VAX Rdb/VMS

VAX Rdb/VMS Guide to Database Design and Definition

Order No. AA-N034B-TE

digital™

software

VAX Rdb/VMS Guide to Database Design and Definition

Order No. AA-N034B-TE

December 1985

This manual shows how to design a logical database compatible with the relational data model. It then demonstrates how to build a physical database using VAX Rdb/VMS data definition statements. The material in this manual is tutorial in nature.

OPERATING SYSTEM: VMS
MicroVMS

SOFTWARE VERSION: VAX Rdb/VMS V2.0

digital equipment corporation, maynard, massachusetts

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.

Contents

How to Use This Manual

VAX Rdb/VMS is a general purpose database management system based on the relational data model.

Purpose of This Manual

This manual demonstrates how to:

- Design a database that is compatible with the relational data model.
- Use the data definition statements of RDO, the interactive VAX Rdb/VMS utility, to translate a logical database model into a physical database.

Intended Audience

If you have not designed a database before, this book provides guidance on how to analyze an information management problem. You can use your analysis to design a database.

This book shows all users how to use the data definition statements of RDO to implement a database design.

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

To verify which versions of your operating system are compatible with this version of VAX Rdb/VMS, check the most recent copy of the following:

- For the VMS operating system -- *VAX/VMS Optional Software Cross Reference Table*, SPD 25.99.xx
- For the MicroVMS operating system -- *MicroVMS Optional Software Cross Reference Table*, SPD 28.99.xx

Structure

This manual contains four chapters, one appendix, and an index.

Chapter 1	Introduces concepts of data management, data organization, and the relational data model. This chapter also shows you how to design a logical database.
Chapter 2	Describes how to translate the logical database into a physical database using DEFINE statements.
Chapter 3	Shows how to define protection for your database using the DEFINE PROTECTION statement. This chapter includes a discussion of access control lists (ACLs).
Chapter 4	Tells how to change or delete an existing database or elements within an existing database.
Appendix A	Provides sample command files for building your own version of the sample database.
Index	

Related Manuals

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

VAX Rdb/VMS Reference Manual

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

VAX Rdb/VMS Guide to Data Manipulation

A tutorial on how to use the components of VAX Rdb/VMS to retrieve, store, change, and erase data

VAX Rdb/VMS Guide to Programming

A tutorial on how to write high-level language programs that use VAX Rdb/VMS for database access

VAX Rdb/VMS Guide to Database Administration and Maintenance

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

The following books provide information about the VMS operating system, VAX DATATRIEVE, and the VAX Common Data Dictionary:

The VAX DATATRIEVE Documentation Set

The VAX Common Data Dictionary Documentation Set

VAX Architecture Handbook

Detailed information about VAX computers and VAX data types

VAX Information Architecture Summary Description

A description of VAX Information Architecture component software products

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.

Conventions used in this manual are:

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

Data Organization 1

As you design a database to manage, control, and disseminate information, you should consider incorporating the following points in your design goals:

- The logical data model you define reflects the data relationships in your organization and the way your users view these relationships.
- The database you create from your logical model supports the needs of your users to deliver the correct information at the right time.
- Data is available to all authorized users.
- The database management system is flexible enough to permit restructuring without inconveniencing people who are using the database.
- The database design balances user needs with the most common types of database activity for efficient performance. Such database activity includes update and retrieval.

Each chapter of this book concerns itself with a specific point from the preceding list.

1.1 Database Management Systems

A database management system is a set of software tools that provides a single environment for storing, retrieving, changing, and protecting data. By using a database management system together with application programs and other software tools, you can turn large amounts of data into usable information.

1.1.1 Relational Database Model

VAX Rdb/VMS is a relational database management product that uses the relational model of database organization. The relational model maintains data in two-dimensional tabular format similar to hard copy tables and "flat" computer files. This tabular format keeps data organization simple and easy to understand. Other DIGITAL software products, such as DATATRIEVE, can take advantage of the features of Rdb/VMS.

1.2 Data

Data is the general term used to describe a collection of facts. A **data item** is a type of fact. For example, such categories as age, height, and price are data items. A **data value** is the specific instance of a data item, for example, 26 years old, 5 feet 11 inches, or $26.95. You can record data values directly from a business transaction or an observation, or compute them from the values of other data items.

Data items are the smallest meaningful units of information. You can manipulate data items by:

- Grouping them with other data items to create unique descriptions of objects
- Changing their values to reflect a current state or condition
- Erasing values of data items if they are invalid or become obsolete
- Adding new values for data items to develop a complete picture of your business activity

To be useful, data items must be organized into logical groups. For instance, when you want to describe an employee, you assemble data items such as age, address, and telephone number. Each data item is a label for the type of value you assign to it. You can group data items together to identify one particular individual, the employee. The more data items you assemble, the more accurately and uniquely you can describe that person. Each item that describes an employee shares a **logical relationship** with the other data items associated with that employee.

The following example shows data values for six data items. Each row describes a different employee.

```
Data    |                             Zip   Telephone
Items   | Name    Address             code  number          Height  Weight

Data    | Smith   10 Main Street      00111 (619) 555-1323  6'      155
Values  | Jones   234 Elm Street      00112 (619) 555-4321  5'10"   162
```

Different departments in an organization might view the employee in different ways. The payroll department might see the employee in terms of annual salary, employee identification number, social security number, and number of dependents. Management might view the employee as an individual who performs specific jobs, with special skills and responsibilities. You must organize the data items in your system to accommodate such differing views.

Different terminology is used to describe the various elements of databases. Table 1-1 shows the correspondences between different sets of terminology.

Table 1-1: Traditional and Relational Terminology

Database Entity	Relational Jargon	File Systems	VAX Rdb/VMS
Table	Relation	File	Relation
Column	Attribute	Field	Field
Row	Tuple	Record	Record

The data items you collect and the way you arrange them in the database depend on what information your organization needs for its day-to-day operations and planning. To determine the data items you need, identify an object, such as an employee, an inventory item, or a discount value, in the organization and list the organizational functions that use it. Table 1-2 lists some of the objects and the organizational functions that use them.

Table 1-2: Many Functions Using the Same Object

Object	Organizational Function
A Product	Inventory Sales Warehousing Advertising Marketing
A Service	Customer Cost, Pricing Personnel, Staff Materials
An Employee	Personnel Payroll Management Security

In addition to sharing certain data items with other functions, each organizational function needs its own set of data items to describe the object. Table 1-3 lists all of the data items needed by each function.

Table 1-3: Data Items Required by Functions

Organizational Function	Object Described	Data Items Describing the Object
Personnel	Employee	Name Address Social security number Sex Birthday
Payroll	Employee	Job classification Social security number Name Department name Job title Salary Dependents
Management	Employee	Job title Name Department name Job history Skills Education
Security	Employee	Badge number Social security number Department number Auto license number Office telephone number

1.3 Defining a Logical Database Model

This chapter shows you how to define a logical database model identifying all necessary data items, the flow of data from one department to another, and the logical relationships among the data items. To define an Rdb/VMS database, you need to perform the following procedures.

- Identify functions

  List organizational functions, or departments, to be included in the proposed system.

- List objects

  Within each function, identify all objects about which you need to maintain data. Objects include such things as employees, parts, vendors, and buildings.

- List data items

  Under each object, list every data item that describes it, as in Table 1-3.

- Normalize your model

  Take maximum advantage of the flexibility of the Rdb/VMS relational model by normalizing the design of your logical database. The process of normalization includes the following procedures:

  - Eliminating duplicates

    If several functions list the same data item (for example, Employee Name) under an object, include it only once. Table 1-3 shows that Personnel, Payroll, and Security list the social security number as a required data item. Decide what department or function has primary responsibility for collecting values for this data item and list it under that function. In this case, Personnel might record values for social security numbers and let other departments share that data. Use a list similar to the one in Table 1-3 to locate and eliminate duplicate data items.

    After testing a working version of your database, you might discover that duplicating certain fields in more than one relation gives you added convenience in retrieving data. For some types of routine database activity, such *controlled data redundancy* can be beneficial. See Chapter 4 of the *VAX Rdb/VMS Guide to Database Administration and Maintenance* for a discussion of the benefits and penalties of controlled redundancy.

  - Identifying primary and foreign keys

    The relational database model relies on both *primary* and *foreign key* values to determine relationships among data elements. Both primary and foreign keys have special characteristics that allow them to function as identifiers and links in the database. Refer to Chapter 4 of the Database Administration and Maintenance Guide for a description of primary and foreign key characteristics.

To determine which data items (fields) can serve as primary or foreign keys, examine data items that uniquely identify the object or provide a link with another logical function. You can use an employee identification number to locate an individual employee in a list of all employees in the organization. Therefore, even when two employees share the same last name, for example, you can isolate a single employee record by using the value of the employee identification number.

- Eliminating repeating fields

  Some fields have a characteristic called indexed, repeating, or group field types. For example, a field called Child can have several values, each representing individual children of an employee. If an employee has three children, the Child field is actually a list of three children. The relational model permits only elementary fields, or single-valued facts. One field can have only one value. Section 1.3.4.1 shows how you can remove repeating fields and use those fields to create a new relation.

- Checking functional dependencies

  Examine each data item for *functional dependency* on the key field or fields you use to identify a single instance of an object. Every data item in each object group should depend explicitly on a key field (or combination of fields that make up the key) that uniquely identifies a record.

  For example, if you use a badge number in an employee record to identify an individual employee, the employee's name depends on the value of the badge number. The employee's name provides some information about the key field, badge number, and each badge number identifies one employee. On the other hand, an employee's department code does not provide any information about his or her badge number and therefore is not functionally dependent on the field, badge number.

• Defining fields

  Specify field characteristics such as field name, data type, and field size.

• Defining relations

  Name each relation, or function, and specify all the data items that each will include. Arranging the data items, or fields, into logical groups simplifies the work of defining such database entities as relations. A relation is merely the relational term for a group of data items, called fields, that are logically related. When you define a relation, you are actually defining a record and its component fields.

1.3.1 Identifying Functions

Most organizations are divided into several departments or groups performing specific tasks. These specialized functions often work with the same objects. For example, if the company markets a service that repairs or maintains computer terminals, one function maintains information about replacement parts inventory. Another keeps outstanding customer service requests and schedules of available service technicians. A third function supports payroll and personnel data about the employees.

Each of these functions collects and maintains data for its own tasks and shares some of this data with other functions in the organization. You can best describe your organization by listing all of the functions, or departments, and the tasks each performs. The complete description, or business model, of your organization is useful in selecting that part of the model you plan to include in your database. Start with parts of the logical model to build and test a manageable database. Later, when you have tested your working database, you can add more departments or functions to that database, or you can create another database for the additional parts.

To illustrate the steps in defining your logical database model, the following sample application is used throughout the rest of this book.

The Design Problem

The Overnite Hotel has approximately four hundred rooms. Its room types include singles, doubles, and suites. The hotel often books rooms a year in advance. Rates vary according to the category of guest. There are different rates for group, government, and standard categories.

Business has increased dramatically in the past few years. The hotel's reservation procedures are inefficient, and the present paper system can no longer handle the volume of reservations. Many rooms remain unsold because of cumbersome cross referencing methods. The Overnite Hotel, therefore, needs a system to manage and control its resources.

A system that handles the hotel's reservations must support the following tasks:

- Controlling rooms inventory

  The hotel must know at all times which rooms are reserved and which are available. This might be a requirement of the Reservation function.

- Managing and controlling billing

  The hotel must be able to compute charges and bill guests for services efficiently and accurately. This might be a requirement from a Billing function.

- Determining effectiveness of advertising and sales force

  By analyzing the types of rooms sold and the types of guests reserving them, the sales force can determine the correct sales emphasis. A Billing function might record data for this output.

- Identifying established customers

  Keeping track of past transactions shows which customers are likely to return for repeat business. This data might be available from a Guest function.

- Identifying market mix

  The hotel studies the ratios of government, commercial, and regular guests to determine its attractions and future sales approaches. Information about such hotel transactions might come from a Reservation function.

- Determining effective and attractive room type mix

  The Reservation or Guest function supplies information about relative demand for each room type and its appeal. This information will help determine room upgrades and possible conversion of some rooms to another room type.

The hotel can retrieve all of this information from the data gathered in the reservation system by summarizing combinations of data elements from different functions, deriving new values from other data items, or simply displaying individual items directly from the database itself.

The routine for reserving a room at the Overnite Hotel identifies some of the data items collected by the hotel. When a guest reserves a room, a transaction takes place. This transaction collects several pieces of data. These data items support the previously listed tasks.

With the new system, the reservation process involves the following steps:

1. A guest calls the Overnite Hotel to reserve a room on a specific date.
2. The hotel requests such information as:

   - Date
   - Name
   - Address
   - Type of room desired
   - Length of the stay
   - Type of rate to which the guest is entitled

3. The reservation clerk checks the inventory of rooms of the specified type that are available for the specified dates.
4. If an appropriate room is available, the clerk tells the guest the rate.
5. If the rate is acceptable, the hotel confirms the reservation.
6. The clerk creates a record containing information collected from the guest.
7. The clerk starts a billing record to store all charges incurred during the stay.
8. The clerk reserves the hotel room. He marks the room to indicate that it is no longer in the inventory of hotel rooms available during the time of the guest's stay.

You can now identify four functions that make up the reservation system. These functions provide information for the transaction, and the transaction supplies input data to the functions:

- Maintaining an inventory of rooms (HOTEL)
- Tracking guest charges for billing (BILLING)
- Maintaining guest data (GUEST)
- Assigning rooms to guests (RESERVATION)

Figure 1-1 shows the sequence of steps for reserving a room at the hotel in a system flowchart.

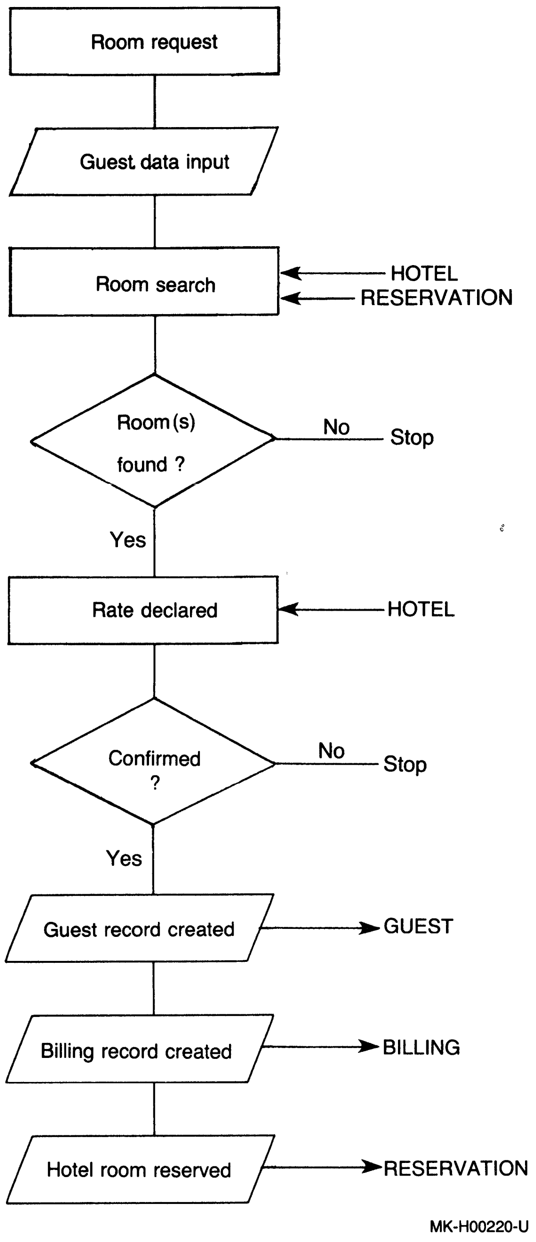


Figure 1-1: Reservation System Flowchart Showing Data Flow

This database model consists of the four basic functions shown in Figure 1-2. The arrows connecting each function indicate relationships and data flow among them.

The Reservation function collects most of the data about the reservation transaction, but also collects data for the Billing and Guest functions. Likewise, the Billing function accumulates data about guest charges and shares it with the Guest function when the total bill is computed. The Hotel function supplies data to the Reservation and Guest functions about the rooms available and their attributes.

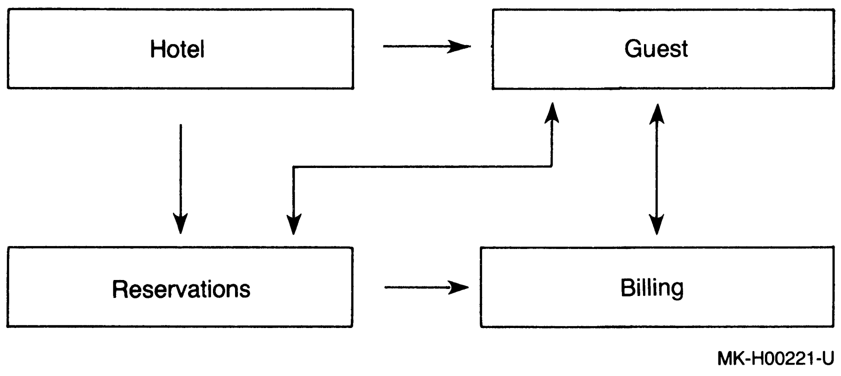


Figure 1-2: Functions Share Data

Each function is responsible for collecting data about its primary object.

- The Hotel function

  Maintains data about the rooms in the hotel.

- The Reservation function

  Records the transaction that sells a room to a guest for specified dates.

- The Billing function

  Accumulates internal transactions about hotel services a guest receives during the stay as well as the cost of the room.

- The Guest function

  Brings together data about the guest for billing and marketing information.

Although the hotel can identify many objects about which it needs to keep information, such as staffing, function rooms, and auxilary services, these four primary functions serve the hotel's reservation system.

1.3.2 Listing Objects

Each function in the hotel maintains data about its primary object, and each function can share data items from other objects. Figure 1-3 shows the objects the reservation system needs to carry out its tasks.

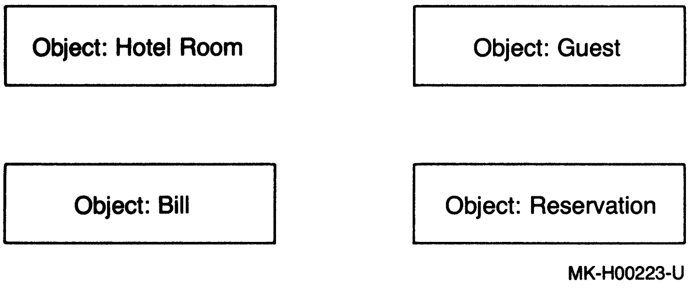


Figure 1-3: Objects in the Hotel Reservation System

1.3.3 Listing Data Items

For each object, the Reservation system collects a number of facts, either by recording them in one function or by gathering them from other functions. After grouping these facts together into some logical relationships, they might look like those in Figure 1-4. These first groupings in Figure 1-4 need not be in their final form. Further testing and refinement might indicate regrouping, eliminating, or adding some items.

