contains" the segmented string. Because you are storing a pointer to the segmented string, rather than the string itself, the segmented string is not constrained by the Rdb/VMS record size limit. See the *VAX Rdb/VMS Guide to Programming* for more information on segmented strings.

The following is an example of storing a segmented string:

```
START_TRANSACTION READ_WRITE RESERVING RESUMES
                             FOR SHARED WRITE

        CREATE_SEGMENTED_STRING RESUME_HANDLE
           STORE SEG IN RESUME_HANDLE USING
              SEG.RDB$VALUE = "This is the first segment"
           END_STORE

           STORE SEG IN RESUME_HANDLE USING
              SEG.RDB$VALUE = "This is the second segment"
           END_STORE


          STORE R IN RESUMES USING
```

```
        R.EMPLOYEE_ID= "00164";
        R.RESUME = RESUME_HANDLE
    END_STORE
    END_SEGMENTED_STRING RESUME_HANDLE
COMMIT
```

7.3 Modifying Values

When you update data in a relation, you first must identify the record stream containing the record or records you want to change. You can assign new values, use existing values in fields from other relations, or specify value expressions to calculate the new value for each field in the selected records of the record stream.

In a single transaction, you can modify data in more than one relation. If the changes to records in these relations depend on the values of fields in another relation (such as the EMPLOYEE relation), you must specify the record stream containing the records of the other relation (in this case, EMPLOYEE).

When you are modifying relations, you can choose to let other users access those same relations. But once you select a record to change, Rdb/VMS locks that record. Other users cannot access that record until you release it. The lock ensures that only current data is available to all users. The changes you make are available to other users only after you make your changes permanent with the COMMIT statement. If you decide that the changes you make should not apply to the database at this time, you can undo the updates with the ROLLBACK statement.

7.3.1 Using One Relation to Modify Values

You can select records in the relation you want to modify, by using only the relation itself. After you create a record stream, Rdb/VMS executes the MODIFY statement to change the values of the specified field or fields for every record in the stream.

The next example uses an RSE that identifies all people currently employed in the Engineering department. It finds the records of all current employees in the JOB_HISTORY relation using the JOB_END MISSING expression. When the records are selected, the RSE uses the MODIFY statement to change all the supervisor identification numbers in Engineering to 00348.

Examples

Example 1

```
START_TRANSACTION READ_WRITE RESERVING JOB_HISTORY
        FOR SHARED WRITE

FOR JH IN JOB_HISTORY
   WITH JH.DEPARTMENT_CODE = "ENG "
   AND JH.JOB_END MISSING
```

```
      MODIFY JH USING
        JH.SUPERVISOR_ID = "00348"
      END_MODIFY

END_FOR
COMMIT
```

Example 2

The following example verifies the changes just made to the JOB_HISTORY relation.

```
FOR JH IN JOB_HISTORY
   WITH JH.DEPARTMENT_CODE = "ENG"
   AND JH.JOB_END MISSING
      PRINT
        JH.EMPLOYEE_ID,
        JH.DEPARTMENT_CODE,
        JH.SUPERVISOR_ID
END_FOR
```

You can modify values in a field by including a value expression. Any valid arithmetic operation can be used in your value expression to assign new values to a field or fields. You can also refer to values from other fields within the same relation.

For example, if you define a new field in the SALARY_HISTORY relation called WEEKLY, you can assign values to that field for every current record in the SALARY_HISTORY relation. The current record in the SALARY_HISTORY relation, like the current record in the JOB_HISTORY relation, is identified by the MISSING relational operator.

Use the MODIFY statement to modify the new WEEKLY field and replace missing values in the WEEKLY field with new values.

Example 3

```
START_TRANSACTION READ_WRITE RESERVING
             SALARY_HISTORY FOR SHARED WRITE

FOR SH IN SALARY_HISTORY
  WITH SH.SALARY_END MISSING
    MODIFY SH
       USING
          SH.WEEKLY = (SH.SALARY_AMOUNT/52)
    END_MODIFY
END_FOR
```

The MODIFY statement in example 3 includes a reference to another field, SALARY_AMOUNT, in the same relation and uses it in an arithmetic operation to compute a weekly salary amount for every current employee in the SALARY_HISTORY relation.

7.3.2 Using More Than One Relation to Modify Values

This section shows you how to modify the data in two relations. The transaction updates every record in the record stream identified by the RSE in the FOR statement. Each MODIFY statement operates on a separate relation.

Assume that because information was unavailable, records in JOB_HISTORY and SALARY_HISTORY were stored with missing values for job code, department code, and salary amount. When the information becomes known, you can locate these records and supply valid data, changing their definitions from MISSING to actual values.

Here are the steps to modify the fields in the JOB_HISTORY and SALARY_HISTORY relations:

1. Form a record selection expression that selects employee records from EMPLOYEES, the current SALARY_HISTORY records, and the current JOB_HISTORY records.
2. Change the field or fields of the SALARY_HISTORY record with a MODIFY statement.
3. Modify the fields of the JOB_HISTORY record.
4. Commit the changes to the database.

You specify three relations, EMPLOYEES, JOB_HISTORY, and SALARY_HISTORY and link them with the CROSS clause. The complete RSE identifies the current record from each of the relations.

Note

Use caution when you modify records in a relation and refer to more than one relation in the RSE. If you attempt to modify a join term in the record selection expression, Rdb/VMS cannot ensure predictable results. Changing such a value might change the contents of the record stream. Therefore, do *not* attempt to modify any field used in your record selection expression. If your changes to one database are committed, there is no way to roll them back. If you use this method of modifying values, your application must be useable if changes are made to one database but not to the others specified in your START_TRANSACTION statement.

