Oracle Rdb7™

Guide to Database Design and Definition

Release 7.0

Part No. A41749-1

Guide to Database Design and Definition

Release 7.0

Part No. A41749-1

Copyright © 1984, 1996, Oracle Corporation. **All rights reserved.**

This software contains proprietary information of Oracle Corporation; it is provided under a license agreement containing restrictions on use and disclosure and is also protected by copyright law. Reverse engineering of the software is prohibited.

The information contained in this document is subject to change without notice. If you find any problems in the documentation, please report them to us in writing. Oracle Corporation does not warrant that this document is error free.

Restricted Rights Legend Programs delivered subject to the DOD FAR Supplement are 'commercial computer software' and use, duplication and disclosure of the programs shall be subject to the licensing restrictions set forth in the applicable Oracle license agreement. Otherwise, programs delivered subject to the Federal Acquisition Regulations are 'restricted computer software' and use, duplication and disclosure of the programs shall be subject to the restrictions in FAR 52.227-14, Rights in Data—General, including Alternate III (June 1987). Oracle Corporation, 500 Oracle Parkway, Redwood City, CA 94065.

The programs are not intended for use in any nuclear, aviation, mass transit, medical, or other inherently dangerous applications. It shall be the licensee's responsibility to take all appropriate fail-safe, back up, redundancy and other measures to ensure the safe use of such applications if the programs are used for such purposes, and Oracle disclaims liability for any damages caused by such use of the programs.

Oracle is a registered trademark of Oracle Corporation, Redwood City, California. Oracle CDD/Repository, Oracle Expert, Oracle Rdb, Oracle RMU, Oracle Trace, and Rdb7 are trademarks of Oracle Corporation, Redwood City, California.

All other company or product names are used for identification purposes only and may be trademarks of their respective owners.

Contents

Send Us Your Comments xxi

Preface xxiii

Technical Changes and New Features xxvii

1 Designing a Relational Database

1.1 Understanding Relational Concepts and Terminology 1–1
1.2 Choosing a Design Method 1–3
1.3 Understanding Logical and Physical Database Design 1–4
1.3.1 Logical Design Concepts 1–4
1.3.2 Physical Design Concepts 1–5
1.3.3 Oracle Rdb On-Disk Structures 1–6
1.3.4 Storage Methods 1–8
1.3.5 Retrieval Methods 1–9
1.4 Introducing the Sample Databases 1–10

2 Making a Logical Database Design

2.1 Analyzing Requirements 2–1
2.2 Translating Requirements into Data Items 2–2
2.3 Mapping Relationships Among Columns and Tables 2–3
2.4 Normalizing Tables 2–8
2.5 Analyzing Transactions 2–9
2.5.1 Tracing Transaction Paths Through the Logical Model 2–10
2.5.2 Prototype Transactions in SQL 2–13
2.6 Archiving Information 2–14
2.7 Developing a Volume Table 2–14

3 Defining a Database

3.1 Overview of Database Definition 3–1
3.2 Summary of Database Elements 3–3
3.3 Options for Executing Statements That Define a Database 3–4
3.4 Using the Repository When You Define a Database 3–5
3.5 Creating the Database and Specifying Its Characteristics 3–6
3.5.1 Specifying a Database with Subordinate Elements 3–9
3.5.2 Creating Databases Using Multiple Character Sets 3–10
3.5.3 Specifying an Alias 3–12
3.5.4 Reserving Slots for After-Image Journal Files 3–12
3.5.5 Reserving Slots for Storage Areas 3–14
3.5.6 Specifying Storage Areas for Multifile Databases 3–14
3.5.7 Creating a Default Storage Area 3–15
3.5.8 Creating Several Storage Areas in Parallel 3–15
3.5.9 Compressing System Indexes 3–16
3.5.10 Choosing Among Snapshot File Options 3–17
3.5.11 Allocating Disk Space and Memory 3–19
3.5.12 Setting Database Key (Dbkey) Scope 3–21
3.5.13 Specifying Who Can Open a Database 3–21
3.5.14 Looking for More Detailed Information About Database Definition 3–22
3.6 Naming Database Elements 3–22
3.7 Using Data Types 3–23
3.8 Specifying the Length of Characters in Octets or Characters 3–25
3.9 Including Comments in Definitions of Elements 3–26
3.10 Creating Domains 3–27
3.10.1 Creating Domains Based on Repository Fields 3–27
3.10.2 Specifying Characteristics of Domains 3–28
3.10.2.1 Specifying Character Sets for Domains 3–31
3.10.2.2 Specifying Default Values for Domains 3–32
3.10.2.3 Specifying Collating Sequences 3–33
3.10.2.4 Specifying SQL Formatting Clauses 3–34
3.10.2.5 Specifying Domain Constraints 3–34
3.11 Creating Tables 3–36
3.11.1 Creating Tables Based on Repository Definitions 3–36
3.11.2 Specifying Elements of Tables 3–38
3.11.2.1 Specifying the Data Type of Columns 3–40
3.11.2.2 Assigning Character Sets to Columns 3–42
3.11.2.3 Specifying the COMPUTED BY Clause 3–42
3.11.2.4 Specifying Default Values for Columns 3–44
3.11.2.5 Creating Constraints 3–45
3.11.2.6 Implementing a UNIQUE OR NULL Constraint 3–51