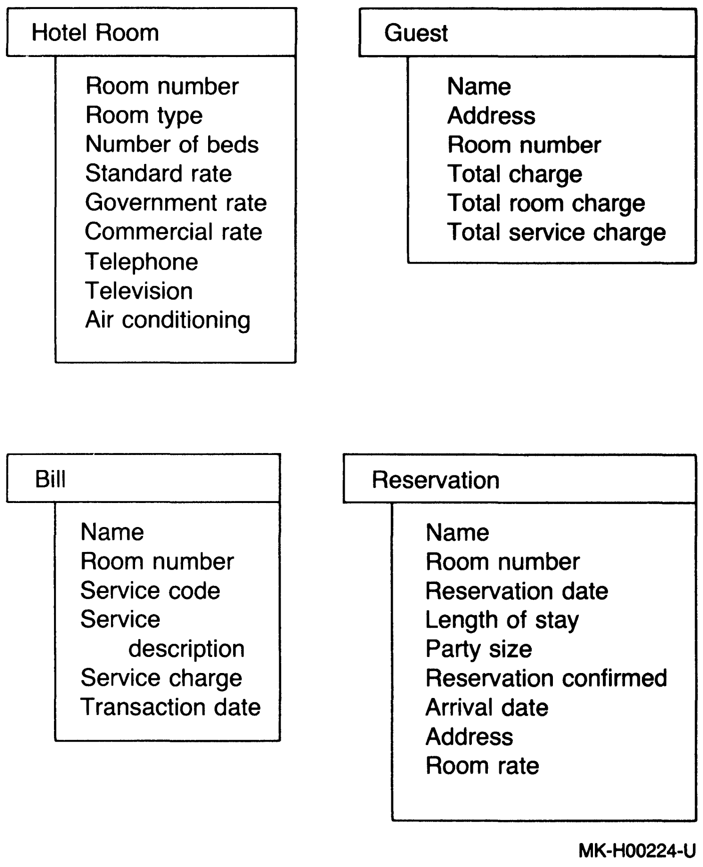


Figure 1-4: Data Items Logically Related by Object

These logical relationships of data correspond to the relations of an Rdb/VMS database. A relation is a set of related data that consists of rows and columns. The columns divide each row (or record) into a set of fields. For a single field in a row, there is only one data item.

For example, the Bill function becomes the BILLING relation. The elements listed under the Bill function become the fields in the relation. The Room number element becomes the ROOM_NUMBER field. Each billing transaction is a record in the database.

1.3.4 Normalizing Data

Once the hotel establishes the logical relationships among the data elements, the database model allows simple information retrieval and update. By normalizing your database, you can benefit from improved performance, efficient data storage, and update consistency. The following steps show you how to refine your database model to enjoy these advantages.

- Eliminate repeating fields.
- Identify primary key fields.
- Check field dependencies.
- Insert foreign keys

Normalizing the Hotel database model results in further changes to the current relation definitions. Such factors as the type of users accessing the database, the number of relations already defined, and the applications that use the database affect the degree to which you normalize your database.

1.3.4.1 Eliminating Repeating Fields -- One step in normalization is to examine the record for possible inefficiencies in the way data is stored and updated. Because the room rate in the HOTEL relation can have three possible values, depending on the type of guest (standard rate, government rate, or group rate), the rate field is really a list of values. Such field types can be efficiently stored in a separate relation and linked to any room number by a foreign key, in this case, room type.

If the hotel keeps three room rates in each of the 400 room records, it must store 400 rooms times three room rates, or about 1200 numeric values. Much of the room rate data is redundant. That is, it is duplicated many times for each room type. These fields are removed from HOTEL in Figure 1-5.

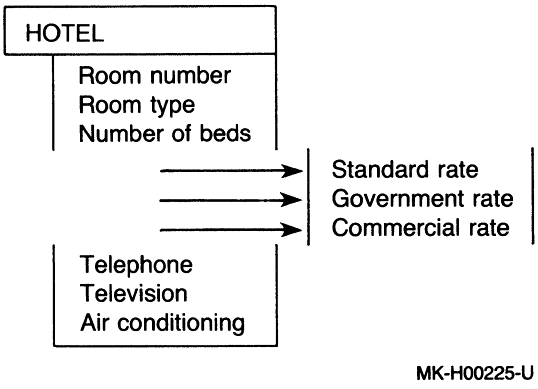


Figure 1-5: Removing Repeating Fields from the HOTEL Relation

Because each of the three room types has a specific room rate, only three records are actually needed to provide the necessary room rate information. Table 1-4 shows sample rates for each of the three room types.

Table 1-4: Three Possible Records in the RATES Relation

Room Type	Standard Rate	Group Rate	Government Rate
1	$50.00	$40.00	$45.00
2	$60.00	$50.00	$55.00
3	$70.00	$60.00	$65.00

The hotel can remove room rate data from the HOTEL relation and create a new relation called RATES, containing three records and a foreign key field called Rate code linking it to the HOTEL relation.

Substitute a Rate code in the HOTEL relation for the list of rate values for each Room type. The Rate code then serves as the link between the HOTEL and RATES relations as illustrated in Figure 1-6.

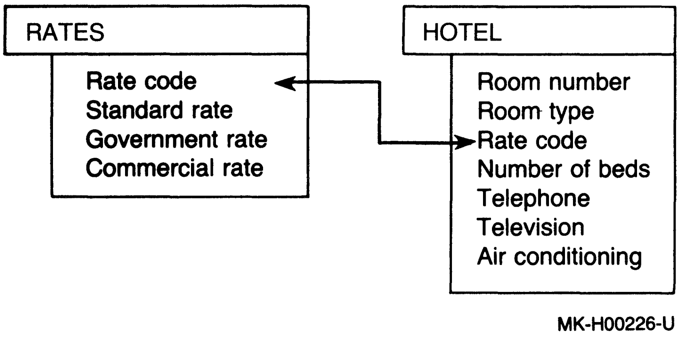


Figure 1-6: Normalizing the HOTEL Relation Creates the RATES Relation

Building two relations from one provides an increase in efficiency. Each record in the RATES relation corresponds to one set of room rates for a room type. When the hotel chooses to change the rates, it at most changes only the three records in the RATES relation instead of many records in the HOTEL relation. Creating two relations from one might appear to require more storage space, but, in fact, it requires less space.

1.3.5 Identifying Primary Keys

Every record you store in the database has at least one field that you can use to locate a single record. Such a field is called a *primary key.* A primary key must have certain features that allow it to locate one record from all of the records in the database. Two very important characteristics of a primary key follow.

A primary key field:

- Must not contain duplicate values

  When the primary key field in each record contains a unique value, you can always use it to locate a single record in the database. You can ensure that the fields you designate as primary key fields have unique values using the DUPLICATES ARE NOT ALLOWED clause of the DEFINE INDEX statement described in Chapter 2.

- Must not contain null values

  Rdb/VMS determines relationships among different fields in a database when you supply the fields and their values at query time. Because the relationships are value-based, Rdb/VMS cannot determine a relationship based on a nonexistent value. That is, if the primary key field value is allowed to be missing, its relationship to the rest of the fields in the record is unknown.

Each record in the HOTEL relation contains two fields: room number and room type. To locate a single record in this relation, you need only the room number. For every room number there is only one room. Therefore, room number uniquely identifies one record and is the primary key for the HOTEL relation. Figure 1-7 shows key fields in both the HOTEL and GUEST relations.

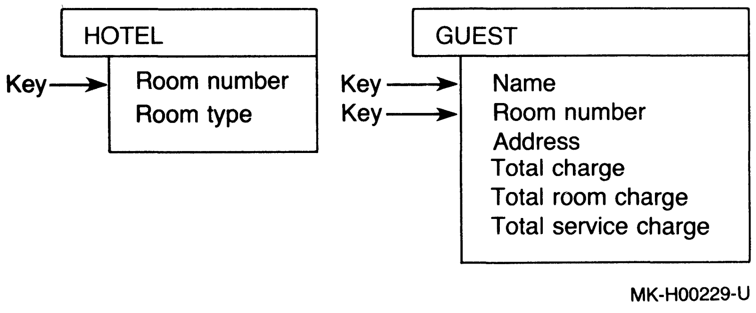


Figure 1-7: Identifying Keys in the HOTEL and GUEST Relations

Sometimes you can identify a single record only by combining two or more fields in a relation to specify a unique key value. Locating a record in the GUEST relation works this way. This relation contains six fields: Name, Room number, Address, and three totals fields. You cannot use the Name field alone to locate a single guest record, because another guest staying in the hotel might have the same name.

One solution is to consider another field along with the Name field to serve as the key. A good candidate is Room number because only one party can stay in any one room at a given time. Searching for "Smith" in room 214, for example, is likely to locate a single record. This solution depends on the assumption that, once a guest checks out of the hotel, all records belonging to that guest are archived or erased. Otherwise, if Smith uses the Overnite Hotel often and likes to stay in his favorite room, records from previous visits will show up in searches for the current record.

The Room number field, already included in another relation, is now used as part of the key in the GUEST relation too. The whole key for the GUEST relation is the combination of Name and Room number. Depending on the type of query, you can use fields other than key fields to locate specific records in a relation.

Figure 1-8 shows that similar analyses identify keys in the BILLING and RESERVATION relations. Using the Name, Transaction date, and Room number fields in the BILLING relation locates *all* of the records belonging to a specific guest. This information is needed to compute the total service charge for the guest's bill.

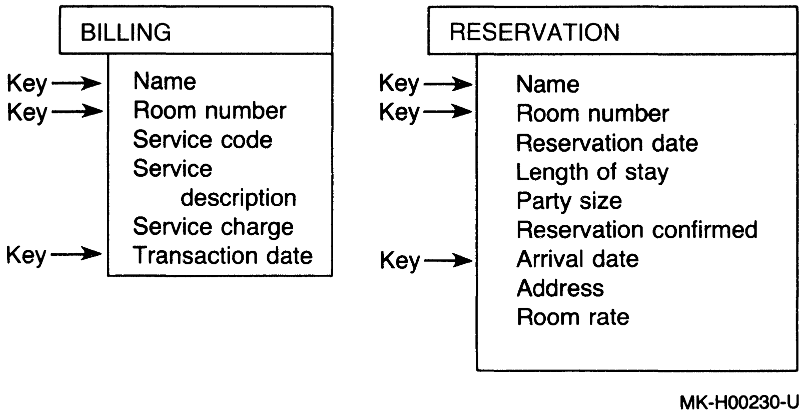


Figure 1-8: Identifying Keys in the BILLING and RESERVATION Relations

You can find records belonging to a guest name, but if more than one guest has the same name, adding the room number to the key locates a specific guest in the hotel. Adding Arrival date pinpoints the day that a particular guest will begin occupying the room.

1.3.5.1 Checking Dependencies on the Key Field -- Each field in a relation should depend on the key field for its meaning. For example, the number of beds in a room of the hotel depends on the room type assigned to that room, not on the room number itself. Similarly, the rate code varies according to the type of each room rather than the room number. Therefore, you can remove these fields from the HOTEL relation and create a new relation, TYPES, that actually holds only one record for each of the three room types in the hotel. The new TYPES relation appears in Figure 1-9.

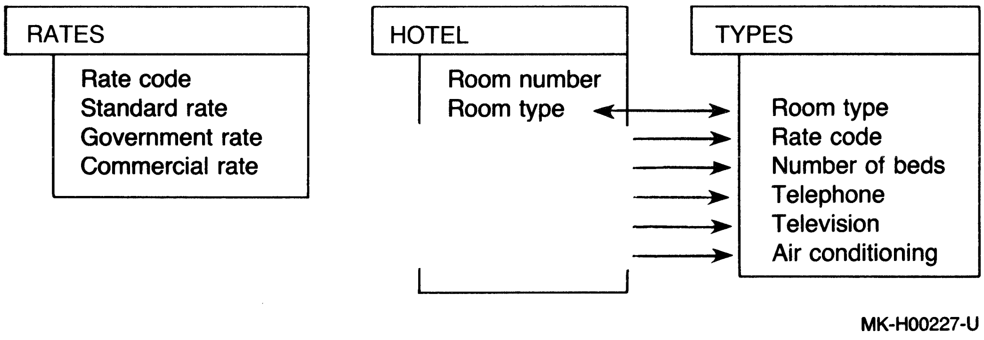


Figure 1-9: Normalizing the HOTEL Relation Creates New Relations

Because each room type has its own rate schedule, removing the Rate code from the HOTEL relation and including it in the TYPES relation further reduces storage space. The original HOTEL relation, then, now becomes the three relations shown in Figure 1-10.

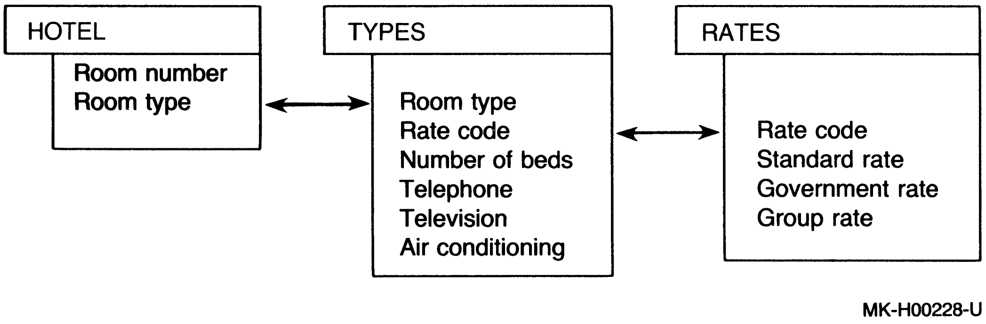


Figure 1-10: Normalized HOTEL Relation

The Room type field appears in both the HOTEL and TYPES relations and provides the link between the two. Similarly, the Rate code field provides the link between the TYPES and RATES relations.

Again, three relations result in increased accuracy, reduced storage space, and efficient and consistent updates. Each record has one field as a primary key that uniquely locates one specific record.

1.3.5.2 Inserting Foreign Keys -- After you have developed a normalized set of logical relations for your database, you should ensure that links exist among them. Each logical relation should have a primary key field (or fields) and each field should contain a single-valued fact about that key. Select two relations that have such a link. Include the name of the primary key of one relation as the foreign key in the second relation. For example, the primary key in the TYPES relation is Room type. To create a link between the TYPES relation and the HOTEL relation, include the Room type in the HOTEL relation as the foreign key.

When you have identified all foreign keys, you should define *indexes* for them. Indexes allow Rdb/VMS to locate individual records directly rather than sequentially. Your index definition can include the DUPLICATES ALLOWED clause. The field definition can include a VALID IF NOT MISSING clause so that all foreign key fields will contain a value. Furthermore, because the foreign key links one relation with another, you might want to ensure that, for every value stored in a foreign key field of one relation, there is a matching value in a primary key field in another relation. You can do this by defining a *constraint* that causes Rdb/VMS to check any new foreign key values against the existing primary key values in the other relation before allowing the value to be stored in the database.

Refer to Chapter 2 for descriptions and examples of defining indexes and constraints.

1.3.6 Defining Fields

You can now formally identify the data items you named in previous steps as fields in your database model. A field is a data item with a name and a specific data type.

Field definitions require at least three basic elements:

- Field name
- Field data type
- Field size

1.3.6.1 Defining Field Names -- Before the hotel can define a relation, each data item, or field, needs a name. Choosing a name for each field is important, because all users, procedures, and programs call that field by the name it has in the database. Unlike traditional applications, which often create different names for the same data elements, the name of a field in the database is the field's only label. To be useful, field names, like other database entities, should be meaningful and tell as much as possible about the values or facts they represent.

Table 1-5 shows how you can derive field names from your planning information.

Table 1-5: Data Items Become Field Names

Data Item	Field Name
Room Number	ROOM_NUMBER
Room Type	ROOM_TYPE
Number of Beds	BEDS
Standard Rate	STANDARD_RATE
Government Rate	GOV_RATE
Group Rate	GROUP_RATE
Name	NAME
Service Code	SERVICE_CODE
Service Charge	SERVICE_CHARGE
Transaction Date	TX_DATE
Arrival Date	ARRIVAL_DATE
Address	ADDRESS
Number in Party	PARTY_SIZE
Reservation Date	RESERVE_DATE
Total Charge	TOTAL_CHARGE
Length of Stay	LENGTH_OF_STAY

1.3.6.2 Defining Field Data Types -- Each field in the database can contain only one type of data. Data types include TEXT, NUMERIC, SIGNED WORD, and DATE. For example, guest names consist of letters, so the data type for the GUEST_NAME field is TEXT. The number of beds is always recorded as digits, so the data type for the BEDS field is SIGNED WORD.

1.3.6.3 Defining Field Sizes -- Finally, each field must have a size limit. This characteristic specifies the number of characters that are needed to hold all possible values adequately. Although GUEST_NAME could mean the guest's first and last names as well as middle initial, the hotel might only require this field to hold the guest's last name. A quick examination of past guest records reveals that no guest's last name was longer than 15 letters. Should a guest arrive whose last name has more than 15 letters, the first 15 are more than enough to identify the person.

Although Rdb/VMS uses data compression to permit efficient storage in the database of repeating characters of field values, specifying a field size that is too large results in moving many blank characters from the database to programs that can waste storage space. Thus, the field labeled GUEST_NAME is given an adequate field size of 15 characters.

1.3.6.4 Field Definition Examples -- For the GUEST_NAME field, the full description includes the following information:

- Name: GUEST_NAME
- Data type: TEXT
- Size: 15

This information is all you need to specify the field definition for GUEST_NAME.

Table 1-6 lists the field names with their data types, sizes, and sample values.

Table 1-6: Field Names Described with Sample Data Values

Field Name	Data Type	Size	Sample Value
ROOM_NUMBER	Numeric	3	207
ROOM_TYPE	Text	2	S
BEDS	Numeric	1	2
GUEST_NAME	Text	15	Smith
SERVICE_CODE	Text	2	BR(eakfast)
SERVICE_CHARGE	Numeric	6	27.50
TX_DATE	Date	11	27-JUL-1983
ADDRESS	Text	25	Boston, MA
ARRIVE_DATE	Date	11	29-JUL-1983
RESERVE_DATE	Date	11	27-JUL-1983
PART_SIZE	Numeric	1	3
ROOM_RATE	Numeric	6	48.50
SERVICE_DESCR	Text	20	Room Service
RATE_CODE	Text	1	C
GROUP_RATE	Numeric	5	32.50
GOV_RATE	Numeric	5	29.95
STD_RATE	Numeric	5	36.50
TELEPHONE	Text	1	Y(es)
TV	Text	1	N(o)
AC	Text	1	Y(es)

1.3.7 Defining Relations

Once you define all of the functions of the reservation system, each function becomes a **relation**, that is, a logical group of data items. The name of the relation can be the name of the function. The relation includes all fields necessary to make it complete and meaningful. The process of normalization helps to ensure that there is as little repetition of fields as possible and that updating the database is consistent and direct.

Figure 1-11 shows the relations that make up the reservation system for the Overnite Hotel. Key fields are marked with asterisks (*).