The following command file changes values in two relations. Note that none of the fields in the record selection expression is modified.

```
START_TRANSACTION READ_WRITE RESERVING
      EMPLOYEES FOR SHARED READ,
      JOB_HISTORY FOR SHARED WRITE,      ! Start transaction with
      SALARY_HISTORY FOR SHARED WRITE    ! write access


FOR E IN EMPLOYEES WITH                  ! Begin outer loop
     E.EMPLOYEE_ID GE "00502"            ! selecting new employees

 FOR JH IN JOB_HISTORY                   ! Begin first inner loop
   WITH E.EMPLOYEE_ID = JH.EMPLOYEE_ID   ! selecting their current
   AND JH.JOB_END MISSING                ! job history records

      MODIFY JH USING                    ! Modify two fields
          JH.JOB_CODE = "MENG";
          JH.DEPARTMENT_CODE = "ENG ";
      END_MODIFY
 END_FOR                                 ! Terminate first inner loop

 FOR SH IN SALARY_HISTORY                ! Begin second inner loop
    WITH SH.EMPLOYEE_ID = E.EMPLOYEE_ID  ! selecting their current
    AND SH.SALARY_END MISSING            ! salary history records

      MODIFY SH USING                    ! Modify salary amount
          SH.SALARY_AMOUNT =
            (SH.SALARY_AMOUNT * 1.1)
      END_MODIFY

 END_FOR                                 ! Terminate second inner loop
END_FOR                                  ! Terminate outer loop

COMMIT
```

Some update tasks require features, such as extensive arithmetic functions and character manipulation, that a host language program can provide. RDO performs limited queries of the database. It helps you to build logically and syntactically correct statements that you can then embed in a host language program. For extensive and complex interactive updates or reports, you can use VAX DATATRIEVE.

7.3.3 Modifying Missing Values

If a field has a value stored in it from a previous STORE statement, you can change that value to a missing value with the MODIFY statement:

```
FOR E IN EMPLOYEES WITH E.MIDDLE_INITIAL MATCHING "%"
MODIFY E USING E.MIDDLE_INITIAL= RDB$MISSING(E.MIDDLE_INITIAL)
END_MODIFY
END_FOR
```

7.4 Updating with the START_STREAM Statement

Unlike the FOR statement, which works with every record identified in the RSE until all records are processed, the START_STREAM statement makes records available but does not automatically retrieve any record in a record stream. In order to see the records in a particular record stream, you must retrieve each one with the FETCH statement. After FETCHing a record, you can test values of fields and process some records while passing over others. The START_STREAM statement is especially useful for conditionally processing the records in a record stream in programs.

The next example performs an update to the JOB_HISTORY and SALARY_HISTORY relations similar to the STORE operation you read about earlier. Follow these steps for the update procedure:

1. Identify and name a group of selected records from EMPLOYEES (new employees) using the START_STREAM statement and the new EMPLOYEE_ID values.
2. Locate the record for the first new employee record using the FETCH statement.
3. For each new employee record, add a record in the JOB_HISTORY and the SALARY_HISTORY relations.
4. Repeat steps two and three until there are no more records in the record stream.
5. Close the stream.
6. Commit the transaction to the database.

You create and name a record stream with the START_STREAM statement. The collection of records included in the stream is identified by its record selection expression.

```
START_STREAM NEW_STAFF
      USING E IN EMPLOYEES WITH
         E.EMPLOYEE_ID > "00501"
```

The previous statement identifies the records you want to test. To locate a record for use, you issue a FETCH statement that finds the first record from the record stream NEW_STAFF.

```
FETCH NEW_STAFF
```

You can now use any other RDO statements to manipulate the values in the record during your update operation. Because no record exists in either history relation for the new employees, you use the STORE statement to add a record in both relations for each new employee.

Examples

Example 1

```
START_TRANSACTION READ_WRITE RESERVING
           JOB_HISTORY FOR SHARED WRITE,
           SALARY_HISTORY FOR SHARED WRITE,
           EMPLOYEES FOR SHARED READ

START_STREAM NEW_STAFF USING
   E IN EMPLOYEES WITH
   E.EMPLOYEE_ID = "00502"


     FETCH NEW_STAFF

       STORE JH IN JOB_HISTORY USING
         JH.EMPLOYEE_ID = E.EMPLOYEE_ID;
         JH.JOB_START = "01-APR-1985"
       END_STORE

       STORE SH IN SALARY_HISTORY USING
         SH.EMPLOYEE_ID = E.EMPLOYEE_ID;
         SH.SALARY_START = "01-APR-1985"
       END_STORE


END_STREAM NEW_STAFF

COMMIT
```

The preceding statements add only those records belonging to the first new employee to the JOB_HISTORY and SALARY_HISTORY relations. But there are six new employee records to process.

To solve the problem of processing all six records in the stream interactively, you could repeat the FETCH and STORE statements six times. Clearly, using RDO interactively to perform these updates can be time-consuming. Remember that when you need to do the same operation several times, you can enter the statements in a command file using your VMS text editor. Then, you can type the name of the command file preceded by the at sign (@) and have RDO execute the statements in the command file in the order you entered them.

To complete the six update operations with the least effort, break down the operation into three steps:

1. Form the record stream with the START_STREAM statement.
2. Execute the update procedure the required number of times.
3. End the stream and commit the transactions to the database.

Rdb/VMS considers any line or portion of a line preceded by an exclamation point to be a comment and ignores any text after it. The following example contains comments to explain each statement. You can include a comment in any RDO statement, either on a separate line or on the same line following an RDO statement. The statements that appear in UPDATE.RDO follow.

Example 2

```
FETCH NEW_STAFF

  STORE JH IN JOB_HISTORY USING
    JH.EMPLOYEE_ID = E.EMPLOYEE_ID;
    JH.JOB_START = "01-APR-1985"
END_STORE

STORE SH IN SALARY_HISTORY USING
    SH.EMPLOYEE_ID = E.EMPLOYEE_ID;
    SH.SALARY_START = "01-APR-1985"
END_STORE
```

The update process using UPDATE.RDO as the command file appears in the next example.

Example 3

```
RDO> PRINT COUNT OF E IN EMPLOYEES WITH
cont>      E.EMPLOYEE_ID > "00502"
  6                                ! Stream has six records


RDO> START_STREAM NEW_STAFF USING  ! Start stream NEW_STAFF
cont>    E IN EMPLOYEES WITH
cont>    E.EMPLOYEE_ID GE "00502"
RDO> @UPDATE                       ! Update - repeat six times
RDO> @UPDATE
RDO> @UPDATE
RDO> @UPDATE
RDO> @UPDATE
RDO> @UPDATE
RDO> END_STREAM NEW_STAFF          ! Close the stream
RDO> COMMIT                        ! Write changes to database
```

When a query like UPDATE.RDO attempts to execute again and there are no more records in the record stream, Rdb/VMS responds with this message:

```
RDO> @UPDATE
%RDB-E-STREAM_EOF, attempt to fetch past end of stream.
```

Remember to terminate your transactions with either the COMMIT or the ROLLBACK statement. Table 7-1 illustrates the effects of these statements on databases and transactions.