3.12 Enforcing Referential Integrity Through Constraints and Triggers 3–53
3.12.1 Using Constraints to Enforce Referential Integrity 3–54
3.12.2 Using Triggers to Enforce Referential Integrity 3–54
3.13 Creating Triggers to Invoke External Functions 3–58
3.14 Creating Indexes 3–60
3.14.1 Creating Sorted Indexes 3–61
3.14.2 Creating Hashed Indexes 3–63
3.14.3 Deciding Between an Index and a Constraint to Enforce Unique Column Values 3–65
3.14.4 Deciding When Indexes Are Beneficial 3–66
3.14.5 Creating Indexes Concurrently 3–69
3.14.6 Creating Compressed Indexes 3–70
3.14.6.1 Creating Run-Length Compressed Indexes 3–71
3.14.6.2 Creating SIZE IS Segment-Truncated Indexes 3–72
3.14.6.3 Creating Mapping Values Compressed Indexes 3–72
3.15 Creating Temporary Tables 3–72
3.15.1 Creating Global and Local Temporary Tables 3–75
3.15.2 Creating Declared Local Temporary Tables 3–79
3.15.3 Estimating Virtual Memory for Temporary Tables 3–82
3.16 Creating Views 3–83
3.16.1 Creating the CURRENT_JOB View 3–85
3.16.2 Creating the CURRENT_SALARY View 3–87
3.16.3 Creating the CURRENT_INFO View 3–88
3.16.4 Creating Views to Calculate Dates 3–90

4 Implementing a Multifile Database

4.1 Deciding on a Storage Design for Your Multifile Database 4–1
4.2 Understanding General Storage Options for a Multifile Database 4–3
4.3 Assigning Tables and Indexes to Storage Areas 4–7
4.3.1 Specifying Storage Map Options 4–10
4.3.2 Enforcing Storage Map Partitioning 4–11
4.4 Choosing Uniform or Mixed Page Format 4–13
4.4.1 Advantages of Uniform Page Format 4–13
4.4.2 Advantages of Mixed Page Format 4–14
4.5 Choosing Read/Write, Read-Only, or Write-Once Storage Areas 4–17
4.6 Achieving Optimal Performance for Queries and Update Operations 4–19
4.6.1 Achieving Optimal Performance for Range Retrieval 4–19
4.6.2 Achieving Optimal Performance for Exact Match Retrieval 4–21
4.6.3 Achieving Optimal Performance for Join Operations or Update of Related Rows 4–24
4.6.4 Achieving Optimal Performance for Retrieving Some Columns in a Table 4–28

4.6.5 Achieving Optimal Performance for List Data 4–30
4.6.5.1 Storing List Data in Isolation 4–30
4.6.5.2 Storing List Data Randomly or Sequentially 4–32
4.6.5.3 Storing List Data on WORM Devices 4–33
4.7 Setting Sorted Index Characteristics for Performance 4–34
4.7.1 Calculating the Size of Sorted Indexes 4–35
4.7.2 Specifying Fullness Percentages for Sorted Indexes 4–38
4.7.3 Balancing Node Size and Fullness Percentages 4–38
4.8 Setting Database and Storage Area Parameters When Using Hashed Indexes 4–40
4.8.1 Understanding the Page Overhead and Record Types 4–41
4.8.2 Calculating the Size of Fixed and Variable Page Overhead 4–42
4.8.3 Calculating the Size of Hashed Index Structures 4–43
4.8.4 Calculating the Size of Hashed Indexes for the Sample Database . . . 4–45
4.8.5 Calculating the Size of Data Rows 4–47
4.8.6 Calculating the Page Size 4–51
4.8.7 Calculating the File Allocation Size 4–52
4.9 Implementing Placement and Clustering Strategies Using Hashed Indexes 4–57
4.9.1 Separate Areas, No Placement Clause 4–58
4.9.2 Separate Areas, with Placement Clause 4–59
4.9.3 Same Area, with Placement Clause (One I/O Operation) 4–59
4.9.4 Clustering: Add Child Rows, Separate Storage Area, with Placement Clause 4–60
4.9.5 Shadowing: Child and Parent in Separate Areas, with Placement Clause 4–61
4.9.6 Clustering: Child and Parent Rows and Hashed Index All in the Same Area, with Placement Clause 4–62