HOTEL
ROOM_NUMBER *
ROOM_TYPE

RATES
RATE_CODE *
STD_RATE
GOV_RATE
GROUP_RATE

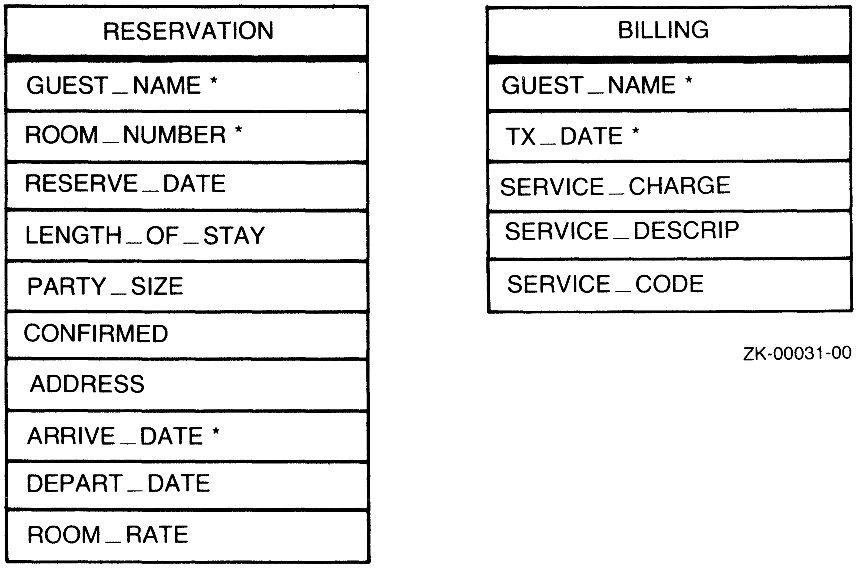
TYPES
ROOM_TYPE *
RATE_CODE
BEDS
TELEPHONE
TV
AC

GUEST
GUEST_NAME *
ROOM_NUMBER *
ADDRESS
TOTAL_CHARGE
TOTAL_ROOM_CHARGE
TOTAL_SERVICE_CHARGE

* = Key fields

(continued on next page)

Figure 1-11: Relations in the Overnite Hotel Reservation System



* = Key fields

Figure 1-11: Relations in the Overnite Hotel Reservation System (Cont.)

An informal inspection of the GUEST relation shows that three fields are already contained in the RESERVATION relation. The other three fields can be computed from information contained in the BILLING relation. In cases where data items are redundant, you might want to use another Rdb/VMS feature, the view, instead of a relation.

Views use fields that already exist in other relations, or create special new fields containing computed values from existing fields. Because views do not themselves store actual values, a view can save storage space. Another advantage of using views is security. You can create views that allow users to see only portions of the data stored in relations.

Views are especially useful in helping end users access parts of several different relations without having to issue complex queries repeatedly. In these cases the view definitions are based on complex record selection expressions (RSEs). For information on creating view, see Chapter 2, Section 2.8.

Creating a Database 2

This chapter shows you how to define the entities of a typical database, except protection. Defining protection is explained in Chapter 3. The elements discussed here include:

- The **database** itself, including the database files and the storage requirements
- The characteristics of the **fields** that make up the database's relations
- **Relations**, which combine fields into logical units
- **Views**, which combine data from one or more relations into "virtual" relations
- **Constraints**, which establish the limits for field values

In this chapter, you learn how to use RDO statements to define the database. There are three ways you can enter the RDO statements:

- Use an editor to create a command file that has an RDO file type (for example, HOTELCOM.RDO). Such a command file can contain all the definition statements required to create the database. This method is efficient if you know that there are no problems with the database definitions.

  You can execute the RDO command procedure at the RDO> prompt. Simply type an at sign (@) followed by the name of the command procedure file:

  ```
  RDO> @REPORT
  ```

- Run RDO and use the EDIT statement editor within RDO. Enter the statements in the editing buffer. If you use this method, you can enter the statements one at a time and check each one for successful execution. If a statement fails, you can simply type EDIT to correct your errors. This method is useful if you are less familiar with the syntax of the statements.
- You can type the definition statements directly at the RDO> prompt. This method is perhaps the least useful because it is harder to verify and correct the statements you have typed.

2.1 The Logical Database

Chapter 1 describes how to define a logical database by determining:

- The necessary data items
- The characteristics of these data items, including the type of data (numeric, text) and the range of values
- How the data items can be divided into relations

After you finish this process you have a model of your database in the form of logical relations. Figure 1-11 contains the logical model for the OVERNITE database.

Each of the following sections uses these relations as an example to show how the OVERNITE database might be built in VAX Rdb/VMS. Complete Rdb/VMS definitions for the database appear in Appendix A.

2.2 Storage of Database Entity Definitions

There are two options for storing database entity definitions:

- In the database only
- In both the database and the VAX Common Data Dictionary (CDD)

If you plan to use VAX DATATRIEVE or any other VAX information management product for your database access tasks, you must include the database definitions in the CDD by specifying a path name.

When you store definitions for fields, relations, views, and constraints in the database only, you specify a file specification in the DEFINE DATABASE statement (see Section 2.3).

If the Common Data Dictionary is not installed on your system, Rdb/VMS stores data definitions only in the database file. However, Rdb/VMS automatically stores the name of the database in the CDD if the CDD is installed.

Storing database definitions in the CDD provides a central source of definitions and allows you to use other VAX information management products with your Rdb/VMS database. To avoid data definition inconsistencies, you should always invoke the database using the CDD path name. In this way, the database definitions are always available to other DIGITAL products that use the CDD. For example, host language programs containing embedded Rdb/VMS data manipulation statements can copy record definitions from the CDD with corresponding compatible data types. Then, whenever data definitions change, the host language programs require little or no modification.

2.3 Defining the Database

The first step in defining a database is to allocate resources using the DEFINE DATABASE statement. This statement performs the following operations:

- Names the database
- Creates a database file
- Creates a snapshot file
- Creates a directory in the Common Data Dictionary if the CDD is installed on your system
- Allows you to determine the physical storage parameters for the database file or to use adequate default values

2.3.1 Naming the Database Files

By default, the name of the database determines the names of the database file, snapshot file, and CDD directory for your database. Assume that the current default directory is DISK2:[BOOKKEEP] and the current default CDD directory is CDD$TOP.BOOKKEEP. The shortest form for the DEFINE DATABASE statement is:

```
DEFINE DATABASE "OVERNITE".
```

Note that all DEFINE statements must end with a period. Also, you should put quotation marks around the database name and the CDD path name whenever you use DEFINE or INVOKE DATABASE statements in RDO.

This DEFINE DATABASE statement creates the following entities:

- DISK2:[BOOKKEEP]OVERNITE.RDB -- the database file where Rdb/VMS stores database definitions and data.

- DISK2:[BOOKKEEP]OVERNITE.SNP -- the snapshot file where Rdb/VMS stores certain versions of records in the database. This file is used by READ_ONLY transactions.
- CDD$TOP.BOOKKEEP.OVERNITE -- the CDD directory where Rdb/VMS stores copies of data definitions if the CDD is installed on your system.

If you want Rdb/VMS to create your database in a directory other than the current default VMS directory, use a full file specification for the database name. For example, suppose the current default directory is DISK2:[WORK] and the current default CDD directory is CDD$TOP.BOOKKEEP. The following statement uses an expanded file specification to store the database in another VMS directory:

```
DEFINE DATABASE "DISK2:[DEPT4.ACCOUNT]OVERNITE".
```

This statement creates the following entities:

- DISK2:[DEPT4.ACCOUNT]OVERNITE.RDB - the database file
- DISK2:[DEPT4.ACCOUNT]OVERNITE.SNP - the snapshot file
- CDD$TOP.BOOKKEEP.OVERNITE - the CDD directory

By default, Rdb/VMS stores the database definitions in the current default CDD directory. You determine this directory either by defining the logical name CDD$DEFAULT or by explicitly naming a dictionary using the SET DICTIONARY statement in RDO.

If you use an RDO command file to define your database in other VMS directories, the default CDD directory might be different each time you execute the command file. You can use the SET DICTIONARY path-name statement in the command file to name the CDD directory explicitly. This statement prevents the command file from depending on the CDD default of the process that invokes it.

The IN path-name clause or the DEFINE DATABASE statement causes Rdb/VMS to store definitions in the specified CDD directory. If the CDD directory does not exist, Rdb/VMS creates it. Assume your VMS default directory is DISK2:[BOOKKEEP.TEST] and your CDD default directory is CDD$TOP.BOOKKEEP. The following statement shows how to use the IN clause to specify a different CDD directory for this test database:

```
DEFINE DATABASE "OVERNITE" IN "CDD$TOP.BOOKKEEP.TEST".
```

This statement creates the following entities:

- DISK2:[BOOKKEEP.TEST]OVERNITE.RDB -- the test database file
- DISK2:[BOOKKEEP.TEST]OVERNITE.SNP -- the test snapshot file
- CDD$TOP.BOOKKEEP.TEST -- the CDD directory

When you use this test database for data manipulation, you can invoke it even if your default CDD or VMS directories use the FILENAME clause:

```
INVOKE DATABASE FILENAME "DISK2:[BOOKKEEP.TEST]OVERNITE"
```

If you intend to change the definitions of the test database using a CHANGE, DELETE, or DEFINE statement, you should always invoke the database with the PATHNAME clause:

```
INVOKE DATABASE PATHNAME "CDD$TOP.BOOKKEEP.TEST"
```

In this case, Rdb/VMS finds the correct database file name by checking the CDD definition for the database. Any changes you make to data definitions are then entered in the CDD.

2.3.2 Database Size

Several clauses of the DEFINE DATABASE statement let you determine how your database uses mass storage and memory. In most cases, the default settings for these parameters are adequate. Furthermore, Rdb/VMS includes the multidisk database capability. You can distribute a large database across several disks and let Rdb/VMS maintain how and where new database growth is placed among the disks. The *VAX Rdb/VMS Guide to Database Administration and Maintenance* explains how to determine the values to specify for many of the DEFINE DATABASE parameters.

If you do not specify any size parameters with the DEFINE DATABASE statement, Rdb/VMS uses the following defaults:

- Number of database pages -- 400
- Number of page blocks -- 2 (1024 bytes)
- Number of users -- 50
- Number of buffer blocks -- 6 (three times the number of page blocks)

2.4 Defining Fields

A VAX Rdb/VMS database consists of a set of one or more relations. A relation definition simply gives a name to a list of field definitions.

There are two ways to define a field for an Rdb/VMS relation:

- With a DEFINE FIELD statement

  The DEFINE FIELD statement adds a field definition to the database file and to the CDD when you use the INVOKE DATABASE PATHNAME statement. Once you have defined the field, you can include it in any relation definition simply by naming it. This is the *recommended* method of defining a field.

- Inside a DEFINE RELATION statement

  You can define a field within a relation definition by naming it and specifying its characteristics.

A field definition consists of a series of field attributes. Attributes can be global or local.

2.4.1 Global Attributes

Global attributes are associated with a global field name. You can include a global field name in any relation definition. When a relation refers to a global field name, the named field in the relation carries with it all the global attributes of the field. Global attributes are:

- Global name (required)
- Data type (required)
- VALID IF clause (optional)
- MISSING_VALUE clause (optional)
- DATATRIEVE support clauses (optional)

When you include any one of these global attributes as part of a definition, whether you define the field with a DEFINE FIELD statement or within a DEFINE RELATION statement, you are defining the field globally. Rdb/VMS checks the list of global definitions for the database. If no existing field has the same name, the new field definition is added to the list of global fields.

2.4.1.1 Global Name -- A global name must be unique among field names in the database. Once you define the field, any relation can refer to the field definition by this name. For example, the following field definition establishes ROOM_NUMBER as a global field name:

```
DEFINE FIELD  ROOM_NUMBER
            DESCRIPTION IS
           /* (Primary key for HOTEL) Hotel room number */
                      DATATYPE IS TEXT SIZE IS 3
                       VALID IF
                       (ROOM_NUMBER GT "100" AND
                       ROOM_NUMBER LT "500" AND
                       ROOM_NUMBER NE "200" AND
                       ROOM_NUMBER NE "300" AND
                       ROOM_NUMBER NE "400") AND
                       ROOM_NUMBER NOT MISSING
                        EDIT_STRING FOR DATATRIEVE IS "XXX".
```

Now the HOTEL, RESERVATION, and BILLING relations can include the global field ROOM_NUMBER. ROOM_NUMBER will have the same name and attributes in all three relations.

```
DEFINE RELATION HOTEL.
     ROOM_NUMBER.
     ROOM_TYPE.
END HOTEL RELATION.
```

Note that the global name attribute is required for all global fields.

2.4.1.2 Data Type -- Rdb/VMS uses a number of data types. These include signed integers, floating point numbers, dates, and ASCII text. See Table 2-1 for the complete list of VAX Rdb/VMS data types.

Note that all global field definitions must include data types.

2.4.1.3 VALID IF Clause -- The VALID IF clause, which is optional, specifies a domain of values for that particular field. Any value that you intend to add for that field must lie within that domain in order to be stored in the database. The VALID IF clause is used to check that a value is within a specified range or that it exactly matches a list of values. When you specify a VALID IF clause for a global field, you ensure that all values assigned to that field in every applicable relation are checked consistently.

For example, you can add a VALID IF clause to the ROOM_TYPE field to ensure that only specific values are assigned to it. Because the HOTEL and TYPES relations include both the ROOM_TYPE field, Rdb/VMS checks values for ROOM_TYPE in both relations.

```
DEFINE FIELD  ROOM_TYPE
            DESCRIPTION IS /* Hotel room type code  */
                      DATATYPE IS TEXT SIZE IS 2
                       VALID IF
                            ROOM_TYPE EQ "S"
                         OR ROOM_TYPE EQ "D"
                         OR ROOM_TYPE EQ "SS"
                         OR ROOM_TYPE MISSING
                         MISSING_VALUE IS "??"
                         EDIT_STRING FOR DATATRIEVE IS "XX".
```

Remember that VALID IF clauses are optional.

2.4.1.4 MISSING_VALUE Clause -- A missing value allows you to account for fields in which no explicit value is stored. When you do not store a specific value in a field, or you store the value defined as the missing value, Rdb/VMS marks this field value as missing. The MISSING operator is used because the value in the field is unknown and cannot be used in relational comparisons. Rdb/VMS also ignores missing values when calculating aggregates. Refer to Chapter 3 of the *VAX Rdb/VMS Reference Manual* for more details on missing values.

When you identify a primary key for each relation, you can ensure that it never contains null values by including the VALID IF NOT MISSING clause in the DEFINE FIELD statement.

The following example uses November 18, 1858 as the missing value for the arrival date field:

```
DEFINE FIELD ARRIVE_DATE
         DESCRIPTION IS /* Date of arrival */
         DATATYPE IS DATE
         MISSING_VALUE IS
         "18-NOV-1858 00:00:00:00".
```

To find records with missing value fields, use the MISSING operator:

```
FOR R IN RESERVATION WITH R.ARRIVE_DATE MISSING
   PRINT R.GUEST_NAME,
         R.ROOM_NUMBER,
         R.RESERVE_DATE
END_FOR
```

MISSING VALUE clauses are optional.

2.4.1.5 DATATRIEVE Support Clauses -- If you intend to access the database with VAX DATATRIEVE, you might want to specify DATATRIEVE clauses, such as a default value and an edit string. DATATRIEVE uses these characteristics when retrieving the value from the Rdb/VMS database. For example, if no value has been stored in a field, DATATRIEVE displays the default value on a PRINT statement. For more details, see the *VAX DATATRIEVE Reference Manual.* Note that DATATRIEVE support clauses are optional.

2.4.2 Local Attributes

Local attributes are defined *only* within a DEFINE RELATION statement and apply only to that relation's version of the field definition. Local attributes are:

- Local field name, when you use the BASED ON clause
- DATATRIEVE support clauses
- COMPUTED BY clause

2.4.2.1 Local Field Names -- The local field name allows you to give a name to a field that is recognized only within the relation. When you use the BASED ON clause, the field name is not entered in the list of field names for the database, and other relations cannot refer to the field definition by that name.

You might want to assign local field names because these names are only needed by persons accessing the relation through DATATRIEVE. Assigning the attribute to the global field definition would be unnecessary.

The following relation definition shows the BASED ON clause for two fields, SERVICE_CHARGE and TX_DATE. The other fields have global names.

```
DEFINE RELATION BILLING.

ROOM_NUMBER.
SERVICE_CHARGE
       BASED ON STANDARD_RATE
       QUERY_HEADER FOR DATATRIEVE IS "SERVICE"/"CHARGE"
         QUERY_NAME FOR DATATRIEVE IS "S_CHG".
TX_DATE
       BASED ON STANDARD_DATE
       QUERY_HEADER FOR DATATRIEVE
            IS "TRANSACTION"/"DATE".
SERVICE_DESCRIP.
SERVICE_CODE.

END BILLING RELATION.
```

2.4.2.2 Local DATATRIEVE Support Clauses -- If you supply a DATATRIEVE support clause as a local attribute, it overrides the global DATATRIEVE support clause for that field.

2.4.2.3 COMPUTED BY Clause -- The COMPUTED BY clause allows you to name a field containing a value that represents the result of a value expression. For example, if the GOV_RATE for a room is 10 percent less than the standard room rate and the GROUP_RATE is 14 percent less than the standard rate, you can define GOV_RATE and GROUP_RATE fields like this:

```
DEFINE RELATION RATES.

RATE_CODE.
STD_RATE BASED ON STANDARD_RATE
      QUERY_HEADER FOR DATATRIEVE IS "STANDARD"/"RATE"
      QUERY_NAME FOR DATATRIEVE IS "ST_RATE".

GOV_RATE
      COMPUTED BY (STD_RATE * 0.90)
      QUERY_HEADER FOR DATATRIEVE IS "GOVERNMENT"/"RATE"
      QUERY_NAME FOR DATATRIEVE IS "G_RATE".

GROUP_RATE
      COMPUTED BY (STD_RATE * 0.86)
      QUERY_HEADER FOR DATATRIEVE IS "GROUP"/"RATE"
      QUERY_NAME FOR DATATRIEVE IS "GRP_RATE".

END RATES RELATION.
```

Using this type of definition means that you have to store values only in the STD_RATE field. The GOV_RATE and GROUP_RATE fields are computed automatically.

If you want to use a floating point number in a COMPUTED BY clause, you must always have at least one digit before the decimal point and one digit after the decimal point. Otherwise, Rdb/VMS interprets the decimal point as a period that terminates the data definition statement.

2.4.3 Using the DEFINE FIELD Statement

You should use the DEFINE FIELD statement to set up definitions for all the data items in the database before issuing any DEFINE RELATION statements. This procedure simplifies the defining of relations by letting you keep a centralized list of global field definitions. Then your relation definitions can simply list the names of the global fields, or you can customize the relation definitions by using local names and local attributes.

Fields that are defined with DEFINE FIELD statements are global fields. The field can be included in any relation. If you should delete a relation, the global fields associated with that relation remain in the database.

When you use global fields, any changes that need to be made in the field definition are made only once. All relations that include that global field automatically reflect the change. For example, if the hotel put on a huge addition, it might need to change the definition of the ROOM_NUMBER field to accommodate 4-digit room numbers. A single change to the field definition would immediately affect the HOTEL, RESERVATION, and BILLING relations.

Once you define a global field with a DEFINE FIELD statement, that field exists as an entity in the database. If you define a relation based on global fields and for some reason the relation definition fails, the global field definitions remain in tact. If you define all your fields locally through a DEFINE RELATION statement and the transaction fails to commit, all work is lost.