Table 7-1: Effects of COMMIT and ROLLBACK on Databases and Transactions

Items	COMMIT	ROLLBACK
Scope of Statement	Includes all invoked databases	Includes all invoked databases
Database	Writes to disk all changes to a database and its data definitions	Does not write to disk changes to a database and its data definitions
Open Streams	Closes all open streams as in STREAM_END	Closes all open streams as in STREAM_END
Position in Stream	No record is available	No record is available
Record Locks	Releases all locks	Releases all locks

7.4.1 Using a View to Modify a Database

You can use a view to modify data. But if you are using a view definition that refers to more than one relation, there is a restriction. If you define a view using a CROSS or REDUCED TO clause, you *cannot* update your database with that view. This restriction is necessary because the effects of modifying some records in one relation and other records in a second relation can be unpredictable. Fields selected and modified in the virtual record may change relationships in other records.

For example, you define a view using the JOB_HISTORY and DEPARTMENTS relations with a CROSS clause.

Examples

Example 1

```
DEFINE VIEW WORKER_DEPARTMENTS
  OF JH IN JOB_HISTORY
      CROSS D IN DEPARTMENTS OVER DEPARTMENT_CODE.
     ID         FROM JH.EMPLOYEE_ID.
     J_CODE     FROM JH.JOB_CODE.
     DEPT_CODE  FROM JH.DEPARTMENT_CODE.
     DEPT       FROM D.DEPARTMENT_NAME.
END WORKER_DEPARTMENTS VIEW.
```

Suppose you want to change a department code for an employee in a specific department. If you use this view to retrieve those records, your query looks like example 2.

Example 2

```
FOR W IN WORKER_DEPARTMENTS
 WITH W.DEPT_CODE = "ENG"
 AND W.ID = "00201"
   PRINT
     W.ID,
     W.J_CODE,
     W.DEPT_CODE,
     W.DEPT
END_FOR
```

Each record listed in the display contains some unique fields from the JOB_HISTORY relation and common fields, DEPARTMENT_CODE and DEPARTMENT_NAME, from the DEPARTMENTS relation. Many employees in the company work in the same department. If you try to use this view to change the department code, your update would fail because you cannot update data in a view derived from more than one relation.

Use a view that refers to records in only one relation to find and update records in the record stream and ensure data integrity.

7.5 Erasing Data in a Relation

This section shows you how to delete one or many records in a relation identified by the RSE using the ERASE statement. Because you are updating the database, you begin the update transaction with the START_TRANSACTION statement and in the READ_WRITE transaction mode. When you terminate the transaction, use the COMMIT statement to make the deletions permanent or enter the ROLLBACK statement to restore the database to the state it was in before you made any modifications.

To erase the employee record in the EMPLOYEES relation with the employee identification number of 00502, use the query shown in example 1:

Examples

Example 1

```
START_TRANSACTION READ_WRITE RESERVING
         EMPLOYEES FOR SHARED WRITE

FOR E IN EMPLOYEES
    WITH E.EMPLOYEE_ID = "00502"
      ERASE E
END_FOR

COMMIT
```

To test that the record for the employee whose identification number is 00502 no longer exists in the EMPLOYEES relation, you can try the following query.

Example 2

```
FOR E IN EMPLOYEES WITH E.EMPLOYEE_ID = "00502"
     PRINT
       E.EMPLOYEE_ID,
       E.FIRST_NAME,
       E.LAST_NAME
END_FOR
```

RDO does not display any data, indicating that the record with the identification number of 00502 has been erased from the EMPLOYEES relation. Of course, there still might be records for the number in the JOB_HISTORY and SALARY_HISTORY relations and perhaps in the DEGREES relation. You can write similar statements to erase these records from these relations. There can be many records belonging to the employee identification number 00502 in each of these three relations. (In other words, EMPLOYEE_ID is not a unique key for some of the relations as it is for EMPLOYEES.)

Before you erase the record in the EMPLOYEES relation, you can use the RSE to find records that should be deleted first in the other relations.

After you erase all records associated with employee 00502, you can verify that the employee's record has been deleted from the EMPLOYEES relation using the query in example 2.

7.6 Summary

The following session demonstrates a sequence of RDO statements updating the database.

```
!
! Invoke the PERSONNEL database.
!
RDO> INVOKE DATABASE PATHNAME 'PERSONNEL'


!
! Signal your access (update) intentions to Rdb/VMS.
!

RDO> START_TRANSACTION READ_WRITE RESERVING
cont>          EMPLOYEES FOR SHARED WRITE


!
! Store a new EMPLOYEE record.
!

RDO> STORE E IN EMPLOYEES USING
```

```
cont>    E.EMPLOYEE_ID = "00503";
cont>    E.FIRST_NAME = "PAUL";
cont>    E.LAST_NAME = "CHRIS"
cont> END_STORE

!
! Make the update permanent.
!

RDO> COMMIT


!
! Store a new SALARY_HISTORY record.
!

RDO> START_TRANSACTION READ_WRITE RESERVING
cont>          SALARY_HISTORY FOR SHARED WRITE,

RDO> STORE SH IN SALARY_HISTORY USING
cont>          SH.EMPLOYEE_ID = "00503"
cont>END_STORE


!
! Make the update permanent.
!

RDO> COMMIT


!
! Request a transaction mode to modify or erase.
!
RDO> START_TRANSACTION READ_WRITE RESERVING
cont>          EMPLOYEES FOR SHARED WRITE,
cont>          SALARY_HISTORY FOR SHARED WRITE


!
! Make first change.
!

RDO>  FOR E IN EMPLOYEES WITH E.EMPLOYEE_ID = "00503"
cont>    CROSS SH.SALARY_HISTORY OVER EMPLOYEE_ID
cont>    WITH SH.SALARY_END MISSING
cont>      MODIFY SH USING
cont>        SH.SALARY_AMOUNT = SH.SALARY_AMOUNT * 1.9
cont>      END_MODIFY
cont> END_FOR
```

```
!
! Mistake! Percent raise is incorrect; undo change.
!

RDO> ROLLBACK



!
! Reset transaction mode and control options.
!

RDO> START_TRANSACTION READ_WRITE RESERVING
cont>          EMPLOYEES FOR SHARED WRITE,
cont>          SALARY_HISTORY FOR SHARED WRITE


!
! Specify correct percent raise.
!

RDO>  FOR E IN EMPLOYEES WITH E.EMPLOYEE_ID = "00503"
cont>   CROSS SALARY_HISTORY OVER EMPLOYEE_ID
cont>   WITH SH.SALARY_END MISSING
cont>     MODIFY SH USING
cont>       SH.SALARY_AMOUNT = SH.SALARY_AMOUNT * 1.1
cont>     END_MODIFY
cont> END_FOR


!
! Erase Employee and Salary History records with ID 00503.
!

RDO> FOR E IN EMPLOYEES WITH E.EMPLOYEE_ID = "00503"
cont>   ERASE E
cont> END_FOR
RDO> FOR SH IN SALARY_HISTORY WITH SH.EMPLOYEE_ID = "00503"
cont>   ERASE SH
cont> END_FOR


!
! Make the change and the deletion permanent.
!

RDO> COMMIT
RDO>
```