Of course, there are reasons to define fields locally as part of the DEFINE RELATION statement. But wherever possible, you should consider using the DEFINE FIELD statement to create global fields.

2.4.4 Creating Field Names

Because you are defining global fields that can be used in more than one relation, you should make field names as general as possible. For example, there are several fields that contain date values. These fields use identical definitions. The DEFINE FIELD statement, then, might give these fields a generic name, like STANDARD_DATE:

```
DEFINE FIELD  STANDARD_DATE
      DESCRIPTION IS /* Standard date field  */
                DATATYPE IS DATE
                 MISSING_VALUE IS "18-NOV-1858 00:00:00.00"
                 EDIT_STRING FOR DATATRIEVE IS "MM/DD/YY".
```

When you define the relation itself, you can use the BASED ON clause and give the fields local names. The global definition is still in effect. The following example shows the list of field names that constitute the RESERVATION relation. The global field definitions appear elsewhere in the database definition file.

The following local field definitions in the RESERVATION relation include BASED ON clauses:

```
DEFINE RELATION RESERVATION.
   GUEST_NAME.
   CITY.
   STATE.
   POSTAL_CODE.
   ROOM_NUMBER.
   LENGTH_OF_STAY.
   PARTY_SIZE.
   RESERVE_DATE
           BASED ON STANDARD_DATE
           QUERY_HEADER FOR DATATRIEVE IS "RESERVATION"/"DATE"
           QUERY_NAME FOR DATATRIEVE IS "RESRV_DATE".
   ARRIVE_DATE
           BASED ON STANDARD_DATE
           QUERY_HEADER FOR DATATRIEVE IS "ARRIVAL"/"DATE"
           QUERY_NAME FOR DATATRIEVE IS "A_DATE".
   DEPART_DATE
           BASED ON STANDARD_DATE
           QUERY_HEADER FOR DATATRIEVE IS "DEPARTURE"/"DATE"
           QUERY_NAME FOR DATATRIEVE IS "D_DATE".
   CONFIRMED
         BASED ON STANDARD_FLAG
         QUERY_HEADER FOR DATATRIEVE IS "RESERVATION"/"CONFIRMED"
         QUERY_NAME FOR DATATRIEVE IS "RESRV_CONF".
   CHECK_OUT
         BASED ON STANDARD_FLAG
         QUERY_HEADER FOR DATATRIEVE IS "CHECKED"/"OUT"
         QUERY_NAME FOR DATATRIEVE IS "CHK_OUT".
   ROOM_RATE
         BASED ON STANDARD_RATE
         QUERY_HEADER FOR DATATRIEVE IS "ROOM"/"RATE"
         QUERY_NAME FOR DATATRIEVE IS "R_RAT".
END RESERVATION RELATION.
```

2.4.5 Specifying Data Types

You must specify a data type with each global field you name. Table 2-1 lists the characteristics for each data type.

Table 2-1: Rdb/VMS Data Types

VAX Rdb/VMS Data Type	Corresponding VAX Data Type	Size	Range/ Precision	Other Parameters
SIGNED WORD	Signed word integer	16 bits	–32768 to 32767	n = scale factor
SIGNED LONGWORD	Signed longword integer	32 bits	–2**31 to (2**31)–1	n = scale factor
SIGNED QUADWORD	Signed quadword integer	64 bits	–2**63 to (2**63)–1	n = scale factor
F_FLOATING	F_floating Single precision floating point number	32 bits	Approximately seven decimal digits	None
G_FLOATING	G_floating Extended precision floating point number	64 bits	Approximately 15 decimal digits	None
DATE	Absolute date and time	64 bits	Not applicable	None
TEXT	ASCII text	n bytes	0 to 16383 characters	n = number of characters (unsigned integer)
VARYING STRING	Varying length ASCII text	Varies	0 to 16383 characters	n = maximum number of characters (unsigned integer)
SEGMENTED STRING	None	Varies	0 to 64k bytes per segment	None

The OVERNITE database uses four data types:

- TEXT

  You use the TEXT data type for names and labels. TEXT is also useful for identification numbers that are not used in calculations, for example, room numbers. The size of the field should be sufficient to hold the longest string of text characters.

```
DEFINE FIELD  GUEST_NAME
            DESCRIPTION IS /* Guest name  */
                      DATATYPE IS TEXT SIZE IS 15
                       VALID IF
                           GUEST_NAME NOT MISSING.
```

- DATE

  The VAX DATE data type is a quadword value giving the time since a base date (17-NOV-1858 00:00:00.00). Many VAX languages and utilities, including DATATRIEVE, use the DATE data type for specifying dates.

```
DEFINE FIELD  STANDARD_DATE
            DESCRIPTION IS /* Standard date field  */
                      DATATYPE IS DATE
                       MISSING_VALUE IS "18-NOV-1858 00:00:00.00"
                       EDIT_STRING FOR DATATRIEVE IS "MM/DD/YY".
```

- SIGNED WORD

  The OVERNITE database uses word integers for unscaled numeric information. If the field requires more than four digits, you must use the SIGNED LONGWORD data type.

```
DEFINE FIELD  LENGTH_OF_STAY
            DESCRIPTION IS
         /* Number of days guest stays in hotel  */
                      DATATYPE IS SIGNED WORD
                       VALID IF
                           LENGTH_OF_STAY GT 0
                        OR LENGTH_OF_STAY MISSING
                       MISSING_VALUE IS -1
            QUERY_HEADER FOR DATATRIEVE IS "LENGTH"/"OF STAY"
              QUERY_NAME FOR DATATRIEVE IS "STAY".
```

- SIGNED LONGWORD

  The OVERNITE database uses the SIGNED LONGWORD data type for money values. SIGNED LONGWORD allows for scaling. For example, the SERVICE_CHARGE field might contain a value like $29.95. You should use the SIGNED LONGWORD SCALE -2 data type to store this kind of data.

```
DEFINE FIELD  STANDARD_RATE
            DESCRIPTION IS /* Standard money field  */
                     DATATYPE IS SIGNED LONGWORD SCALE -2
                     EDIT_STRING FOR DATATRIEVE IS "$$$$.$$".



  SERVICE_CHARGE
       BASED ON STANDARD_RATE
       QUERY_HEADER FOR DATATRIEVE IS "SERVICE"/"CHARGE"
       QUERY_NAME FOR DATATRIEVE IS "S_CHG".
```

2.4.6 Field Definition Options

Appendix A shows a procedure that defines all of the OVERNITE fields and relations for the OVERNITE database. Each field definition uses some optional components of the DEFINE FIELD statement. You can add the following optional clauses to the field definitions:

1. DATATRIEVE support clause
2. MISSING_VALUE clause
3. VALID IF clause

2.4.6.1 DATATRIEVE Support Clauses -- You can use DATATRIEVE to display Rdb/VMS data on the terminal and to create reports from Rdb/VMS databases. Therefore, Rdb/VMS lets you define display characteristics for DATATRIEVE in the field definitions.

For example, you might want to format dates in a standard format such as "11/17/83". To do this, you include a DATATRIEVE edit string clause in the STANDARD_DATE field definition:

```
DEFINE FIELD  STANDARD_DATE
            DESCRIPTION IS /* Standard date field  */
                     DATATYPE IS DATE
                      MISSING_VALUE IS "18-NOV-1858 00:00:00.00"
                      EDIT_STRING FOR DATATRIEVE IS "MM/DD/YY".
```

The local fields that depend on the STANDARD_DATE field inherit all the field attributes of the global field. Added to these are the local field attributes that are defined within relations. For example, the BILLING relation defines transaction date in terms of the STANDARD_DATE field and includes a DATATRIEVE support clause.

```
TX_DATE BASED ON STANDARD_DATE
    QUERY_HEADER FOR DATATRIEVE IS
        "TRANSACTION"/"DATE".
```

Local fields in the RESERVATION relation are defined similarly:

```
RESERVE_DATE BASED ON STANDARD_DATE
           QUERY_HEADER FOR DATATRIEVE
                 IS "RESERVATION"/"DATE"
           QUERY_NAME FOR DATATRIEVE
                 IS "RESRV_DATE".
ARRIVE_DATE BASED ON STANDARD_DATE
           QUERY_HEADER FOR DATATRIEVE
                 IS "ARRIVAL"/"DATE"
           QUERY_NAME FOR DATATRIEVE
                 IS "A_DATE".
DEPART_DATE BASED ON STANDARD_DATE
           QUERY_HEADER FOR DATATRIEVE
                 IS "DEPARTURE"/"DATE"
           QUERY_NAME FOR DATATRIEVE
                 IS "D_DATE".
```

The *VAX DATATRIEVE Reference Manual* provides more information on DATATRIEVE edit strings.

2.4.6.2 MISSING_VALUE Clause -- Including a missing value clause in the field definition enables you to handle situations when you do not have the information you need to enter a valid data value. The clause specifies what character(s) you enter to indicate lack of information.

It is important to remember that Rdb/VMS always checks the VALID IF clause in the DEFINE FIELD statement. Therefore, if you have defined any fields for which you *might not* have explicit values, be sure to extend the VALID IF clause to include VALID IF MISSING. Whenever your database is backed up and restored using the RDO BACKUP and RESTORE statements, Rdb/VMS checks all the data when it is reapplied to the database. If no value is available for a specific field in a record, and you have *not* included the VALID IF MISSING clause, Rdb/VMS returns an error; your database could now be inconsistent.

In the OVERNITE database, the field TELEPHONE can have two normal values, "Y" or "N". When the hotel assigns values to the characteristics of a room and information is unavailable about the presence or absence of a telephone, the hotel can ignore this field or store the missing value "?" during data entry. Because no legal value is stored in that field, DATATRIEVE signals Rdb/VMS to flag that field as having the missing value, "?". Notice that the VALID IF clause confirms the fact that it is acceptable and valid for this field to be null.

```
DEFINE FIELD  TELEPHONE
            DESCRIPTION IS /* Is telephone in the hotel room  */
                      DATATYPE IS TEXT SIZE IS 1
                       VALID IF
                           TELEPHONE EQ "Y"
                        OR TELEPHONE EQ "N"
                        OR TELEPHONE MISSING
                       MISSING_VALUE IS "?"
            QUERY_HEADER FOR DATATRIEVE IS "TELEPHONE"
              QUERY_NAME FOR DATATRIEVE IS "PHONE".
```

2.4.6.3 VALID IF Clause -- Several data items are restricted to a range of values. Ranges are best enforced with the VALID IF clause. For example, room numbers in the hotel range from 101 to 499, but they exclude certain values: 200, 300, and 400. If someone tries to enter either a number outside this range, or one of the excluded number values, Rdb/VMS returns an error and prevents the incorrect value from being stored. The VALID IF clause performs this function.

```
DEFINE FIELD  ROOM_NUMBER
            DESCRIPTION IS /* (PK for HOTEL) Hotel room number  */
                      DATATYPE IS TEXT SIZE IS 3
                       VALID IF
                       (ROOM_NUMBER GT "100" AND
                       ROOM_NUMBER LT "500" AND
                       ROOM_NUMBER NE "200" AND
                       ROOM_NUMBER NE "300" AND
                       ROOM_NUMBER NE "400") AND
                       ROOM_NUMBER NOT MISSING
                        EDIT_STRING FOR DATATRIEVE IS "XXX".
```

As mentioned in Section 2.4.6.2, you can also use the VALID IF clause to allow a missing value for the field. For example, a missing value for departure date might be acceptable, but not allowed as an arrival date value. In defining the departure date field, you might want to include a VALID IF MISSING clause.

If your tasks require checking fields in other relations in the database, Rdb/VMS allows you to define a *constraint* for this purpose. Section 2.6 shows you how to use the DEFINE CONSTRAINT statement to add a formal constraint to a field. You can define optional characteristics for all the fields in the sample database. See Appendix A for complete definitions.

2.5 Defining Relations

A relation definition is composed of the following components:

- Field names
- Local field names
- DATATRIEVE support options

The simplest way to define a relation is to list existing field names. If you use this method, you need only choose the name for the relation. You can also add local field names (using BASED ON clauses) and local DATATRIEVE support options (QUERY_NAME and QUERY_HEADER clauses) in the DEFINE RELATION statement. Of course, some of these same features can be part of a global field. You can either include them when you create the field with a DEFINE FIELD statement, or you can add them later on using the CHANGE FIELD statement.

Table 2-2 lists some of the different was to use the DEFINE RELATION statement.

Table 2-2: Options in the DEFINE RELATION Statement

Contents of DEFINE RELATION Statement	Result
Existing field name	Copies existing field name and definition.
Existing field name, DATATRIEVE clauses	Copies existing field name and definition, adds DATATRIEVE query header and/or query name.
New field name, new field definitions, with or without DATATRIEVE clauses	Creates new definition.
New field name, BASED ON clause, with or without DATATRIEVE clauses	Copies existing field definition, gives field new name.
New field name, COMPUTED BY clause, with or without DATATRIEVE clauses	Creates new field definition based on a value expression.

The following example shows the procedure that defines the five relations for the Overnite Hotel. Once these definitions are stable, you can include them in a command file similar to the one shown in Appendix A.

```
!  Define HOTEL relation

DEFINE RELATION HOTEL.
   ROOM_NUMBER
         QUERY_HEADER FOR DATATRIEVE IS "ROOM"/"NUMBER"
         QUERY_NAME FOR DATATRIEVE IS "RNUM".
   ROOM_TYPE
         QUERY_HEADER FOR DATATRIEVE IS "ROOM"/"TYPE"
         QUERY_NAME FOR DATATRIEVE IS "RTYPE".
END HOTEL RELATION.

!  Define TYPE relation

DEFINE RELATION TYPE.
   ROOM_TYPE
         QUERY_HEADER FOR DATATRIEVE IS "ROOM"/"TYPE"
         QUERY_NAME FOR DATATRIEVE IS "RTYPE".
   RATE_CODE
         QUERY_HEADER FOR DATATRIEVE IS "RATE"/"CODE"
         QUERY_NAME FOR DATATRIEVE IS "RATCOD".
   BEDS
         QUERY_HEADER FOR DATATRIEVE IS "NUMBER"/"OF BEDS"
         QUERY_NAME FOR DATATRIEVE IS "NUM_BED".
   TELEPHONE
         QUERY_HEADER FOR DATATRIEVE IS "TELEPHONE"
         QUERY_NAME FOR DATATRIEVE IS "PHONE".
   TV
         QUERY_HEADER FOR DATATRIEVE IS "TELEVISION"
         QUERY_NAME FOR DATATRIEVE IS "TV".
   AC
         QUERY_HEADER FOR DATATRIEVE IS "AIR"/"CONDITIONING"
         QUERY_NAME FOR DATATRIEVE IS "AIR".
END TYPE RELATION.

!  Define RATES relation

DEFINE RELATION RATES.
   RATE_CODE
         QUERY_HEADER FOR DATATRIEVE IS "RATE"/"CODE"
         QUERY_NAME FOR DATATRIEVE IS "R_CODE".
   STD_RATE BASED ON STANDARD_RATE
         QUERY_HEADER FOR DATATRIEVE IS "STANDARD"/"RATE"
         QUERY_NAME FOR DATATRIEVE IS "ST_RATE".
   GOV_RATE
          COMPUTED BY (STD_RATE * 0.90)
         QUERY_HEADER FOR DATATRIEVE IS "GOVERNMENT"/"RATE"
         QUERY_NAME FOR DATATRIEVE IS "G_RATE".
   GROUP_RATE
          COMPUTED BY (STD_RATE * 0.86)
         QUERY_HEADER FOR DATATRIEVE IS "GROUP"/"RATE"
         QUERY_NAME FOR DATATRIEVE IS "GRP_RATE".
END RATES RELATION.
```

(continued on next page)

```
!  Define RESERVATION relation

DEFINE RELATION RESERVATION.
   GUEST_NAME
         QUERY_HEADER FOR DATATRIEVE IS "GUEST"/"NAME"
         QUERY_NAME FOR DATATRIEVE IS "NAME".
   CITY
         QUERY_HEADER FOR DATATRIEVE IS "CITY".
   STATE
         QUERY_HEADER FOR DATATRIEVE IS "STATE".
   POSTAL_CODE
         QUERY_HEADER FOR DATATRIEVE IS "POSTAL"/"CODE".
   RESERVE_DATE BASED ON STANDARD_DATE
           QUERY_HEADER FOR DATATRIEVE IS "RESERVATION"/"DATE"
           QUERY_NAME FOR DATATRIEVE IS "RESRV_DATE".
   ARRIVE_DATE BASED ON STANDARD_DATE
           QUERY_HEADER FOR DATATRIEVE IS "ARRIVAL"/"DATE"
           QUERY_NAME FOR DATATRIEVE IS "A_DATE".
   DEPART_DATE BASED ON STANDARD_DATE
           QUERY_HEADER FOR DATATRIEVE IS "DEPARTURE"/"DATE"
           QUERY_NAME FOR DATATRIEVE IS "D_DATE".
   LENGTH_OF_STAY
         QUERY_HEADER FOR DATATRIEVE IS "LENGTH"/"OF STAY"
         QUERY_NAME FOR DATATRIEVE IS "STAY".
   PARTY_SIZE
         QUERY_HEADER FOR DATATRIEVE IS "PARTY"/"SIZE"
         QUERY_NAME FOR DATATRIEVE IS "P_SIZE".
   CONFIRMED BASED ON STANDARD_FLAG
         QUERY_HEADER FOR DATATRIEVE IS "RESERVATION"/"CONFIRMED"
         QUERY_NAME FOR DATATRIEVE IS "RESRV_CONF".
   CHECK_OUT BASED ON STANDARD_FLAG
         QUERY_HEADER FOR DATATRIEVE IS "CHECKED"/"OUT"
         QUERY_NAME FOR DATATRIEVE IS "CHK_OUT".
   ROOM_NUMBER
         QUERY_HEADER FOR DATATRIEVE IS "ROOM"/"NUMBER"
         QUERY_NAME FOR DATATRIEVE IS "RNUMB".
   ROOM_RATE BASED ON STANDARD_RATE
         QUERY_HEADER FOR DATATRIEVE IS "ROOM"/"RATE"
         QUERY_NAME FOR DATATRIEVE IS "R_RAT".
END RESERVATION RELATION.
```

```
! Define BILLING relation

DEFINE RELATION BILLING.
   ROOM_NUMBER
         QUERY_HEADER FOR DATATRIEVE IS "ROOM"/"NUMBER"
         QUERY_NAME FOR DATATRIEVE IS "RNUMB".
   SERVICE_CHARGE BASED ON STANDARD_RATE
         QUERY_HEADER FOR DATATRIEVE IS "SERVICE"/"CHARGE"
         QUERY_NAME FOR DATATRIEVE IS "S_CHG".
   TX_DATE BASED ON STANDARD_DATE
          QUERY_HEADER FOR DATATRIEVE IS "TRANSACTION"/"DATE".
   SERVICE_DESCRIP
         QUERY_HEADER FOR DATATRIEVE IS "SERVICE"/"DESCRIPTION"
         QUERY_NAME FOR DATATRIEVE IS "S_DESCR".
   SERVICE_CODE
         QUERY_HEADER FOR DATATRIEVE IS "SERVICE"/"CODE"
         QUERY_NAME FOR DATATRIEVE IS "S_CODE".
END BILLING RELATION.
```

The logical model used to create the physical database definition also included an entity called GUEST. When you examine the field and relation definitions, you see that the GUEST relation includes fields from relations already defined. Defining a GUEST relation could lead to inconsistencies and other udpate problems. A GUEST relation, therefore, is not the best solution. You can make the same data available by defining a GUEST *view.*

You can create a query containing an RSE that refers to all fields in other relations necesary to describe a GUEST. Using the Rdb/VMS view feature, you can make such a query a permanent part of the database.

Section 2.8 shows you how to define a view called GUEST. Since all the fields in the GUEST view are actually in other relations, the GUEST view can use these existing fields and values rather than storing its own.

2.6 Defining Constraints

Rdb/VMS provides you with a feature that helps the database to maintain referential integrity. That is, for every value of a foreign key in a relation, you want a matching value in the primary key field of another relation. When no such constraint checking is performed, it is possible to add a value to the foreign key field in one relation that does not refer to the primary key value in another relation. Therefore, even though your database design is normalized, you want to ensure that the links between the foreign key in a relation and a primary key in another are secure. You use the Rdb/VMS constraint feature to check another relation for the presence of specific values.

You can place constraints on fields in VAX Rdb/VMS in two ways:

- With the VALID IF clause in the DEFINE FIELD statement

  VALID IF is intended primarily to allow Rdb/VMS to check the range of a value when it is entered or stored.

- With the DEFINE CONSTRAINT statement

  DEFINE CONSTRAINT allows you more flexibility than VALID IF. A formal constraint checks the validity of one field in terms of others in the database.

This section shows how to use DEFINE CONSTRAINT. See Section 2.4.1.3 for information about the VALID IF clause.

The DEFINE CONSTRAINT statement consists of three parts:

- A name
- A FOR clause, which specifies a record selection expression

  The record selection expression determines which records will be checked to see if they meet the conditions of the constraint.

- A REQUIRE clause, which specifies a conditional expression

  The conditional expression sets up the conditions a record must meet to be entered in the database.

For example, when someone enters a RATE_CODE, the foreign key in the TYPES relation, you want to be sure that the rate code already is valid. If the rate code exists in the RATES relation, then it is valid. To check the value by looking it up in a relation, define the constraint like this:

```
DEFINE CONSTRAINT RATE_CODE_EXISTS
  FOR T IN TYPES
    REQUIRE (ANY R IN RATES
    WITH R.RATE_CODE = T.RATE_CODE).
```

In the following example, the constraint checks to see if the billing transaction matches an actual guest staying in the hotel:

```
DEFINE CONSTRAINT SERVICE_CHECK
  FOR B IN BILLING
    REQUIRE (ANY R IN RESERVATION
    WITH R.ROOM_NUMBER = B.ROOM_NUMBER
    AND R.CONFIRMED = "Y").
```

It is important to remember that using constraints affects performance in certain ways. Rdb/VMS must place locks on one or more relations to check field values. This means that Rdb/VMS might have to perform several join operations for a complex constraint evaluation. To ensure maximum performance for constraint evaluation, you should define indexes for primary and foreign keys. In general, avoid very complex constraint definitions that refer to many relations.

2.7 Defining Indexes

Indexes are special tables added to the database internally to speed searching relations for selected records. When you use the DEFINE INDEX statement to add an index key to a relation, Rdb/VMS builds an index using the field you specify. When you perform an operation that requires searching or joining by the indexed field, Rdb/VMS uses the index to find records directly, without a sequential scan of the records in the relation.

Index keys are especially important in a relational database, because you are joining records from different relations frequently. Index keys make it easier for join operations to retrieve data quickly and directly.

On the other hand, index nodes might have to be updated to reflect changes in the data. When values change in the database, Rdb/VMS must update the corresponding index nodes automatically to reflect these changes. Some update processes can create a large number of changes to indexed fields. Updating indexed fields can take up valuable time and resources, such as locks. The nature of your database activity can determine when to use indexes to your advantage and when to avoid them.

The following guidelines can help you decide where to use indexed fields. Define an index for a field when you:

- Identify primary and foreign keys
- Retrieve data often from the relation

  If you specify READ_ONLY in your START_TRANSACTION statement, using indexed fields to locate records results in fast and efficient retrieval.
- Use complex queries that contain a CROSS clause to combine several relations

  When the join operation uses common (primary and foreign key) fields that are also indexed, retrieval time improves.
- Use statistical functions

  You should use indexed fields when finding values from MAX, MIN, AVERAGE, and TOTAL.

Avoid indexes when you:

- Store large numbers of records in a single transaction
- Delete many records from the database

Picking a successful strategy for defining indexes is a complex task. You should always define indexes for primary and foreign keys. Such a policy ensures that primary key fields cannot contain duplicate values but that foreign key fields can hold duplicate values.

Primary and foreign keys tend to be relatively stable. You are less likely to modify key values than other fields in the record. You should be concerned with the overhead required for tasks that update the index nodes only when you store or erase large numbers of records. Indexed primary and foreign key fields normally provide an efficient and dependable search path to the desired records.

Often you can tell which fields, other than primary and foreign key fields, should be indexed only after you monitor the usage of the database for some time. If you notice that some fields are used for joins and retrievals, you can define indexes for them. On the other hand, if you see that a relation is frequently updated, you might decide to delete indexes for those fields in that relation. Note that using indexed fields to locate records containing other, nonindexed, fields does not impair update performance.

When you are updating a large relation, either by storing many new records or erasing them, performance might be improved if you delete indexes for fields in that relation, run the update procedure, and then redefine the index when the task is finished. In this way, Rdb/VMS rebuilds the index structure only once, rather than once for each update operation.

As a rule of thumb, however, you should define all the indexes you believe will improve the performance of retrievals. Here are some additional guidelines for determining which fields to index:

- Choose those fields in the relation you use frequently to locate records. Primary key fields are almost always used to select one or more records.
- Include fields common to two or more relations, because these are the fields used in CROSS operations. A foreign key field in one relation forms the link to the primary key field in another relation. It is good practice to define indexes for all primary and foreign key fields in the database.
- Decide whether or not that field can store duplicate values. Primary key fields must have unique values; they cannot allow duplicate values. On the other hand, foreign keys often contain identical values.

Sometimes a primary key field actually consists of two fields. Each of these fields by itself could hold duplicate values, but when used in combination, they form a primary key field that contains only unique values. Defining an index for such a primary key field is called a multisegment index. Rdb/VMS lets you name the index, include the names of the fields used in the key, and specify whether the index is allowed to store duplicate values.

The RESERVATION relation is one that lends itself to multisegment indexes. It contains the following fields:

```
GUEST_NAME
CITY
STATE
POSTAL_CODE
RESERVE_DATE
ARRIVE_DATE
DEPART_DATE
LENGTH_OF_STAY
PARTY_SIZE
CONFIRMED
CHECK_OUT
ROOM_NUMBER
ROOM_RATE
```

No single field value for any of these fields can guarantee retrieval of a unique record. For example, the GUEST_NAME field in one record can contain the same value as other occurrences of that field in other records. The various date fields are not unique. Even the address information could be duplicated.

By selecting a combination of two fields, however, you can create an index value that is unique. The combination of the two fields, GUEST_NAME and POSTAL_CODE, might result in a unique value. Therefore, you might select the following combinations of fields to create an index that can satisfy the NO DUPLICATES ALLOWED clause of the DEFINE INDEX statement.

```
GUEST_NAME and POSTAL_CODE
GUEST_NAME and ARRIVE_DATE
GUEST_NAME and PARTY_SIZE
GUEST_NAME and ROOM_NUMBER
```

Refer to Chapter 4 of the *VAX Rdb/VMS Guide to Database Administration and Maintenance* for a complete description of indexes and how you can use them.

The following examples define indexes for some fields in the OVERNITE relations:

```
DEFINE INDEX HOTEL_ROOM_NUMBER
      DESCRIPTION IS /* Primary key for the HOTEL relation */
      FOR HOTEL
      DUPLICATES ARE NOT ALLOWED.
         ROOM_NUMBER.
END HOTEL_ROOM_NUMBER INDEX.
```

(continued on next page)

```
DEFINE INDEX HOTEL_ROOM_TYPE
      DESCRIPTION IS /* foreign key for the HOTEL relation */
      FOR HOTEL
      DUPLICATES ARE ALLOWED.
         ROOM_TYPE.
END HOTEL_ROOM_TYPE INDEX.

DEFINE INDEX CODE_RATE
      DESCRIPTION IS /* Primary key for the RATES relation */
      FOR RATES
      DUPLICATES ARE NOT ALLOWED.
         RATE_CODE.
END CODE_RATE INDEX.
```

2.8 Defining Views

The definition of the OVERNITE database separated the hotel's data into logically related groups. Because the normalization process often results in defining additional relations, the task of gathering data from these relations can be cumbersome. Accessing data that occurs in several different relations might mean entering the same complex queries repeatedly. However, Rdb/VMS provides an efficient method to make these queries "permanent." You can create views to combine different portions of many relations in the database.

You can think of a view as a "virtual relation." To the user who is not familiar with the database definitions, a view looks just the same as a relation: it has a name, a set of fields, and a number of records. Because a view, like a query, is created from a record selection expression, it simply refers to the fields contained in the existing relations by naming them in an RSE. It stores no data of its own. Views have the following advantages:

- Security

  You can prevent unauthorized users from accessing sensitive data by specifying only those records and fields you want certain users to see.

- Easy access

  Queries using complex selection criteria can be formalized in a view definition to make access to selected portions of the database easy.

- Easy update

  You can update views that are based on a *single* relation.

- Organization

  You can assemble different groups of fields from existing relations for host language program access or for DATATRIEVE users. Defining views for programs can significantly reduce complex RSEs embedded in the program source code.

As the examples in the following sections show, joining relations can be complex. If you frequently form the same RSE to retrieve records from several relations, you might consider creating a view definition. A view can bring together fields from one or more 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 use RDO statements to display or manipulate field values. Thus, a user who might not readily understand the syntax for a complex join can still access data from such a join when it is defined in a view.

If you use such a complex query frequently, you can create a view definition to refer to that restricted record stream from several relations. The DEFINE VIEW statement uses an RSE to specify the record stream you want to establish. You also must indicate which fields from the stream you want included in the view. You can use the resulting view definition instead of the query itself.

When Rdb/VMS is installed on your system, a sample PERSONNEL database is available to try the examples shown in the Rdb/VMS documentation. The PERSONNEL database contains three view definitions. Two of these views, CURRENT_JOB and CURRENT_SALARY are similar to the GUEST view; each refers to two relations in a database. The third PERSONNEL view, CURRENT_INFO, is more complex than the others because it refers to both relations and views in its definition. The examples in the following sections illustrate how to create the GUEST view first and then the three PERSONNEL database views.

Before defining a view, you can join two, three, or more relations and issue a query to be sure that you are accessing the correct data. Once you have determined that the data is correct, you can use the same fields from the join to create a view. The PERSONNEL database examples first create joins and then define the views.

2.8.1 Creating the GUEST View

Chapter 1 showed you how to create five relations for the OVERNITE database. Because a GUEST relation would include only fields that other relations already contained, defining a relation called GUEST would invite inconsistencies and unnecessary duplication of data. The hotel finds it convenient to assemble this special group of fields for routine queries. Defining a GUEST view, therefore, solves the problems of inconsistency and redundancy while providing the hotel reservation system with a relation-like entity called a view.

The first three fields (GUEST_NAME, ROOM_NUMBER, and ADDRESS) appear in the RESERVATION relation. The last three fields (TOTAL_CHARGE, TOTAL_ROOM_CHARGE, and TOTAL_SERVICE_CHARGE) can be computed from information contained in the BILLING relation.

A GUEST view can better serve the hotel than a GUEST relation. For example, it might be a good idea for the cashier to have access only to total charge information, rather than be able to see itemized charges in the BILLING relation. In addition to specifying certain fields in a view definition to restrict access to sensitive or confidential data, you can also specify view access rights that precisely identify which tasks authorized users can perform with that view.

The GUEST view definition uses the CROSS clause to join the RESERVATION and BILLING relations using the ROOM_NUMBER field from each relation. The view includes the three global field definitions from the RESERVATION relation. The remaining three fields in the view are created using COMPUTED BY clauses that access information from the BILLING relation.

The GUEST view definition looks like this:

```
DEFINE VIEW GUEST OF R IN RESERVATION
 CROSS B IN BILLING OVER ROOM_NUMBER.
        R.GUEST_NAME.
        R.ROOM_NUMBER.
        R.ADDRESS.
        TOTAL_ROOM
          COMPUTED BY
                (R.LENGTH_OF_STAY * R.ROOM_RATE).
        TOTAL_SERVICE
          COMPUTED BY
                 TOTAL X.SERVICE_CHARGE OF X IN BILLING
                  WITH X.ROOM_NUMBER = R.ROOM_NUMBER.
        TOTAL_BILL
          COMPUTED BY
                ((R.LENGTH_OF_STAY * R.ROOM_RATE) +
                (TOTAL X.SERVICE_CHARGE OF X IN BILLING
                  WITH X.ROOM_NUMBER = R.ROOM_NUMBER)).

END VIEW.
```

You refer to this view by its defined name, GUEST. The definition gives new, convenient names to the total charges fields. When users refer to this view, they use context variables and the new field names as though the view were a relation in the database. The database still maintains the same information.

The following example shows how you can retrieve information from the GUEST view:

```
FOR G IN GUEST WITH G.ROOM_NUMBER = "204"
    PRINT
     G.ROOM_NUMBER,
     G.GUEST_NAME,
     G.ADDRESS,
     G.TOTAL_ROOM,
     G.TOTAL_SERVICE,
     G.TOTAL_BILL
END_FOR
```

Views also provide performance enhancements. Performing a join that involves many relations could be time-consuming. You can improve performance by defining a view that includes the join operation. Note that although you can define a view based on one or more existing views, in most cases it is more efficient to base all view definitions on the database relations themselves.

2.8.2 Creating the CURRENT_JOB View

The definitions for the relations in the PERSONNEL database do not provide a simple procedure to retrieve only information about an employee's current job. The necessary data for such a query is distributed between two relations: EMPLOYEES and JOB_HISTORY. To access the data you require, you need to include the following fields from the two relations:

- EMPLOYEES relation (basic data on an employee and supervisor)
  - EMPLOYEE_ID
  - FIRST_NAME
  - LAST_NAME
- JOB_HISTORY relation (all jobs held by an employee)
  - JOB_START
  - JOB_CODE
  - SUPERVISOR_ID
  - DEPARTMENT_CODE

Now you can form a query that joins these two relations. Because both the relations contain the EMPLOYEE_ID field, you can use this field as the join term in your record selection expression (RSE):

```
FOR JH IN JOB_HISTORY
  CROSS E IN EMPLOYEES OVER EMPLOYEE_ID
PRINT
      E.LAST_NAME,
      E.FIRST_NAME,
      E.EMPLOYEE_ID,
      JH.JOB_CODE,
      JH.DEPARTMENT_CODE,
      JH.SUPERVISOR_ID,
      JH.JOB_START
END_FOR
```

Remember that the JOB_HISTORY relation can contain many job history records for an employee. This query then retrieves all job history records for every employee. You can restrict the record stream further by requiring only the current job history record for each employee. No data value is stored in the JOB_END field for a current job history record; that is, the value is missing. Therefore you can find a current job history record by selecting records in the JOB_HISTORY relation where the JOB_END field is missing. The following query adds this clause to the RSE to include only current job history records with records from the EMPLOYEES relation.

```
FOR JH IN HOB_HISTORY
  CROSS E IN EMPLOYEES OVER EMPLOYEE_ID
    WITH JH.JOB_END MISSING
PRINT
      E.LAST_NAME,
      E.FIRST_NAME,
      E.EMPLOYEE_ID,
      JH.JOB_CODE,
      JH.DEPARTMENT_CODE,
      JH.SUPERVISOR_ID,
      JH.JOB_START
END_FOR
```

This query brings together just the fields you need from both relations and restricts the record stream to only current job history information. You can now turn this query into a view definition and add it to other database entity definitions in the database.

```
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,
END VIEW.
```

The following example shows how you can use the CURRENT_JOB view to find the current job history record for an individual employee:

```
FOR CJ IN CURRENT_JOB WITH CH.EMPLOYEE_ID = "00164"
  PRINT
    CJ.*
END_FOR
```

2.8.3 Creating the CURRENT SALARY View

You can follow the same steps to create the CURRENT_SALARY view as are shown in the previous example. CURRENT_SALARY joins the EMPLOYEES relation with the SALARY_HISTORY relation. First determine which fields you need from each relation:

- LAST_NAME from the EMPLOYEES relation
- FIRST_NAME from the EMPLOYEES relation
- EMPLOYEE_ID from the EMPLOYEES relation
- SALARY_START from the SALARY_HISTORY relation
- SALARY_AMOUNT from the SALARY_HISTORY relation

You use the following query to ensure that you are retrieving the current data:

```
FOR SH IN SALARY_HISTORY
  CROSS E IN EMPLOYEES OVER EMPLOYEE_ID
    WITH SH.SALARY_END MISSING
PRINT
      E.LAST_NAME,
      E.FIRST_NAME,
      E.EMPLOYEE_ID,
      SH.SALARY_START,
      SH.SALARY_AMOUNT
END_FOR
```

Now that you see the join works successfully, you can create the CURRENT_SALARY view:

```
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.
```

2.8.4 Creating the CURRENT_INFO View

The third PERSONNEL database view uses the first two views and two other relations in the database. Although this approach is not recommended when performance is a critical factor in your routine database tasks, it provides convenience to database users who need to assemble data values from fields distributed among several relations in the database. Again, you start by selecting the list of fields you need from each of these database entities:

- LAST_NAME from CURRENT_JOB view
- FIRST_NAME from CURRENT_JOB view
- EMPLOYEE_ID from CURRENT_JOB view
- DEPARTMENT_NAME from DEPARTMENTS relation
- JOB_TITLE from JOBS relation
- JOB_START from CURRENT_JOB view
- SALARY_START from CURRENT_SALARY view
- SALARY_AMOUNT from CURRENT_SALARY view

Views offer another feature that allows you to create customized field names from the fields in the referenced relations and views. You name a new, local field name using the FROM clause in the view definition and specify the name of the field in the relation or view on which it is based.

The view definition for CURRENT-INFO includes an RSE to join the two views and the two relations and specifies the new field names to refer to the original field names:

```
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.
     FIRSTNAME  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.
```

The new fields you name in the view definition have the same field attributes as the fields in the original relations. Furthermore, you can add DATATRIEVE QUERY_HEADER and QUERY_NAME characteristics for fields in the view that do not already have such characteristics in the base relations. In addition, you might want to include a COMPUTED BY field, WEEKLY, for the weekly salary rate. The following modification of the CURRENT_INFO view definition shows the new field characteristics for the field in the CURRENT_INFO view called WEEKLY:

```
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.
     FIRSTNAME  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.
     WEEKLY
       QUERY_HEADER FOR DATATRIEVE IS "WEEKLY"/"SALARY"
       QUERY_NAME FOR DATATRIEVE IS "WEEK"
         COMPUTED BY (CS.SALARY_AMOUNT/52).
END VIEW.
```

2.9 Loading the Database

There are three ways to enter data into your database:

- Use the STORE statement to add individual records.
- Use DATATRIEVE to load an existing RMS file.
- Write a high-level language program to load an existing file.