Definitions for the PERSONNEL Database A

The definitions shown here create the PERSONNEL database used thoughout this book. Included are the relation names and field names for the PERSONNEL database. Set your default directory to RDM$DEMO and look at the PERSONNEL.COM command file for the definitions.

```
SET DICTIONARY CDD$HOME
!
! *** Define the database ***
!
DEFINE DATABASE 'RDB$HOME:PERSONNEL'
  IN 'CDD$HOME.PERSONNEL'.
!
!
! *** Define fields for the PERSONNEL database ***
!
!
DEFINE FIELD  ID_NUMBER
        DESCRIPTION IS /* Generic employee ID */
        DATATYPE IS TEXT SIZE IS 5.
!
!
DEFINE FIELD  LAST_NAME
        DESCRIPTION IS /* Generic last name */
        DATATYPE IS TEXT SIZE IS 14.
!
!
DEFINE FIELD  FIRST_NAME
        DESCRIPTION IS /* Generic first name */
        DATATYPE IS TEXT SIZE IS 10.
!
!
DEFINE FIELD  MIDDLE_INITIAL
        DESCRIPTION IS /* Generic middle initial */
        DATATYPE IS TEXT SIZE IS 1
        EDIT_STRING FOR DATATRIEVE IS 'X.'
        MISSING_VALUE IS ' '.
!
!
```

```
DEFINE FIELD  ADDRESS_DATA_1
        DESCRIPTION IS /* Street name */
        DATATYPE IS TEXT SIZE IS 25
        MISSING_VALUE IS '                         '.
!
!
DEFINE FIELD  ADDRESS_DATA_2
        DESCRIPTION IS /* Mail stops, suite addresses,
                          street numbers, etc.*/
        DATATYPE IS TEXT SIZE IS 20
        MISSING_VALUE IS '                    '.
!
!
DEFINE FIELD  CITY
        DESCRIPTION IS /* City name */
        DATATYPE IS TEXT SIZE IS 20
        MISSING_VALUE IS '                    '.
!
!
DEFINE FIELD  STATE
        DESCRIPTION IS /* State abbreviation (or DISTRICT)*/
        DATATYPE IS TEXT SIZE IS 2
        MISSING_VALUE IS '  '.
!
!
DEFINE FIELD  POSTAL_CODE
        DESCRIPTION IS /* Postal Code (in US = ZIP) */
        DATATYPE IS TEXT SIZE IS 5
        MISSING_VALUE IS '     '.
!
!
DEFINE FIELD  SEX
       DESCRIPTION IS /* M, F */
        DATATYPE IS TEXT SIZE IS 1
        VALID IF SEX = 'M' OR SEX = 'F' OR
        SEX MISSING
        MISSING_VALUE IS '?'.
!
!
DEFINE FIELD  STANDARD_DATE
        DESCRIPTION IS /* Generic date field */
        DATATYPE IS DATE
        MISSING_VALUE IS '17-NOV-1858 00:00:00.00'
        EDIT_STRING FOR DATATRIEVE IS 'DD-MMM-YYYY'.


!
!
DEFINE FIELD  SALARY
        DESCRIPTION IS /* Generic salary field */
        DATATYPE IS SIGNED LONGWORD SCALE -2
        VALID IF SALARY > 0 OR
        SALARY MISSING
        EDIT_STRING FOR DATATRIEVE IS '$$$$,$$9.99'.
!
!
DEFINE FIELD  RESUME
        DESCRIPTION IS /* Employee resume */
        DATATYPE IS SEGMENTED STRING.
!
```

```
!
DEFINE FIELD  DEPARTMENT_CODE
        DESCRIPTION IS /* Department code or abbreviation */
        DATATYPE IS TEXT SIZE IS 4
        MISSING_VALUE IS 'None'.
!
!
DEFINE FIELD  JOB_CODE
        DESCRIPTION IS /* Generic job code */
        DATATYPE IS TEXT SIZE IS 4
        MISSING_VALUE IS 'None'.
!
!
DEFINE FIELD  WAGE_CLASS
        DESCRIPTION IS /* Wage class -- 1 to 4 */
        DATATYPE IS TEXT SIZE IS 1
        VALID IF WAGE_CLASS = '1' OR
                 WAGE_CLASS = '2' OR
                 WAGE_CLASS = '3' OR
                 WAGE_CLASS = '4' OR
                 WAGE_CLASS MISSING.
!
!
DEFINE FIELD  JOB_TITLE
        DESCRIPTION IS /* Generic job title */
        DATATYPE IS TEXT SIZE IS 20
        MISSING_VALUE IS 'None'.
!
!
DEFINE FIELD  DEPARTMENT_NAME
        DESCRIPTION IS /* Department name */
        DATATYPE IS TEXT SIZE IS 30
        MISSING_VALUE IS 'None'.
!
!
DEFINE FIELD  BUDGET
        DESCRIPTION IS /* Generic budget data */
        DATATYPE IS SIGNED LONGWORD SCALE 0
        EDIT_STRING FOR DATATRIEVE IS '$$$,$$$,$$$'.
!
!
DEFINE FIELD  COLLEGE_NAME
        DESCRIPTION IS /* Halls of ivy */
        DATATYPE IS TEXT SIZE IS 25.
!
!
DEFINE FIELD  COLLEGE_CODE
        DESCRIPTION IS /* Four-letter college code */
        DATATYPE IS TEXT SIZE IS 4.
!
!
DEFINE FIELD  YEAR_GIVEN
        DESCRIPTION IS /* Year degree awarded */
        DATATYPE IS SIGNED WORD.
!
!
DEFINE FIELD  DEGREE
        DESCRIPTION IS /* Degree awarded */
        DATATYPE IS TEXT SIZE IS 3
        VALID IF DEGREE = 'BA ' OR
```

```
                DEGREE = 'BS ' OR
                DEGREE = 'MA ' OR
                DEGREE = 'MS ' OR
                DEGREE = 'AA ' OR
                DEGREE = 'PhD' OR
                DEGREE MISSING.
!
!
DEFINE FIELD  DEGREE_FIELD
        DESCRIPTION IS /* Field in which
                          degree was awarded */
        DATATYPE IS TEXT SIZE IS 15
        MISSING_VALUE IS 'Unknown'.
!
!
DEFINE FIELD  STATUS_CODE
        DESCRIPTION IS /* A number */
        DATATYPE IS TEXT SIZE IS 1
        MISSING_VALUE IS 'N'
        VALID IF STATUS_CODE = '0' OR
                 STATUS_CODE = '1' OR
                 STATUS_CODE = '2' OR
                 STATUS_CODE MISSING.
!
!
DEFINE FIELD        STATUS_NAME
        DESCRIPTION IS /* Active or inactive */
        DATATYPE IS TEXT SIZE IS 8
        VALID IF STATUS_NAME = 'ACTIVE' OR
                 STATUS_NAME = 'INACTIVE' OR