See the *VAX Rdb/VMS Guide to Database Administration and Maintenance* for information on loading the database.

2.10 Verifying the Definition Phase

Once you have defined the database, fields, and relations, you can verify that each step has been successful by using the RDO SHOW statement. If you do not know what a certain statement is supposed to do or cannot remember the proper syntax of a statement, you can use the HELP statement. Both statements provide online assistance that allows you to continue your interactive sessions without interruption. For further details about the SHOW and HELP statements, see the *VAX Rdb/VMS Reference Manual.*

Defining Database Protection 3

VAX Rdb/VMS provides a security mechanism to protect your database against browsing or modification by unauthorized users. The Rdb/VMS security mechanism applies specifically to Rdb/VMS operations and is independent of the security defined by the VAX Common Data Dictionary (CDD) and the VMS operating system. You should always use Rdb/VMS protection statements to manage the security of your database. (Note that although Rdb/VMS security is separate from VMS security, the Rdb/VMS security mechanism is based on the VMS security mechanism.)

Rdb/VMS security depends on **access control lists** (ACLs) attached to databases and relations. These lists define which users can access database entities and what operations they can perform. You can create these lists interactively by issuing DEFINE PROTECTION statements RDO. You can also build a command file of DEFINE PROTECTION statements and them process these after invoking the database.

When you first create an access control list, it is generally easier to build a command file so that you can edit your entries and put them in the optimum order. The first part of this chapter describes access control lists and then shows how to create entries and organize them.

You must invoke the database in order to process the command file or to interactively issue DEFINE PROTECTION statements. Section 3.7 discusses invoking the database.

You must also invoke the database to modify or delete ACL entries, as well as to verify the access control lists. Section 3.8 covers the CHANGE PROTECTION and DELETE PROTECTION statements for modifying or deleting entires. The SHOW PROTECTION statement, which enables you to verify ACLs, is covered in Section 3.9.

The access control lists maintained by the CDD apply only to the copies of the Rdb/VMS definitions stored in the CDD. You can use the CDD protection mechanism to protect the copies of the data definitions in the CDD from unauthorized access. You should *not* use the CDD Data Management Utility (DMU) to change protection for Rdb/VMS database entities.

3.1 The Access Control List

Each access control entry (ACE) consists of an identifier and the Rdb/VMS access rights assigned to the identifier. You must have control over the database or relation in order to create access control entries for that entity. When you create a database, Rdb/VMS automatically creates an ACL granting you CONTROL rights to that database. When you create a relation, Rdb/VMS automatically grants you the CONTROL privilege for that relation. Relation access control is not a privilege that depends on the database ACL.

When a user tries to perform an Rdb/VMS operation on a database or relation, Rdb/VMS reads the associated access control list from top to bottom, comparing the user's identifier with each entry. As soon as Rdb/VMS finds the first match, it grants the rights listed in that entry. For this reason, both the ACEs themselves and their order in the list are important.

To see the access control list for a database or relation, use the SHOW PROTECTION statement. SHOW PROTECTION displays the access control list in its correct order so you can see where to place new entries.

To define protection for a database or relation, you perform the following steps:

1. Decide what access rights you want to grant certain users and create a set of access control entries (ACEs).
2. Arrange these entries in the proper order.
3. Build the access control list using a series of DEFINE PROTECTION statements.

3.2 Creating Access Control List Entries

Each entry in an access control list contains:

- An identifier that specifies a user or set of users.
- A set of access rights to specify what operations that user or user group can perform on the database or database entity. Tables 3-1, 3-2, and 3-3 list the access rights.

You create ACEs with the DEFINE PROTECTION statement, using the IDENTIFIER and ACCESS clauses.

3.2.1 User Identifiers

The user identifier consists of the standard VMS identifier. There are three types of identifiers:

- UIC identifiers

  UIC identifiers depend on the user identification codes (UICs) that uniquely identify each user on the system. The UIC can be in either numeric format or alphanumeric format. The following are all valid UIC identifiers:

  ```
  [SYSTEM3, K_JONES]
  K_JONES
  [341,311]
  ```

- General identifiers

  General identifiers are defined by the VAX/VMS system manager in the system rights database to identify groups of users on the system. The following are possible general identifiers:

  ```
  DATAENTRY
  SECRETARIES
  MANAGERS
  ```

- System-defined identifiers

  System-defined identifiers are automatically defined by the system when the rights database is created at system installation time. System-defined identifiers are assigned depending on the type of login you execute. The following are all valid system-defined identifiers:

  ```
  BATCH
  NETWORK
  INTERACTIVE
  LOCAL
  DIALUP
  REMOTE
  ```

You can specify more than one identifier. However, you should regard the six system-defined identifiers as mutually exclusive. You can combine them with other identifiers (UICs and general identifiers), but when you specify two or more identifiers, separate them with plus signs (+). The following is a multiple identifier that specifies all users who are associated with the general identifier DATAENTRY and use RDO interactively:

```
DATAENTRY+INTERACTIVE
```

For more information about these types of identifiers, see the *Guide to VAX/VMS System Security* or the *VAX/VMS DCL Dictionary.*

3.2.2 Rdb/VMS Access Rights

Tables 3-1, 3-2, and 3-3 show you the access rights you can grant or deny Rdb/VMS users. Each access right corresponds to a set of Rdb/VMS statements. For example, if you did not specify DEFINE in a user's ACE for the database, Rdb/VMS returns an error message when the user tries to execute the DEFINE FIELD statement.

Users must have privileges both to the database and to any relations or views they need to perform data manipulation tasks. When you use a view to access the database, Rdb/VMS determines your access rights from that view's ACL, not from the ACLs of the underlying relations or views.

Table 3-1: Rdb/VMS Data Manipulation Access Rights

Access Right	To Grant	To Deny
Read data	READ	NOREAD
Store data	WRITE	NOWRITE
Modify data	MODIFY	NOMODIFY
Erase data	ERASE	NOERASE

Table 3-2: Data Definitions Statements Controlled by Database ACL

Access Right	To Grant	To Deny
Define global field or relation	DEFINE	NODEFINE
Change global field or database	CHANGE	NOCHANGE
Delete global field	DELETE	NODELETE
Define, change, delete protection for database	CONTROL	NOCONTROL
[Reserved for future versions]	SHOW	NOSHOW

Table 3-3: Data Definitions Statements Controlled by ACL for Each Relation or View in Statement

Access Right	To Grant	To Deny
Define view, index, or constraint	DEFINE	NODEFINE
Change relation	CHANGE	NOCHANGE
Delete relation, index, view, or constraint	DELETE	NODELETE
Define, change, delete protection for relation	CONTROL	NOCONTROL
[Reserved for future versions]	SHOW	NOSHOW

Table 3-4: Rdb/VMS Utility Statement Access Rights

Access Right	To Grant	To Deny
CHANGE PROTECTION DEFINE PROTECTION DELETE PROTECTION	CONTROL	NOCONTROL
[Reserved for future versions]	OPERATOR	NOOPERATOR
[Reserved for future versions]	ADMINISTRATOR	NOADMINISTRATOR

Note that database users must have OPERATOR privilege in order to use ANALYZE statements. Include OPERATOR access for any users who are responsible for analyzing the database.

3.2.3 Using the DEFINE PROTECTION Statement

You use a DEFINE PROTECTION statement to create each ACE for a database or relation. The statement specifies the following parameters for an entry:

- Whether the ACL you are building is for a database or relation. If the entry is for a relation, you must specify the name of the relation.
- The position of the entry within the ACL. You can use the POSITION clause to place the entry at a given sequence number or you can use the AFTER clause to place the entry after an entry associated with another identifier.
- The identifier of the user or user group to which the entry applies.
- The list of access rights to be granted or denied to the user or user group. If you want to grant all access rights to a user, you can specify the keyword ALL in the ACCESS clause.

After you have entered the DEFINE PROTECTION statements, you can use the SHOW PROTECTION statement to review the access control entries.

3.2.3.1 Specifying the Target of the DEFINE PROTECTION Statement -- You must specify whether the DEFINE PROTECTION statement applies to a database or a relation. If the target is the database, the statement operates on the most recently invoked database. If you are in doubt about which database is the target of the DEFINE PROTECTION statement, enter a FINISH statement for all other databases. If the target is a relation, you must include the relation name with the DEFINE PROTECTION statement.

```
DEFINE PROTECTION FOR DATABASE

DEFINE PROTECTION FOR RELATION HOTEL
```

The keyword FOR is optional.

3.2.3.2 Specifying the Location of the Entry -- Next you can specify the position of the entry. You can use either the AFTER or POSITION clause. If you do not include either clause, Rdb/VMS places the entry at the top of the list.

You include a user identifier with the AFTER clause to show which entry you want your entry to follow. For example:

```
DEFINE PROTECTION FOR DATABASE
     AFTER [42,350]
```

When you use the POSITION clause, you must specify the exact position you want your entry to have in the list:

```
DEFINE PROTECTION FOR DATABASE
     POSITION 5
```

If you specify a position when there are fewer than that number of entries in the list, Rdb/VMS places the entry last. For example, if you specify position 12 and there are only 10 entries in the list, the new entry is placed in position 11 and given that position number.

In general, when you are adding ACEs to an existing list, you know what position you want the entry to have. If you are creating a new ACL, you might need to organize your list before you can determine the position for each entry. Section 3.4 discusses ordering ACEs.

3.2.3.3 The IDENTIFIER Clause -- The IDENTIFIER clause contains the user identifier for the entry you are creating. You can use the UIC number or an identifier name.

```
DEFINE PROTECTION FOR DATABASE
     POSITION 6
     IDENTIFIER [42,360]

DEFINE PROTECTION FOR DATABASE
     AFTER [42,350]
     IDENTIFIER [ADMIN,FORD]
```

General and system-defined identifiers are also allowed. If you specify two or more identfiers for an entry, separate them with plus signs (+).

```
DEFINE PROTECTION FOR DATABASE
     POSITION 10
     IDENTIFIER SECRETARIES + DIALUP
```

You can use the asterisk (*) wildcard character as part of a UIC identifier. For example, if you want to specify all users in a group, you can enter [42,*] as the identifier. When Rdb/VMS creates a database, it automatically creates an ACE with the identifier [*,*], which grants all privileges, except CONTROL, to any user.

3.2.3.4 The ACCESS Clause -- You include the various privileges you want to grant in the ACCESS clause. Most access rights have a NO version (for example, CONTROL and NOCONTROL) so that you can specifically deny a privilege. You can use the keyword ALL to grant all privileges to a user. A combination of ALL and one or more NO accesses is often easier to enter than listing a large number of access rights.

You use plus signs (+) to separate the access rights. If you have too many access rights to fit on one line, you can use the hyphen (-) to continue the list on the next line. The list of access rights must be enclosed in quotation marks (").

The following examples show different elements of the ACCESS clause:

```
DEFINE PROTECTION FOR DATABASE
     POSITION 7
     IDENTIFIER [BOARD,ROBERTS]
     ACCESS "ALL+NOCONTROL"

DEFINE PROTECTION FOR DATABASE
     POSITION 8
     IDENTIFIER [ADMIN, FORD]
     ACCESS "READ+WRITE+MODIFY+ERASE+DEFINE+CHANGE -
          +DELETE"
```

3.3 Building Access Control Lists

When you define a database, Rdb/VMS automatically creates a default ACL for the database at database creation time. Rdb/VMS creates a default ACL for each relation when you enter a DEFINE RELATION statement. All of these ACLs have two entries:

- The owner's, which grants all access rights. These rights include the CONTROL privilege, which lets you change ACLs. If you have the CONTROL access right, there is no way for you to deny yourself that privilege.
- An entry with the identifier [*,*], which grants all users all rights except CONTROL. If you, as owner, want to make use of the Rdb/VMS security mechanism, you should delete or change this entry as part of the process of defining protection.

If you issue SHOW PROTECTION statements just after the database has been created, the results look like this:

```
RDO> SHOW PROTECTION FOR DATABASE
 (IDENTIFIER=[GROUP2,JONES],ACCESS=READ+WRITE+MODIFY+ERASE+SHOW+
   DEFINE+CHANGE+DELETE+CONTROL+OPERATOR+ADMINISTRATOR)
 (IDENTIFIER=[*,*],ACCESS=READ+WRITE+MODIFY+ERASE+SHOW+
   DEFINE+CHANGE+DELETE+OPERATOR+ADMINISTRATOR)
RDO> !
RDO> !
RDO> SHOW PROTECTION FOR RELATION BILLING
 (IDENTIFIER=[GROUP2,JONES],ACCESS=READ+WRITE+MODIFY+ERASE+SHOW+
   DEFINE+CHANGE+DELETE+CONTROL+OPERATOR+ADMINISTRATOR)
 (IDENTIFIER=[*,*],ACCESS=READ+WRITE+MODIFY+ERASE+SHOW+
   DEFINE+CHANGE+DELETE+OPERATOR+ADMINISTRATOR)
RDO>
```

As part of defining the database, you probably want to restrict access more than the default protection does, both for the database and for some or all of the relations.

Rdb/VMS uses two ACLs for each data manipulation access to a relation: one for the database and one for the relation. For a particular user, Rdb/VMS allows a data manipulation access right to a relation only if that right is granted in *both* the database ACL and the relation ACL. That is, a user has WRITE privilege to the EMPLOYEES relation only if that user has WRITE privilege to both the PERSONNEL database and the EMPLOYEES relation. Thus, the database ACL should grant to each user or group of users all the data manipulation privileges they might need for any relation. Privileges can then be denied at the relation level.

If you wish to grant users the privilege to define indexes, views, or constraints for a relation, you must grant them DEFINE privilege for that particular relation. However, you do not need to grant users DEFINE privilege for the database itself.

See Tables 3-2 and 3-3 for additional information about which operations can be controlled by the ACLs of the database or relation.

To create an ACL, you can enter the individual DEFINE PROTECTION statements interactively at the RDO prompt. In general, however, it is easier to use a text editor to build a command file that defines protection for the whole database. The command file method is also useful when you are adding a number of ACEs to a database and its relations. You can use the interactive method when you want to add a few new entries to an existing ACL.

You start building the command file by creating the ACL for the database. Then you add the relation ACLs. The steps for creating a relation ACL are the same as for a database ACL.

1. Type the identifier and access privileges for each user or group you want to have access to the database or relation.
2. Arrange the entries in the order you want them in the ACL.
3. Edit the entries to create the DEFINE PROTECTION FOR DATABASE statements.

The following discussion shows the first step in creating a command file to add ACEs to the database. Assume that you are the owner and your UIC is [GROUP2.JONES]. There is no need to include your own ACL, because it is first on the list by default and it grants you all privileges. You can use comment fields to make your restrictions clear.

An analysis shows user classes and their associated privileges. The examples in the following list include the comments, IDENTIFIER clauses, and ACCESS clauses. The initial DEFINE PROTECTION portion of the statement as well as the POSITION and/or AFTER clauses are added later.

- You are the owner, user [GROUP2,JONES]. Protection for the owner is defined by default to have all privileges and is placed in position 1 of the ACL.
- User [ADMIN,SMITH] is the manager of your department. She wants clear access to all data at all times. However, you do not want to grant her data definition or database maintenance privileges.

```
! Manager -- needs to be able to use all data manipulation
! statements.
!

     IDENTIFIER [ADMIN,SMITH]
     ACCESS "READ+WRITE+MODIFY+ERASE"
```

- User [GROUP2,CLARK] is going to help you with restructuring databases. Therefore, she must have the right to use DEFINE, CHANGE, and DELETE in the ACLs for *any* relations she may be restructuring. To perform data definition statements, she must also have READ access to system relations. However, she should not be able to change data in the database. Deny her access to update statements and to CONTROL, OPERATOR, and ADMINISTRATOR statements:

```
!
! Assistant -- needs to be able to use data
! definition statements.
!

     IDENTIFIER [GROUP2,CLARK]
     ACCESS "READ+DEFINE+CHANGE+DELETE"
```

- User [GROUP2,LAWRENCE] is the nighttime operator. He performs maintenance functions, like backup and restore. The BACKUP statement requires READ access to the database and to every relation. Grant him READ access:

```
!
! Operator -- needs to be able to perform database
! maintenance tasks.
!

     IDENTIFIER [GROUP2,LAWRENCE]
     ACCESS "READ"
```

- Programmers are defined with the general identifier "PROGRAMMERS" in the system rights database. They must be able to modify database definitions and check the results. Grant them all the rights except those associated with database maintenance:

```
!
! Programmers -- need to perform data
! definition and data manipulation on some
! relations to test application programs.
!

        IDENTIFIER PROGRAMMERS
        ACCESS "READ+WRITE+MODIFY+ERASE+DEFINE+CHANGE+DELETE"
```

- Users in ADMIN are clerks who are only allowed to generate reports. They cannot run programs that modify information in the database. Grant them access only to the READ statement:

```
!
! Clerks -- need to be able only to read
! data. No access to modify, erase, store, data
! definition, or maintenance statements.
!

        IDENTIFIER [ADMIN,*]
        ACCESS "READ"
```

- User [ADMIN,FORD] is a secretary who runs programs that update the database. He needs to be able to read, write, and delete information in the database. Grant him access only to the data manipulation statements:

```
!
! Secretary -- needs to be able to read,
! write, and delete data. No access to data
! definition or maintenance.
!

        IDENTIFIER [ADMIN,FORD]
        ACCESS "READ+WRITE+MODIFY+ERASE"
```

- You want to deny database access to all other users. The final entry in the default list grants all users all rights except CONTROL access. Therefore, you need to delete the final entry, identified by [*,*].

You can review the contents of a command file by issuing the VMS TYPE command with the command file's specification:

```
$ TYPE DEFINEPRO.RDO
```

3.4 Putting the Access Control List in Order

The next step is to place the entries in order in the ACL for the database.

When a user tries to perform an Rdb/VMS operation on a database or relation, Rdb/VMS reads the access control list for the database entity from top to bottom, comparing the user's identifier with the identifier(s) listed in each entry. When Rdb/VMS finds the first match, it grants the rights listed in that entry and stops the search.

All UICs that do not match a previous entry "fall through" to the entry [*,*], if it exists. If there is no entry with the UIC [*,*], then unmatched UICs are denied all access to the database or relation.

Assume user [GROUP2,JONES] has the numeric UIC [250,210]. He would also match any of the following UICs from an access control list:

```
[250,210]
[250,*]
[*,JONES]
[GROUP2,*]
[*,210]
[*,*]
```

Here are two general guidelines for ordering access control entries:

- The less restrictive the user identifier, the lower on the list that ACL should go.
- The more powerful the privilege, the higher on the list that ACL should go.

Because Rdb/VMS reads the list from top to bottom, you should place entries with more specific identifiers earlier and those with more general ones later. For example, if you place the entry with the most general UIC identifier [*,*], first in the list, all users match it, and Rdb/VMS grants or denies all the access rights specified there to all users.