                 STATUS_NAME NOT MISSING.
!
!
DEFINE FIELD        STATUS_TYPE
        DESCRIPTION IS /* Full-time,
                          part-time, or expired */
        DATATYPE IS TEXT SIZE IS 14
        VALID IF STATUS_TYPE = 'RECORD EXPIRED' OR
                 STATUS_TYPE = 'FULL TIME' OR
                 STATUS_TYPE = 'PART TIME' OR
                 STATUS_TYPE MISSING.
!
!
!*************************************************************
!
!
! *** Define Relations ***
!
DEFINE RELATION EMPLOYEES.
   EMPLOYEE_ID
      BASED ON ID_NUMBER.
   LAST_NAME.
   FIRST_NAME.
   MIDDLE_INITIAL.
   ADDRESS_DATA_1.
   ADDRESS_DATA_2.
   CITY.
   STATE.
   POSTAL_CODE.
   SEX.
```

```
    BIRTHDAY
        BASED ON STANDARD_DATE.
    STATUS_CODE.
END EMPLOYEES RELATION.
!
!
! Job_History Relation:
!
DEFINE RELATION JOB_HISTORY.
    EMPLOYEE_ID
        BASED ON ID_NUMBER.
    JOB_CODE.
    JOB_START
        BASED ON STANDARD_DATE.
    JOB_END
        BASED ON STANDARD_DATE.
    DEPARTMENT_CODE.
    SUPERVISOR_ID
        BASED ON ID_NUMBER.
END JOB_HISTORY RELATION.
!
!
! Salary_History Relation:
!
DEFINE RELATION SALARY_HISTORY.
    EMPLOYEE_ID
        BASED ON ID_NUMBER.
    SALARY_AMOUNT
        BASED ON SALARY.
    SALARY_START
        BASED ON STANDARD_DATE.
    SALARY_END
        BASED ON STANDARD_DATE.
END SALARY_HISTORY RELATION.
!
!
! Jobs Relation:
!
DEFINE RELATION JOBS.
    JOB_CODE.
    WAGE_CLASS.
    JOB_TITLE.
    MINIMUM_SALARY
        BASED ON SALARY.
    MAXIMUM_SALARY
        BASED ON SALARY.
END JOBS RELATION.
!
!
! Departments Relation:
!
DEFINE RELATION DEPARTMENTS.
    DEPARTMENT_CODE.
    DEPARTMENT_NAME.
    MANAGER_ID
        BASED ON ID_NUMBER.
    BUDGET_PROJECTED
        BASED ON BUDGET.
    BUDGET_ACTUAL
        BASED ON BUDGET.
```

```
END DEPARTMENTS RELATION.
!
!
! Colleges Relation:
!
DEFINE RELATION COLLEGES.
    COLLEGE_CODE.
    COLLEGE_NAME.
    CITY.
    STATE.
    POSTAL_CODE.
END COLLEGES RELATION.
!
!
! Degrees Relation:
!
DEFINE RELATION DEGREES.
    EMPLOYEE_ID
        BASED ON ID_NUMBER.
    COLLEGE_CODE.
    YEAR_GIVEN.
    DEGREE.
    DEGREE_FIELD.
END DEGREES RELATION.
!
!
! Work_Status Relation:
!
DEFINE RELATION WORK_STATUS.
    STATUS_CODE.
    STATUS_NAME.
    STATUS_TYPE.
END WORK_STATUS RELATION.
!
DEFINE RELATION RESUMES.
    EMPLOYEE_ID
        BASED ON ID_NUMBER.
    RESUME.
END RESUMES RELATION.
!
COMMIT
!
!
!************************************************************
!
!
! *** Define three views to get current information ***
!
!
! Current job information
!
DEFINE VIEW CURRENT_JOB OF JH IN JOB_HISTORY
  CROSS E IN EMPLOYEES OVER EMPLOYEE_ID
      WITH JH.JOB_END MISSING.
       E.LAST_NAME.
       E.FIRST_NAME.
       E.EMPLOYEE_ID.
       JH.JOB_CODE.
       JH.DEPARTMENT_CODE.
       JH.SUPERVISOR_ID.
```

```
      JH.JOB_START.
END VIEW.
!
!
! Current salary information
!
DEFINE VIEW CURRENT_SALARY OF SH IN SALARY_HISTORY
 CROSS E IN EMPLOYEES OVER EMPLOYEE_ID
   WITH SH.SALARY_END MISSING.
     E.LAST_NAME.
     E.FIRST_NAME.
     E.EMPLOYEE_ID.
     SH.SALARY_START.
     SH.SALARY_AMOUNT.
END VIEW.
!
!
! Current salary and job information
!
DEFINE VIEW CURRENT_INFO OF CJ IN CURRENT_JOB
  CROSS D IN DEPARTMENTS OVER DEPARTMENT_CODE
  CROSS J IN JOBS OVER JOB_CODE
  CROSS CS IN CURRENT_SALARY OVER EMPLOYEE_ID.
     LAST FROM CJ.LAST_NAME.
     FIRST FROM CJ.FIRST_NAME.
     ID FROM CJ.EMPLOYEE_ID.
     DEPARTMENT FROM D.DEPARTMENT_NAME.
     JOB FROM J.JOB_TITLE.
     JSTART FROM CJ.JOB_START.
     SSTART FROM CS.SALARY_START.
     SALARY FROM CS.SALARY_AMOUNT.
END VIEW.
!
COMMIT
!
!        Store three Work Status Codes in WORK_STATUS relation
!
START_TRANSACTION READ_WRITE RESERVING WORK_STATUS FOR SHARED WRITE
STORE W IN WORK_STATUS USING
W.STATUS_CODE="0";
W.STATUS_NAME="INACTIVE";
W.STATUS_TYPE="RECORD EXPIRED";END_STORE
STORE W IN WORK_STATUS USING
W.STATUS_CODE="1";
W.STATUS_NAME="ACTIVE";
W.STATUS_TYPE="FULL TIME";END_STORE
STORE W IN WORK_STATUS USING
W.STATUS_CODE="2";
W.STATUS_NAME="ACTIVE";
W.STATUS_TYPE="PART TIME";END_STORE
COMMIT
!
FINISH
EXIT
!
!
!
INVOKE DATABASE PATHNAME 'CDD$HOME:PERSONNEL'
!
!  Display statistics for PERSONNEL
```

```
!
SET NOVERIFY
SET OUTPUT COUNT.LOG
PRINT "   "
PRINT "Statistics for PERSONNEL, the sample Rdb/VMS database:"
PRINT "  "
PRINT "Count of Employees -------> ",
              COUNT OF E IN EMPLOYEES
PRINT "Count of Jobs ------------> ",
              COUNT OF J IN JOBS
PRINT "Count of Degrees ---------> ",
              COUNT OF D IN DEGREES
PRINT "Count of Salary_History --> ",
              COUNT OF SH IN SALARY_HISTORY
PRINT "Count of Job_History -----> ",
              COUNT OF JH IN JOB_HISTORY
PRINT "Count of Work_Status -----> ",
              COUNT OF W IN WORK_STATUS
PRINT "Count of Departments -----> ",
              COUNT OF D IN DEPARTMENTS
PRINT "Count of Colleges --------> ",
              COUNT OF C IN COLLEGES
PRINT "Count of Resumes ---------> ",
              COUNT OF R IN RESUMES
PRINT "  "
PRINT "  "
EXIT
```