Similarly, if you place the general entry [ADMIN,*] before the specific entry [ADMIN,FORD], Rdb/VMS matches user [ADMIN,FORD] with [ADMIN,*] and denies the access rights WRITE, MODIFY, and ERASE, which user [ADMIN,FORD] needs.

Using the sample file from Section 3.3, you might put the entries in the following order:

```
!
! Owner -- already defined, in position 1 of the
! ACL, with all privileges
!

     [GROUP2,JONES]

!
! Assistant -- needs to be able to use data
! definition statements.
!

     [GROUP2,CLARK]

!
! Operator -- needs to perform database maintenance
! tasks.
!

     [GROUP2,LAWRENCE]

!
! Manager -- needs to be able to use all data
! manipulation statements.
!

     [ADMIN,SMITH]

!
! Secretary -- needs to be able to read,
! write, and delete data. No access to data
! definition or maintenance.
!

     [ADMIN,FORD]

!
! Programmers -- need to be able to perform data
! definition and data manipulation on some
! relations to test application programs.
!

     PROGRAMMERS

!
! Clerks  -- need to be able only to read
! data. No access to modify, erase, store, data
! definition, or maintenance statements.
!

     [ADMIN,*]

!
! Deny access to all users not explicitly granted
! access to the database.
!
```

3.5 Building an Access Control List for a Relation

The list you have compiled grants database access to all the users who need it. However, you might want to put additional restrictions on certain relations in the database.

For example, the BILLING relation contains sensitive information. Only the department manager should have the privileges to run the programs that read, write, and modify the BILLING relation. Therefore, you must deny all other users access to BILLING.

By default, you receive all privileges. Delete the [*,*] entry to restrict access to the relation. Then, specify the rights you want the manager to have.

```
!
! Manager -- needs to be able to use all data manipulation
!            statements.
!

    IDENTIFIER [ADMIN,SMITH]
    ACCESS "READ+WRITE+MODIFY+ERASE"
```

3.6 Defining Protection for Views

The discussion of views in Chapter 2 mentioned security as one of the advantages to creating these "virtual" relations. You can use a view to restrict access to specific fields of one or more relations or views. You can also apply precise database access rights to those fields in the view definition to maintain the required level of security for your database.

You can define a view based on:

- One or more relations
- One or more views
- A combination of views and relations

Rdb/VMS allows you to specify access rights for every relation. However, granting a user READ access to a relation makes every field in the record available for retrieval by that user. You cannot restrict access to specific fields in that record with relation level protection. Your intention, however, might be to allow that user to access only two fields in each of two relations.

The first step is to secure the fields in the base relations by denying at the relation level certain access rights for that group of users.

Next, you can define a view that includes only four fields, two from each relation. You can then define protection for the view that allows certain users read access to the four fields from the two base relations.

In this way, you can make a subset of a relation's fields, records, or a subset of both fields and records available to authorized users. Views, therefore, provide field level protection for your database.

Remember, however, that, if you grant any user-restricted access to the data in a relation, you should not include the DEFINE privilege at the same time. In that case, a user may define his or her own views to access a relation's data and defeat the original restrictions.

When you grant or deny access rights for a particular view, Rdb/VMS evaluates only the ACLs for that view, but does not evaluate the ACLs from the underlying relations or views. For example, the following view definition provides the front desk with the records only of the hotel's guests who have not yet left. From those records, the desk clerk can access only four fields. Those fields have new names in the view definition. The desk clerk needs to be able to update one or more of those fields to indicate that the guest has either checked out or extended his or her stay.

Now you can restrict access rights to the base relation and grant them for the subset of fields and records defined in the view.

```
DEFINE VIEW GUEST_EXIT
  OF R IN RESERVATION
  WITH R.DEPART_DATE GT "01-SEP-1985".
   GUEST       FROM GUEST_NAME.
   ROOM        FROM ROOM_NUMBER.
   EXIT_DAY    FROM DEPART_DATE.
   GONE        FROM CHECK_OUT.
END GUEST_EXIT VIEW.
```

The following example shows how you can restrict access by the front desk to the fields in the RESERVATION relation while specifying update access for the GUEST_EXIT view. Remember, you can update views defined on a single base relation.

```
DEFINE PROTECTION FOR RELATION RESERVATION
  IDENTIFIER DESK
  ACCESS "NOREAD+NOWRITE+NOMODIFY+NOCONTROL+NODEFINE+NODELETE+ -
         NOERASE+NOCHANGE"

DEFINE PROTECTION FOR VIEW GUEST_EXIT
  IDENTIFIER DESK
  ACCESS "READ+WRITE+MODIFY"
```

You can provide other views based on the same relation to allow other groups of users only the access rights they require. In this way, you can control the update of an entire relation by one or more groups responsible for the data in that relation while maintaining security for all of the data in the database.

Refer to Chapter 2 for details about defining views.

3.7 Invoking the Database

You must invoke the database in order to process an ACL command file or to issue any of the following RDO statements:

```
DEFINE PROTECTION
SHOW PROTECTION
CHANGE PROTECTION
DELETE PROTECTION
```

After you call up RDO, you use the INVOKE DATABASE statement at the RDO prompt and supply the filename of the database:

```
$ RDO
RDO> INVOKE DATABASE
cont> FILENAME 'DISK2:[ACCOUNTING]OVERNITE
RDO>
```

You can include the filename on the same line as the INVOKE DATABASE statement.

When you want Rdb/VMS to process a command file, first invoke the database and then use the execute procedure (@file-spec) at the next RDO prompt.

For example, suppose the ACL command file is called DEFINEPRO.RDO. You first issue the RDO command at the DCL prompt. Then you use the INVOKE DATABASE statement to invoke the database. Once the database is invoked, you can execute the command file:

```
$ RDO
RDO> INVOKE DATABASE 'DISK2:[ACCOUNTING]OVERNITE
RDO> @DEFINEPRO
```

3.8 Changing and Deleting Protection

The mechanism for changing the protection on a database or relation is nearly identical to defining protection. Of course, you can change protection by adding entries to an ACL. You can also change protection by modifying or deleting existing entries. Remember to invoke the database before issuing CHANGE PROTECTION and DELETE PROTECTION statements.

This section describes modifying and deleting existing ACEs from the list.

3.8.1 Changing an Access Control Entry (ACE)

The CHANGE PROTECTION statement has the following clauses:

- An identifier that points to the target entry in the ACL
- An access clause that specifies a new set of access rights for that entry

When you use the CHANGE PROTECTION statement, the user's ACE inherits all the rights from the ACE that you are replacing. Therefore, to modify an entry, you must specify only the rights you want changed.

Suppose you want to upgrade the rights of your clerks so they can add records to the database as well as read data. They need to use the STORE statement, for which WRITE privilege is required. Currently, their access to OVERNITE is limited by the following entry in the ACL:

```
IDENTIFIER=[ADMIN,*],ACCESS=READ
```

To add WRITE access, issue the following statement:

```
CHANGE PROTECTION FOR DATABASE
     [ADMIN,*]
     ACCESS "WRITE".
```

If you know the position in the ACL of the target entry, you can use that number instead of the identifier. The SHOW PROTECTION statement shows you the sequence of the ACL. The following example is equivalent to the previous one, because the group UIC [ADMIN,*] is in position 7 on the ACL:

```
CHANGE PROTECTION FOR DATABASE
     7
     ACCESS "WRITE".
```

If you change the protection for an ACE that was defined with multiple identifiers, specify the identifiers in the same order in which they appeared in the DEFINE PROTECTION statement. For example, assume the identifier clause for an entry looks like this:

```
IDENTIFIER [250,*]+MANAGER+INTERACTIVE
```

In the CHANGE PROTECTION statement, make sure the identifiers are in the same order:

```
CHANGE PROTECTION FOR DATABASE
     [250,*]+MANAGER+INTERACTIVE
     ACCESS "NOCHANGE+NODELETE+NODEFINE".
```

3.8.2 Deleting an Entry from an ACL

To delete a protection restriction, use the DELETE PROTECTION statement. This statement is similar to CHANGE PROTECTION. You specify a database entity and a sequence number or an identifier, and Rdb/VMS deletes the corresponding ACE.

The command file that defined the ACL for the OVERNITE database used this statement to delete the final entry, [*,*]:

```
DELETE PROTECTION FOR DATABASE [*,*].
```

If you specify a sequence number in the DELETE PROTECTION statement and Rdb/VMS does not find an entry for that position, no entry is deleted. Instead, Rdb/VMS returns an error message indicating that it did not find a matching access control list entry.

The effect of the deletion depends on the entry. For example, if you delete an entry that refers to a specific user, that user might fall through to a more general level of restriction when Rdb/VMS tries to match the user's identifier with other entries. Thus, if you deleted the entry for the secretary who runs update programs (UIC=[ADMIN,FORD]), he would match the following entry in the ACL ([ADMIN,*]) and would still have the right to run report programs. The following example shows this operation:

```
DELETE PROTECTION FOR DATABASE [ADMIN,FORD].
```

3.9 Verifying the ACLs for a Database

You can use the SHOW PROTECTION statements to verify the ACLs for databases and relations. You must issue a separate SHOW PROTECTION statement for each ACL. Before issuing any statements, be sure you have invoked the database. You must specify whether you want to verify the database ACL or a relation ACL. If you want to look at a relation ACL, you must include the name of the relation in the statement.

```
$ RDO
RDO> INVOKE DATABASE 'DISK2[ACCOUNTING]OVERNITE'
RDO> SHOW PROTECTION FOR DATABASE
 (IDENTIFIER=[GROUP2,JONES],ACCESS=READ+WRITE+MODIFY+ERASE+SHOW+
   DEFINE+CHANGE+DELETE+CONTROL+OPERATOR+ADMINISTRATOR)
 (IDENTIFIER=[GROUP2,CLARK],ACCESS=READ+DEFINE+CHANGE+DELETE)
 (IDENTIFIER=[GROUP2,LAWRENCE],ACCESS=READ)
 (IDENTIFIER=[ADMIN,SMITH],ACCESS=READ+WRITE+MODIFY+ERASE)
 (IDENTIFIER=[ADMIN,FORD],ACCESS=READ+WRITE+MODIFY+ERASE)
 (IDENTIFIER=PROGRAMMERS,ACCESS=READ+WRITE+MODIFY+ERASE+
   DEFINE+CHANGE+DELETE)
 (IDENTIFIER=[ADMIN,*],ACCESS=READ)
RDO>
RDO> SHOW PROTECTION FOR RELATION BILLING
 (IDENTIFIER=[GROUP2,JONES],ACCESS=READ+WRITE+MODIFY+ERASE+SHOW+
   DEFINE+CHANGE+DELETE+CONTROL+OPERATOR+ADMINISTRATOR)
 (IDENTIFIER=[ADMIN,SMITH],ACCESS=READ+WRITE+MODIFY+ERASE)
```

To see just your access rights, use the SHOW PRIVILEGES statement. The SHOW PRIVILEGES statement displays your ACE when Rdb/VMS matches your identifier with the identifier specified in the ACE. Remember that Rdb/VMS reads the list from top to bottom. Although your identifier might match many ACEs, Rdb/VMS grants you access rights when it finds the first match between your identifier and an identifier in the ACE.

```
RDO> SHOW PRIVILEGES FOR DATABASE
 (IDENTIFIER=[GROUP2,JONES],ACCESS=READ+WRITE+MODIFY+ERASE+SHOW+
   DEFINE+CHANGE+DELETE+CONTROL+OPERATOR+ADMINISTRATOR)
```

Restructuring a Database 4

VAX Rdb/VMS allows you to restructure your database dynamically. That is, as the needs of your organization change, or as you improve your understanding of those needs, you can easily change the design of your database. You can add, delete, and modify the database elements that make up that design. The more care and thought you put into the initial design of the database, the better, but if changes are required, you can often make them without disturbing users and without making major changes to application programs.

This chapter uses the OVERNITE database to demonstrate how to use the CHANGE and DELETE statements for relations, fields, and the database itself. A full description of these statements appears in the *VAX Rdb/VMS Reference Manual.*

4.1 Changing Relations

As the OVERNITE database grows and the requirements for the application become clearer, you want to add new fields to provide more information. The following examples demonstrate how to do this.

Example 1

The TYPES relation might need a description field that tells how large the room is. The current definition for the TYPES relation looks like this:

```
DEFINE RELATION TYPES.
   ROOM_TYPE.
   RATE_CODE.
   BEDS.
   TELEPHONE.
   TV.
   AC.
END TYPES RELATION.
```

You want to add a new field, called ROOM_SIZE, to indicate the size of the room. The easiest way to add a field to a relation is to define a global field and simply name that field in the CHANGE RELATION statement. You can create an indirect command file containing both statements:

```
DEFINE FIELD ROOM_SIZE
    DESCRIPTION IS /* Size in square feet */
    DATATYPE SIGNED WORD.

CHANGE RELATION TYPES.
   DEFINE ROOM_SIZE.
END TYPES RELATION.
```

After you execute this command file, each record in the TYPES relation has an added field that can contain up to four digits. To store data in those fields, you can write a program or procedure to modify each existing record with the new data.

Example 2

The RESERVATION relation needs a field to indicate whether a guest has checked out of the hotel or has extended his or her stay. This field is useful when attempting to reserve that room for another guest. Two possible values for this field are "Y" and "N".

Because the OVERNITE database already contains a global field called STANDARD_FLAG with the required characteristics, the hotel can use this field to create the new field using the BASED ON clause:

```
CHANGE RELATION RESERVATION.
    DEFINE CHECKED_OUT BASED ON STANDARD_FLAG.
END RESERVATION RELATION.
```

4.2 Changing Fields

As your understanding of your database grows, you might want to add new characteristics to your fields. For example, users of VAX DATATRIEVE might want to access the data in the database, and you can add VAX DATATRIEVE characteristics to your fields. Furthermore, you may also want to take advantage of the BASED ON qualifier to declare local names for globally defined fields.

The following examples show how to use the the CHANGE FIELD and CHANGE RELATION statements to add details to the field definitions.

Example 1

You added a global field called ROOM_SIZE to the database, with this definition:

```
DEFINE FIELD ROOM_SIZE
    DESCRIPTION IS /* Size in square feet */
    DATATYPE SIGNED WORD
```

You can add characteristics to this global field with the CHANGE FIELD statement. To add a DATATRIEVE EDIT_STRING clause, use the CHANGE FIELD statement:

```
CHANGE FIELD ROOM_SIZE
     EDIT_STRING FOR DATATRIEVE IS "9999".
```

Example 2

This example adds characteristics to both global and local field definitions. In the first part, the CHANGE FIELD statement adds a MISSING_VALUE clause to the global field ROOM_NUMBER. This field was previously defined with data type TEXT and a VALID IF clause.

In the second part of the example you must use the CHANGE RELATION statement to add two local DATATRIEVE support clauses. The QUERY_HEADER and QUERY_NAME clauses are only in effect for the ROOM_NUMBER field in this particular relation, HOTEL.

```
CHANGE FIELD ROOM_NUMBER
       MISSING_VALUE IS "---". ----------①

CHANGE RELATION HOTEL.
    CHANGE ROOM_NUMBER
        QUERY_HEADER FOR DATATRIEVE IS "ROOM"----------②
        QUERY_NAME FOR DATATRIEVE IS "ROOM".
END HOTEL RELATION.
```

Example 3

The global field definitions for the OVERNITE database include a field called STANDARD_RATE. All local fields in the database that contain money values can base their definitions on this field definition.

If the OVERNITE database did not have the STANDARD_RATE generic money field, you could use the following commands to provide this kind of data type consistency. You could specify DATATRIEVE support clauses in the local field definition to further distinguish one local money field from another.

```
DEFINE FIELD  STANDARD_RATE
            DESCRIPTION IS /* Standard money field  */
                      DATATYPE IS SIGNED LONGWORD SCALE -2
                      EDIT_STRING FOR DATATRIEVE IS "$$$$.$$".

CHANGE RELATION RATES.
  CHANGE GOV_RATE BASED ON STANDARD_RATE
    QUERY_HEADER FOR DATATRIEVE IS "GOVERNMENT"/"RATE".
  CHANGE GROUP_RATE BASED ON STANDARD_RATE
    QUERY_HEADER FOR DATATRIEVE IS "GROUP"/"RATE".
  CHANGE STD_RATE BASED ON STANDARD_RATE
    QUERY_HEADER FOR DATATRIEVE IS "STANDARD"/"RATE".
END RATES RELATION.

CHANGE RELATION BILLING.
  CHANGE SERVICE_CHARGE BASED ON STANDARD_RATE.
END BILLING RELATION.
```

This process requires the following steps:

1. Define a new field, STANDARD_RATE, with the generic characteristics
2. Change the local definition of each money field to refer to the global STANDARD_RATE field definition
3. Add local characteristics to the fields (for DATATRIEVE support). Here, you add QUERY_HEADER information to make the fields more "local".

Note

Data stored in any of these relations would not be affected by the change. All you have done is change certain characteristics associated with the field definition. If the old and new fields have different data types, Rdb/VMS performs the data type conversion automatically.

4.3 Changing the Database

The CHANGE DATABASE statement takes the same clauses and parameters as the DEFINE DATABASE statement. In addition, you must use the CHANGE DATABASE statement to enable the Rdb/VMS after-image journaling feature.

In most cases, Rdb/VMS manages the allocation of database pages automatically to allow more space for the database. You can experience acceptable performance using all the default database parameter values.

However, you can use the CHANGE DATABASE statement to increase or decrease the following database parameters:

- The size of the EXTENT by which the database can grow on disk using the DATABASE EXTENT clause.
- The size of the SNAPSHOT file that is created by default with the DEFINE DATABASE statement using the SNAPSHOT ALLOCATION clause.
- The size of the EXTENT by which the SNAPSHOT file can expand using the SNAPSHOT EXTENT clause.
- The size of the EXTENT by which the database expands on a multidisk volume. Although Rdb/VMS creates a single file database, you can specify that it reside on a multidisk volume and you can control how the database file is distributed across each disk of the multidisk volume. Use the multivolume extent clause of the CHANGE DATABASE statement.

You can also use the CHANGE DATABASE statement to change the filename or pathname of the database.

For complete details about changing database parameters, see Chapter 4 of the *VAX Rdb/VMS Guide to Database Administration and Maintenance.*

In addition, you can use the CHANGE DATABASE statement to access the after-image journaling feature of Rdb/VMS. Once you have defined the database and its entities, stored data in the database relations, and tested the system, the database is ready for use. At this point you can use the CHANGE DATABASE statement to turn on the after-image journaling feature.

When after-image jounraling is in effect, Rdb/VMS records all committed database updates in a special file called the *after-image journal file.* You can maintain this file on a regular basis by storing daily or weekly copies on another backup medium, such as tape. In the event that a software or hardware failure causes your database to become corrupt, you can use the journal file to recover the database to a known, uncorrupted state. You can then resume normal database access.

To create an after-image journal file and start the process of journaling all committed changes to the database, you use the JOURNAL FILE IS clause with CHANGE DATABASE statement. The following example starts an after-image journal file for the OVERNITE database.

```
CHANGE DATABASE PATHNAME 'CDD$TOP.HOTEL.OVERNITE'
      JOURNAL FILE IS DISK2:[JOURNAL]OVERNITE.
```

The first transaction that attaches to the database automatically opens the journal file. You have the option of issuing an OPEN statement to open the after-image journal file. The OPEN statement permits Rdb/VMS to map certain data structures automatically for all users of the database. Opening the journal file with the OPEN statement results in some performance improvement when the database is invoked.