Using a VMS Text Editor for Program Development B

This appendix shows how you can use the Relational Database Operator (RDO) and the VAX EDT or VAXTPU text editors to test and debug data manipulation statements to be included in application programs.

RDO and the VAX text editors provide a single environment for developing database applications. RDO lets you try on the terminal almost every operation you can use in a program. You can type a command and see the results immediately. The EDIT command lets you use EDT or VAXTPU to modify an RDO statement, so that you can continuously refine that statement until it does exactly what you want it to do.

The EDIT command is suited to the structure of the Rdb/VMS data manipulation statements. Each clause of the record selection expression defines the record stream more specifically. Thus you can use the EDIT command to add clauses, until you have the data you need. When the results are what you want, you can use VAX EDT or VAXTPU to write the RDO statement that writes to an external file. Then you can include that file in the source code for your program.

When you type the EDIT command, you place that last complete RDO statement in the editing buffer. RDO keeps every command and statement you enter in your terminal session in a separate buffer. You can retrieve any complete statement or all of them for editing by using the following VAX EDT or VAXTPU options from RDO:

- EDIT

  This command places the last complete command or statement you typed into the editing buffer. You can execute the query again by exiting from VAX EDT or VAXTPU with the VAX EDT or VAXTPU EXIT command or you can direct RDO not to execute it by entering the VAX EDT or VAXTPU QUIT command.

- EDIT 0

  This command invokes EDT or VAXTPU with an empty editing buffer. You can enter a query in the editing buffer and write it to a file for later execution without exiting from RDO. When you finish entering the query, you can also test it immediately by exiting from VAX EDT or VAXTPU with the VAX EDT or VAXTPU EXIT command.

- EDIT *n*

  This command replaces *n* with an integer, places the last *n* commands in the editing buffer.

- EDIT *

  This command places every command or statement you have entered during your current interactive session in the editing buffer. This method allows you to change the order of your statements or to include only specific, correct statements in a single transaction. You can then write these statements to a file and use them again.

You cannot edit a command file with the EDIT command in RDO. Specifying a file name after the EDIT command returns a syntax error. To modify an existing command file, you can:

- Exit from RDO, edit the file, and return to RDO
- Type EDIT 0 and use the VAX EDT or VAXTPU INCLUDE <file-name> command to place the command file in the empty buffer

Depending on the command you use to exit from the editor, RDO responds in different ways. When you type:

- EXIT

  RDO automatically executes the contents of the editing buffer and the RDO> prompt appears on completion of the edited statements.

- QUIT

  RDO does not attempt to execute the contents of your editing buffer. RDO ignores the changes you made during the edit session. The RDO> prompt appears immediately.

The sequence that follows demonstrates the technique of progressively specifying a query and reproduces a log file of an RDO terminal session. Each statement is modified using EDIT.

Suppose you must retrieve the current employees from a particular department. List last name, first name, job title, and department name.

```
SET OUTPUT QUERY.LOG
START_TRANS READ_ONLY
!
! Print the employees.
!

FOR FIRST 5 E IN EMPLOYEES
PRINT
    E.LAST_NAME,
    E.FIRST_NAME
END_FOR
 Toliver            Alvin
 Smith              Terry
 Dietrich           Rick
 Kilpatrick         Janet
 Nash               Norman
EDIT
!
! Join EMPLOYEES to JOB_HISTORY to get JOB_CODE.
!
FOR FIRST 5 E IN EMPLOYEES CROSS
    J IN JOB_HISTORY OVER EMPLOYEE_ID
PRINT
    E.LAST_NAME,
    E.FIRST_NAME,
    J.JOB_CODE
END_FOR
 Toliver            Alvin           DMGR
 Toliver            Alvin           SPGM
 Toliver            Alvin           MENG
 Smith              Terry           DGFR
 Smith              Terry           DGFR
EDIT
!
! Join JOB_HISTORY to DEPARTMENTS to get DEPARTMENT_NAME.
!
FOR FIRST 5 E IN EMPLOYEES CROSS
    J IN JOB_HISTORY OVER EMPLOYEE_ID CROSS
    D IN DEPARTMENTS OVER DEPARTMENT_CODE


PRINT
    E.LAST_NAME,
    E.FIRST_NAME,
    J.JOB_CODE,
    D.DEPARTMENT_NAME
END_FOR
 Toliver            Alvin           SPGM   Cabinet and Frame Manufacturing
 Toliver            Alvin           DMGR   Board Manufacturing North
 Toliver            Alvin           MENG   Board Manufacturing North
 Smith              Terry           DGFR   Board Manufacturing
 Smith              Terry           DGFR   Telecommunications Industries
EDIT
!
! Join JOB_HISTORY to JOBS to get JOB_TITLE.
!
```

```
FOR FIRST 5 E IN EMPLOYEES CROSS
    J IN JOB_HISTORY OVER EMPLOYEE_ID CROSS
    D IN DEPARTMENTS OVER DEPARTMENT_CODE CROSS
    JO IN JOBS WITH JO.JOB_CODE = J.JOB_CODE
PRINT
    E.LAST_NAME,
    JO.JOB_TITLE,
    D.DEPARTMENT_NAME
END_FOR
 Toliver          Systems Programmer      Cabinet & Frame Manufacturing
 Toliver          Department Manager      Board Manufacturing North
 Toliver          Mechanical Engineer     Board Manufacturing North
 Smith            Deputy                  Large Systems Engineering
 Smith            Deputy                  Telecommunications Industries
EDIT
!
! Specify JOB_END MISSING to limit stream to current jobs.
!
FOR FIRST 5 E IN EMPLOYEES CROSS
    J IN JOB_HISTORY OVER EMPLOYEE_ID CROSS
    D IN DEPARTMENTS OVER DEPARTMENT_CODE CROSS
    JO IN JOBS WITH JO.JOB_CODE = J.JOB_CODE AND
    J.JOB_END MISSING
PRINT
    E.LAST_NAME,
    JO.JOB_TITLE,
    D.DEPARTMENT_NAME
END_FOR
 Williams         Janitor                 Corporate Administration
 O'Sullivan       Mechanical Engineer     Corporate Administration
 Gramby           Electrical Engineer     Corporate Administration
 Wilkins          Mechanical Engineer     Corporate Administration
 Myotte           Vice President          Corporate Administration
EDIT
!
! Specify a particular department.
!
FOR FIRST 5 E IN EMPLOYEES CROSS
    J IN JOB_HISTORY OVER EMPLOYEE_ID CROSS
    D IN DEPARTMENTS OVER DEPARTMENT_CODE CROSS
    JO IN JOBS WITH JO.JOB_CODE = J.JOB_CODE AND
    J.JOB_END MISSING AND
    D.DEPARTMENT_CODE = "ELEL"

PRINT
    E.LAST_NAME,
    JO.JOB_TITLE,
    D.DEPARTMENT_NAME
END_FOR
 Iacobone         Systems Analyst         Electronics Engineering
 Lobdell          Associate Programmer    Electronics Engineering
 Siciliano        Mechanical Engineer     Electronics Engineering
 Leger            Mechanical Engineer     Electronics Engineering
 Jones            Programmer              Electronics Engineering
EDIT
!
! Sort the records by LAST_NAME.
!
FOR FIRST 5 E IN EMPLOYEES CROSS
    J IN JOB_HISTORY OVER EMPLOYEE_ID CROSS
```

```
    D IN DEPARTMENTS OVER DEPARTMENT_CODE CROSS
    JO IN JOBS WITH JO.JOB_CODE = J.JOB_CODE AND
    J.JOB_END MISSING AND
    D.DEPARTMENT_CODE = "ELEL"
    SORTED BY E.LAST_NAME
PRINT
    E.LAST_NAME,
    E.FIRST_NAME,
    JO.JOB_TITLE,


    D.DEPARTMENT_NAME
END_FOR
 Augusta        Thomas     Electrical Engineer     Electronics Engineering
 Boutin         George     Clerk                   Electronics Engineering
 Clairmont      Rick       Clerk                   Electronics Engineering
 Flynn          Peter      Electrical Engineer     Electronics Engineering
 Gutierrez      Ernest     Systems Analyst         Electronics Engineering
```

Now that you have arrived at the correct form of the query, you can write the results to a file:

```
RDO> EDIT
* WRITE QUERY.RDO
* QUIT

RDO> ROLLBACK
RDO> FINISH
RDO> EXIT
```

Now you can use VAX EDT or VAXTPU to create the program.

```
$ EDIT QUERY.RBA
* INCLUDE QUERY.RDO
* CHANGE

!
! Sort the records by LAST_NAME.
!
FOR FIRST 5 E IN EMPLOYEES CROSS
    J IN JOB_HISTORY OVER EMPLOYEE_ID CROSS
    D IN DEPARTMENTS OVER DEPARTMENT_CODE CROSS
    JO IN JOBS WITH JO.JOB_CODE = J.JOB_CODE AND
    J.JOB_END MISSING AND
    D.DEPARTMENT_CODE = "ELEL"
    SORTED BY E.LAST_NAME
PRINT
    E.LAST_NAME,
    E.FIRST_NAME,
    JO.JOB_TITLE,
    D.DEPARTMENT_NAME
END_FOR
[EOB]
```