You can use the NOJOURNAL clause of the CHANGE DATABASE statement to turn off journaling.

For complete details about the after-image journaling feature, see Chapter 3 of the *VAX Rdb/VMS Guide to Database Administration and Maintenance.*

4.4 Deleting Relations

If you have sufficient access privileges, deleting relations is easy. You simply name the relation you want to delete in the DELETE RELATION statement. However, you cannot delete a relation if other relations depend on it for:

- View definitions
- COMPUTED BY fields
- Constraint definitions

If you try to delete a relation with such dependencies, Rdb/VMS returns an error message.

Suppose you decide you no longer want to store information about individual rooms in the TYPES relation. The number of types has increased until there are nearly as many types as there are rooms. Therefore, you decide to eliminate the TYPES relation and store all the room information in the HOTEL relation.

If you have not yet stored data in the database, deleting a relation is simple. You merely name the relation in a delete statement. If there is data, deleting relations becomes more complicated, since you will lose the data in the TYPES relation if you delete it. If data is present, you must transfer the data to an existing relation before deleting the old one. Perform the following steps:

1. Change the HOTEL relation to add new fields (derived from TYPES).

```
CHANGE RELATION HOTEL.
  DEFINE RATE_CODE.
  DEFINE BEDS.
  DEFINE TELEPHONE.
  DEFINE TV.
  DEFINE AC.
  DEFINE ROOM_SIZE.
END HOTEL RELATION.
```

2. Using a CROSS and a MODIFY statement, copy the data from the TYPES relation to the HOTEL relation.

```
FOR H IN HOTEL
   CROSS T IN TYPES OVER ROOM_TYPE
     MODIFY H USING
                 H.RATE_CODE    = T.RATE_CODE;
                 H.BEDS         = T.BEDS;
                 H.TELEPHONE    = T.TELEPHONE;
                 H.TV           = T.TV;
                 H.AC           = T.AC;
                 H.ROOM_SIZE    = T.ROOM_SIZE
     END_MODIFY
END_FOR
```

3. Delete the TYPES relation. The TYPES relation is used in the constraint definition RATE_CODE_EXISTS. Therefore, you must delete that constraint first.

```
DELETE CONSTRAINT RATE_CODE_EXISTS.
DELETE RELATION TYPES.
```

4. Moreover, you may no longer need the ROOM_TYPE field in HOTEL, because it was there only to allow joins with TYPES. Delete the ROOM_TYPE field from HOTEL. However, since you defined ROOM_TYPE as an index in HOTEL, you must delete the index before you can delete the field from the relation.

```
DELETE INDEX HOTEL_ROOM_TYPE.

CHANGE RELATION HOTEL.
  DELETE ROOM_TYPE.
END.
```

4.5 Deleting Fields

To delete a field, you issue the DELETE FIELD statement at the RDO> prompt. If a field is referred to in a relation definition, you must use the CHANGE RELATION statement to delete the field from the relation. If a field is referred to in a constraint or index definition, you must first delete the constraint or index.

For example, if you performed the steps in the previous section, you see that the ROOM_TYPE field is now obsolete. This field served as a link between the HOTEL and TYPES relation. The field has already been deleted from the both relations. You can now delete the ROOM_TYPE field from the database:

```
DELETE FIELD ROOM_TYPE.
```

4.6 Deleting the Database

You can delete the entire database by simply typing:

```
DELETE DATABASE PATHNAME 'OVERNITE'.
```

This statement deletes the database file, the snapshot file, and the CDD definitions. Do not use the INVOKE statement first.

You can delete the CDD definitions only with the following statement:

```
DELETE PATHNAME 'CDD$TOP.BOOKEEP.OVERNITE'.
```

This statement deletes the CDD directory and all its descendants. However, it does not delete the database file or the snapshot file. If you need to recreate the CDD definitions from the metadata in the database file, you can use the following statement:

```
INTEGRATE DATABASE 'OVERNITE' IN 'CDD$TOP.BOOKEEP.OVERNITE'
```

The preceding statement copies the metadata from the system relations in the OVERNITE database file into the CDD. If a database already exists with the same CDD path name, you will receive an error message. Do not use the INVOKE statement before you issue the INTEGRATE statement. The INTEGRATE statement automatically invokes the database after the database definitions are successfully entered into the CDD.

You can use the INTEGRATE statement if a CDD definition is corrupt; that is, if the CDD definitions no longer match the definitions in the database file. You can also use the INTEGRATE statement if the CDD was not installed when you defined the database, or if you neglected to put all the data definitions in the CDD.

Definitions for the OVERNITE Database A

The definitions shown in the following command file create the OVERNITE database used thoughout this book. Included are the relation, field, and view definitions for the OVERNITE database.

```
SET VERIFY

SET OUTPUT CREATE_OVERNITE.LOG

!
!  Define OVERNITE database
!

DEFINE DATABASE "DISK4:[NEWDB]OVERNITE"
         IN "CDD$TOP.ACCOUNTING.HOTEL".
!
! Global field definitions for OVERNITE database
!

DEFINE FIELD  STANDARD_DATE
            DESCRIPTION IS /* Standard date field  */
                      DATATYPE IS DATE
                       MISSING_VALUE IS "18-NOV-1858 00:00:00.00"
                       EDIT_STRING FOR DATATRIEVE IS "MM/DD/YY".

DEFINE FIELD  STANDARD_RATE
            DESCRIPTION IS /* Standard money field  */
                      DATATYPE IS SIGNED LONGWORD SCALE -2
                      EDIT_STRING FOR DATATRIEVE IS "$$$$.$$".

DEFINE FIELD  STANDARD_FLAG
            DESCRIPTION IS /* Standard flag field for any use  */
                      DATATYPE IS TEXT SIZE IS 1
                       MISSING_VALUE IS "?".
```

(continued on next page)

```
DEFINE FIELD  ROOM_NUMBER
            DESCRIPTION IS /* (PK for HOTEL) Hotel room number  */
                      DATATYPE IS TEXT SIZE IS 3
                       VALID IF
                       (ROOM_NUMBER GT "100" AND
                       ROOM_NUMBER LT "500" AND
                       ROOM_NUMBER NE "200" AND
                       ROOM_NUMBER NE "300" AND
                       ROOM_NUMBER NE "400") AND
                       ROOM_NUMBER NOT MISSING
              EDIT_STRING FOR DATATRIEVE IS "XXX"
             QUERY_HEADER FOR DATATRIEVE IS "ROOM"/"NUMBER"
               QUERY_NAME FOR DATATRIEVE IS "RNUM".

DEFINE FIELD  ROOM_TYPE
            DESCRIPTION IS /* Hotel room type code  */
                      DATATYPE IS TEXT SIZE IS 2
                       VALID IF
                            ROOM_TYPE EQ "S"
                         OR ROOM_TYPE EQ "D"
                         OR ROOM_TYPE EQ "SS"
                         OR ROOM_TYPE MISSING
                         MISSING_VALUE IS "??"
              EDIT_STRING FOR DATATRIEVE IS "XX"
             QUERY_HEADER FOR DATATRIEVE IS "ROOM"/"TYPE"
               QUERY_NAME FOR DATATRIEVE IS "RTYPE".

DEFINE FIELD  RATE_CODE
            DESCRIPTION IS /* Hotel room rate code  */
                      DATATYPE IS TEXT SIZE IS 2
                       VALID IF
                           RATE_CODE EQ "A"
                        OR RATE_CODE EQ "B"
                        OR RATE_CODE EQ "C"
                        OR RATE_CODE MISSING
                       MISSING_VALUE IS "?"
             EDIT_STRING FOR DATATRIEVE IS "XX"
            QUERY_HEADER FOR DATATRIEVE IS "RATE"/"CODE"
              QUERY_NAME FOR DATATRIEVE IS "RATCOD".

DEFINE FIELD  GUEST_NAME
            DESCRIPTION IS /* Guest name  */
                      DATATYPE IS TEXT SIZE IS 15
                       VALID IF
                           GUEST_NAME NOT MISSING
            QUERY_HEADER FOR DATATRIEVE IS "GUEST"/"NAME"
              QUERY_NAME FOR DATATRIEVE IS "NAME".

DEFINE FIELD  CITY
            DESCRIPTION IS /* City of Hotel Guest  */
                      DATATYPE IS TEXT SIZE IS 10
                      VALID IF
                          CITY NOT MISSING
            QUERY_HEADER FOR DATATRIEVE IS "CITY".
```

```
DEFINE FIELD  STATE
            DESCRIPTION IS /* State of hotel guest  */
                      DATATYPE IS TEXT SIZE IS 2
                      VALID IF
                          STATE NOT MISSING
            QUERY_HEADER FOR DATATRIEVE IS "STATE".


DEFINE FIELD  POSTAL_CODE
            DESCRIPTION IS /* Postal code of hotel guest  */
                      DATATYPE IS TEXT SIZE IS 5
                      VALID IF
                          POSTAL_CODE NOT MISSING
            QUERY_HEADER FOR DATATRIEVE IS "POSTAL"/"CODE".

DEFINE FIELD  LENGTH_OF_STAY
            DESCRIPTION IS /* Number of days guest stays in hotel  */
                      DATATYPE IS SIGNED WORD
                       VALID IF
                           LENGTH_OF_STAY GT 0
                        OR LENGTH_OF_STAY MISSING
                       MISSING_VALUE IS -1
            QUERY_HEADER FOR DATATRIEVE IS "LENGTH"/"OF STAY"
              QUERY_NAME FOR DATATRIEVE IS "STAY".


DEFINE FIELD  PARTY_SIZE
            DESCRIPTION IS /* Number of people in guest party  */
                      DATATYPE IS SIGNED WORD
                       VALID IF
                           PARTY_SIZE GT 0
                       AND PARTY_SIZE NOT MISSING
            QUERY_HEADER FOR DATATRIEVE IS "PARTY"/"SIZE"
              QUERY_NAME FOR DATATRIEVE IS "P_SIZE".

DEFINE FIELD  SERVICE_DESCRIP
            DESCRIPTION IS /* Description of service rendered  */
                      DATATYPE IS TEXT SIZE IS 20
                       MISSING_VALUE IS "MISCELLANEOUS"
            QUERY_HEADER FOR DATATRIEVE IS "SERVICE"/"DESCRIPTION"
              QUERY_NAME FOR DATATRIEVE IS "S_DESCR".

DEFINE FIELD  SERVICE_CODE
            DESCRIPTION IS /* Service code of service rendered  */
                      DATATYPE IS TEXT SIZE IS 2
                      VALID IF
                          SERVICE_CODE NOT MISSING
            QUERY_HEADER FOR DATATRIEVE IS "SERVICE"/"CODE"
              QUERY_NAME FOR DATATRIEVE IS "S_CODE".

DEFINE FIELD  TELEPHONE
            DESCRIPTION IS /* Is telephone in the hotel room  */
                      DATATYPE IS TEXT SIZE IS 1
                       VALID IF
                           TELEPHONE EQ "Y"
                        OR TELEPHONE EQ "N"
                        OR TELEPHONE MISSING
                       MISSING_VALUE IS "?"
            QUERY_HEADER FOR DATATRIEVE IS "TELEPHONE"
              QUERY_NAME FOR DATATRIEVE IS "PHONE".
```

(continued on next page)

```
DEFINE FIELD  TV
            DESCRIPTION IS /* Is TV in the hotel room  */
                      DATATYPE IS TEXT SIZE IS 1
                       VALID IF
                           TV EQ "Y"
                        OR TV EQ "N"
                        OR TV MISSING
                       MISSING_VALUE "?"
            QUERY_HEADER FOR DATATRIEVE IS "TELEVISION"
              QUERY_NAME FOR DATATRIEVE IS "TV".

DEFINE FIELD  AC
            DESCRIPTION IS /* Does room have air conditioning  */
                      DATATYPE IS TEXT SIZE IS 1
                       VALID IF
                           AC EQ "Y"
                        OR AC EQ "N"
                        OR AC MISSING
                       MISSING_VALUE "?"
            QUERY_HEADER FOR DATATRIEVE IS "AIR"/"CONDITIONING"
              QUERY_NAME FOR DATATRIEVE IS "AIR".

DEFINE FIELD  BEDS
            DESCRIPTION IS /* Number of beds in hotel room  */
                      DATATYPE IS SIGNED WORD
                       VALID IF
                           BEDS GT 0
            QUERY_HEADER FOR DATATRIEVE IS "NUMBER"/"OF BEDS"
              QUERY_NAME FOR DATATRIEVE IS "NUM_BED".


!************************************************************
!  Define Relations in OVERNITE Database
!************************************************************
!
!  Define HOTEL relation

DEFINE RELATION HOTEL.
   ROOM_NUMBER.
   ROOM_TYPE.
END HOTEL RELATION.

!
!  Define TYPE relation

DEFINE RELATION TYPE.
   ROOM_TYPE.
   RATE_CODE.
   BEDS.
   TELEPHONE.
   TV.
   AC.
END TYPE RELATION.
```

```
!  Define RESERVATION relation

DEFINE RELATION RESERVATION.
   GUEST_NAME.
   CITY.
   STATE.
   POSTAL_CODE.
   ROOM_NUMBER.
   LENGTH_OF_STAY.
   PARTY_SIZE.
   RESERVE_DATE
           BASED ON STANDARD_DATE
           QUERY_HEADER FOR DATATRIEVE IS "RESERVATION"/"DATE"
           QUERY_NAME FOR DATATRIEVE IS "RESRV_DATE".
   ARRIVE_DATE
           BASED ON STANDARD_DATE
           QUERY_HEADER FOR DATATRIEVE IS "ARRIVAL"/"DATE"
           QUERY_NAME FOR DATATRIEVE IS "A_DATE".
   DEPART_DATE
           BASED ON STANDARD_DATE
           QUERY_HEADER FOR DATATRIEVE IS "DEPARTURE"/"DATE"
           QUERY_NAME FOR DATATRIEVE IS "D_DATE".
   CONFIRMED
         BASED ON STANDARD_FLAG
         QUERY_HEADER FOR DATATRIEVE IS "RESERVATION"/"CONFIRMED"
         QUERY_NAME FOR DATATRIEVE IS "RESRV_CONF".
   CHECK_OUT
         BASED ON STANDARD_FLAG
         QUERY_HEADER FOR DATATRIEVE IS "CHECKED"/"OUT"
         QUERY_NAME FOR DATATRIEVE IS "CHK_OUT".
   ROOM_RATE
         BASED ON STANDARD_RATE
         QUERY_HEADER FOR DATATRIEVE IS "ROOM"/"RATE"
         QUERY_NAME FOR DATATRIEVE IS "R_RAT".
END RESERVATION RELATION.

!  Define RATES relation

DEFINE RELATION RATES.
   RATE_CODE.
   STD_RATE
         BASED ON STANDARD_RATE
         QUERY_HEADER FOR DATATRIEVE IS "STANDARD"/"RATE"
         QUERY_NAME FOR DATATRIEVE IS "ST_RATE".
   GOV_RATE
          COMPUTED BY (STD_RATE * 0.90)
         QUERY_HEADER FOR DATATRIEVE IS "GOVERNMENT"/"RATE"
         QUERY_NAME FOR DATATRIEVE IS "G_RATE".
   GROUP_RATE
          COMPUTED BY (STD_RATE * 0.86)
         QUERY_HEADER FOR DATATRIEVE IS "GROUP"/"RATE"
         QUERY_NAME FOR DATATRIEVE IS "GRP_RATE".
END RATES RELATION.
```

(continued on next page)

```
!  Define BILLING relation

DEFINE RELATION BILLING.
   ROOM_NUMBER.
   SERVICE_CHARGE
         BASED ON STANDARD_RATE
         QUERY_HEADER FOR DATATRIEVE IS "SERVICE"/"CHARGE"
         QUERY_NAME FOR DATATRIEVE IS "S_CHG"
   TX_DATE
          BASED ON STANDARD_DATE
          QUERY_HEADER FOR DATATRIEVE IS "TRANSACTION"/"DATE".
   SERVICE_DESCRIP.
   SERVICE_CODE.
END BILLING RELATION.

!
!  Define GUEST View
!

DEFINE VIEW GUEST OF R IN RESERVATION
 CROSS B IN BILLING OVER ROOM_NUMBER.
        R.GUEST_NAME.
        R.ROOM_NUMBER.
        TOTAL_ROOM
          COMPUTED BY
                (R.LENGTH_OF_STAY * R.ROOM_RATE).
        TOTAL_SERVICE
          COMPUTED BY
                TOTAL X.SERVICE_CHARGE OF X IN BILLING
           WITH X.ROOM_NUMBER = R.ROOM_NUMBER.
        TOTAL_BILL
          COMPUTED BY
                ((R.LENGTH_OF_STAY * R.ROOM_RATE) +
                (TOTAL X.SERVICE_CHARGE OF X IN BILLING
                  WITH X.ROOM_NUMBER = R.ROOM_NUMBER)).

END VIEW.

SET NOOUTPUT
```

Index

HOW TO ORDER ADDITIONAL DOCUMENTATION

DIRECT TELEPHONE ORDERS

In Continental USA and Puerto Rico call **800–258–1710**	In Canada call **800–267–6146**	In New Hampshire, Alaska or Hawaii call **603–884–6660**

DIRECT MAIL ORDERS (U.S. and Puerto Rico*)

DIGITAL EQUIPMENT CORPORATION
P.O. Box CS2008
Nashua, New Hampshire 03061

DIRECT MAIL ORDERS (Canada)

DIGITAL EQUIPMENT OF CANADA LTD.
940 Belfast Road
Ottawa, Ontario, Canada K1G 4C2
Attn: P&SG Business Manager

INTERNATIONAL

DIGITAL EQUIPMENT CORPORATION
P&SG Business Manager
c/o Digital's local subsidiary
or approved distributor

Internal orders should be placed through the Software Distribution Center (SDC), Digital Equipment Corporation, Northboro, Massachusetts 01532

*Any prepaid order from Puerto Rico must be placed
with the Local Digital Subsidiary:
809–754–7575

VAX Rdb/VMS
Guide to Database
Design and Definition
AA-N034B-TE

Reader's Comments

Note: This form is for document comments only. DIGITAL will use comments submitted on this form at the company's discretion. If you require a written reply and are eligible to receive one under Software Performance Report (SPR) service, submit your comments on an SPR form.

Did you find this manual understandable, usable, and well-organized? Please make suggestions for improvement. ______

Did you find errors in this manual? If so, specify the error and the page number. ______

Please indicate the type of user/reader that you most nearly represent.

- ☐ Assembly language programmer
- ☐ Higher-level language programmer
- ☐ Occasional programmer (experienced)
- ☐ User with little programming experience
- ☐ Student programmer
- ☐ Other (please specify) ______

Name ______ Date ______

Organization ______

Street ______

City ______ State ______ Zip Code or Country ______

-------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

ATTN: DISG Documentation ZKO2-2/N53
DIGITAL EQUIPMENT CORPORATION
110 SPIT BROOK ROAD
NASHUA, N.H. 03062

-------Do Not Tear - Fold Here and Tape --

digital

DIGITAL EQUIPMENT CORPORATION

Printed in U.S.A.