At this point, you have the central logic for a VAX BASIC report program. Now you can add the rest of the required and optional elements. The final program might look like the following (with no error checking).

```
1  DECLARE STRING LAST_NAME, FIRST_NAME, JOB_TITLE, &
        DEPARTMENT_NAME, DEPARTMENT_CODE

&RDB&         INVOKE DATABASE FILENAME "DISK2:[DEPT3]PERSONNEL"
&RDB&         START_TRANSACTION READ_ONLY

        INPUT "Department code"; DEPARTMENT_CODE

&RDB&         FOR D IN DEPARTMENTS
&RDB&              WITH D.DEPARTMENT_CODE = DEPARTMENT_CODE
&RDB&           GET DEPARTMENT_NAME = D.DEPARTMENT_NAME
&RDB&      END_GET
&RDB&         END_FOR

        PRINT " "
        PRINT "Employee list for "; DEPARTMENT_NAME
        PRINT " "
        PRINT "Last name", ,"First name", "Job"
        PRINT " "

&RDB&         FOR E IN EMPLOYEES CROSS
&RDB&        J IN JOB_HISTORY OVER EMPLOYEE_ID CROSS
&RDB&        D IN DEPARTMENTS OVER DEPARTMENT_CODE CROSS
&RDB&        JO IN JOBS WITH JO.JOB_CODE = J.JOB_CODE AND
&RDB&        J.JOB_END MISSING AND
&RDB&        D.DEPARTMENT_CODE = DEPARTMENT_CODE
&RDB&        SORTED BY E.LAST_NAME
&RDB&    GET
&RDB&        LAST_NAME = E.LAST_NAME;
&RDB&        FIRST_NAME = E.FIRST_NAME;
&RDB&        JOB_TITLE = JO.JOB_TITLE;
&RDB&    END_GET

        PRINT LAST_NAME, FIRST_NAME, JOB_TITLE

&RDB&    END_FOR
&RDB&         COMMIT
&RDB&         FINISH

32767         END
```

Index

In this index, a page number followed by a "t" indicates a table reference. A page number followed by an "f" indicates a figure reference.

D

E

F

G

H

I

J

L

M

N

O

P

Q

R

S

T

U

V

W

How to Order Additional Documentation

If you live in:	Call:	or Write:
New Hampshire, Alaska	603-884-6660	Digital Equipment Corp. P.O. Box CS2008 Nashua, NH 03061-2698
Continental USA, Puerto Rico, Hawaii	1-800-258-1710	Same as above.
Canada (Ottawa-Hull)	613-234-7726	Digital Equipment Corp. 940 Belfast Road Ottawa, Ontario K1G 4C2 Attn: P&SG Business Manager or approved distributor
Canada (British Columbia)	1-800-267-6146	Same as above.
Canada (All other)	112-800-267-6146	Same as above.
All other areas	—	Digital Equipment Corp. Peripherals & Supplies Centers P&SG Business Manager c/o DIGITAL's local subsidiary

Note: Place prepaid orders from Puerto Rico with the local DIGITAL subsidiary (phone 809-754-7575).

Place internal orders with the Software Distribution Center, Digital Drive, Westminster, MA 01473-0471.

Reader's Comments

VAX Rdb/VMS
Guide to Data Manipulation
AA–N036C–TE

Please use this postage-paid form to comment on this manual. If you require a written reply to a software problem and are eligible to receive one under Software Performance Report (SPR) service, submit your comments on an SPR form.

Thank you for your assistance.

I rate this manual's:	Excellent	Good	Fair	Poor
Accuracy (software works as manual says)	☐	☐	☐	☐
Completeness (enough information)	☐	☐	☐	☐
Clarity (easy to understand)	☐	☐	☐	☐
Organization (structure of subject matter)	☐	☐	☐	☐
Figures (useful)	☐	☐	☐	☐
Examples (useful)	☐	☐	☐	☐
Index (ability to find topic)	☐	☐	☐	☐
Page layout (easy to find information)	☐	☐	☐	☐

I would like to see more/less _____

_____

What I like best about this manual is _____

_____

What I like least about this manual is _____

_____

I found the following errors in this manual:

Page	Description
_____	_____
_____	_____
_____	_____

Additional comments or suggestions to improve this manual:

_____

_____

_____

I am using **Version** _____ of the software this manual describes.

Name/Title _____ Dept. _____

Company _____ Date _____

Mailing Address _____

_____ Phone _____

– Do Not Tear - Fold Here and Tape ----------

digital™

No Postage
Necessary
if Mailed
in the
United States

BUSINESS REPLY MAIL

FIRST CLASS PERMIT NO. 33 MAYNARD MASS.

POSTAGE WILL BE PAID BY ADDRESSEE

DIGITAL EQUIPMENT CORPORATION
Corporate User Publications—Spit Brook
ZK01–3/J35
110 SPIT BROOK ROAD
NASHUA, NH 03062-9987

– Do Not Tear - Fold Here ----------

Cut Along Dotted Line

Reader's Comments

VAX Rdb/VMS
Guide to Data Manipulation
AA–N036C–TE

Please use this postage-paid form to comment on this manual. If you require a written reply to a software problem and are eligible to receive one under Software Performance Report (SPR) service, submit your comments on an SPR form.

Thank you for your assistance.

I rate this manual's:	Excellent	Good	Fair	Poor
Accuracy (software works as manual says)	☐	☐	☐	☐
Completeness (enough information)	☐	☐	☐	☐
Clarity (easy to understand)	☐	☐	☐	☐
Organization (structure of subject matter)	☐	☐	☐	☐
Figures (useful)	☐	☐	☐	☐
Examples (useful)	☐	☐	☐	☐
Index (ability to find topic)	☐	☐	☐	☐
Page layout (easy to find information)	☐	☐	☐	☐

I would like to see more/less _____

_____

What I like best about this manual is _____

_____

What I like least about this manual is _____

_____

I found the following errors in this manual:

Page	Description
_____	_____
_____	_____
_____	_____

Additional comments or suggestions to improve this manual:

_____

_____

_____

I am using **Version** _____ of the software this manual describes.

Name/Title _____ Dept. _____

Company _____ Date _____

Mailing Address _____

_____ Phone _____

-- Do Not Tear - Fold Here and Tape ----------

digital™

No Postage
Necessary
if Mailed
in the
United States

BUSINESS REPLY MAIL

FIRST CLASS PERMIT NO. 33 MAYNARD MASS.

POSTAGE WILL BE PAID BY ADDRESSEE

DIGITAL EQUIPMENT CORPORATION
Corporate User Publications—Spit Brook
ZK01–3/J35
110 SPIT BROOK ROAD
NASHUA, NH 03062-9987

-- Do Not Tear - Fold Here ----------

Cut Along Perforated Line