5 Implementing a Multischema Database

5.1 Understanding Multischema Databases 5–1
5.2 Creating Multischema Databases 5–2
5.3 Creating Catalogs 5–3
5.4 Creating Schemas 5–3
5.5 Naming Elements 5–5
5.5.1 Using Qualified Names 5–6
5.5.2 Using Stored Names and SQL Names 5–7
5.6 Using Aliases 5–9
5.7 Creating Schema Elements 5–10

6 Loading Data

6.1 Improving Performance When Loading Data 6–1
6.2 Using the PLACEMENT ONLY RETURNING DBKEY Clause 6–5
6.2.1 Using the INSERT Statement to Get the Dbkey for Each Row 6–6
6.2.2 Sorting the Dbkeys in Ascending Order 6–9
6.2.3 Reading the Rows in Sorted Order and Storing Them in the Database 6–10
6.3 Modifying the Database to Load Data 6–13
6.3.1 Adjusting Database-Wide Parameters 6–13
6.3.2 Adjusting Storage Area Parameters 6–14
6.3.3 Modifying Tables 6–15
6.3.4 Modifying Indexes 6–15
6.3.5 Modifying Storage Maps 6–16
6.4 Troubleshooting Data Load Operations 6–16
6.5 Loading Data from a Flat File Using SQL Programs 6–19
6.5.1 Using the SQL Module Language and BASIC to Load Data 6–20
6.5.2 Using the SQL Module Language, COBOL, and Repository Definitions to Load Data 6–24
6.5.3 Using SQL Precompiled C Programs to Load Data 6–28
6.6 Loading and Unloading Data Using the RMU Load and RMU Unload Commands 6–32
6.6.1 Improving Performance While Using the RMU Load Command 6–34
6.6.2 Understanding the Format of the Record Definition File 6–37
6.6.3 Loading Data into a Database Table from a Flat File 6–38
6.6.4 Loading Null Values 6–40
6.6.5 Unloading Null Values 6–42
6.6.6 Restructuring Databases Using the RMU Load and RMU Unload Commands 6–43
6.6.7 Loading and Unloading Data from Oracle Rdb Databases 6–45
6.6.8 Loading Data from One Database to Another 6–46
6.7 Using Parallel Load 6–52
6.7.1 Using Parallel Load Without a Plan File 6–56
6.7.2 Generating a Plan File with RMU Load 6–58
6.7.3 Using Parallel Load with a Plan File 6–59
6.8 Modifying Database Definitions Following a Load Operation 6–60

7 Modifying Databases and Storage Areas

7.1 Modifying Databases and Storage Areas — A Summary 7–2
7.2 Modifying Data Definitions While Users Are Attached to the Database 7–10
7.3 Freezing Data Definition Changes 7–18
7.4 Modifying Database Characteristics 7–19
7.4.1 Enabling After-Image Journaling 7–23
7.4.2 Adding After-Image Journal Files 7–25
7.4.3 Modifying Allocation Characteristics for After-Image Journal Files 7–26
7.4.4 Modifying the JOURNAL FAST COMMIT Options 7–26
7.4.5 Modifying Extent Values for the Database 7–27
7.4.6 Modifying the Maximum Number of Users 7–28
7.4.7 Modifying the Maximum Number of Cluster Nodes 7–29
7.4.8 Modifying Database Lock Characteristics 7–29
7.4.9 Selecting Locking Levels for Storage Areas 7–32
7.4.10 Enabling or Disabling Global Buffers 7–33
7.4.11 Modifying the Buffer Size 7–33
7.4.12 Modifying the Number of Local Database Buffers 7–33
7.4.13 Modifying the Number of Database Recovery Buffers 7–34
7.4.14 Controlling Snapshot Files 7–34
7.4.15 Using Deferred Snapshot Files 7–37
7.4.16 Modifying Extent Characteristics for Snapshot Files 7–38
7.4.17 Modifying the Allocation for Snapshot Files 7–38
7.5 Modifying the Requirement for Using the Repository 7–40
7.6 Modifying Storage Areas and Storage Area Parameters 7–40
7.6.1 Adding New Storage Areas for Multifile Databases 7–42
7.6.2 Adjusting Storage Area Parameters to Cluster Rows 7–44
7.6.3 Adjusting the RDB$SYSTEM Storage Area 7–47
7.6.4 Moving Storage Areas 7–48
7.6.5 Moving Read/Write Data to Write-Once Storage Areas 7–50
7.6.6 Moving Data from a Write-Once Storage Area 7–52
7.6.7 Adding List Data to Write-Once Storage Areas 7–52
7.6.8 Modifying Read/Write Storage Areas to Read-Only Storage Areas . . . 7–54
7.6.9 Deleting Storage Areas 7–54
7.7 Modifying Indexes 7–58
7.7.1 Modifying Sorted Indexes 7–59
7.7.2 Modifying Hashed Indexes 7–60
7.7.3 Disabling Indexes 7–63
7.8 Deleting Indexes 7–64
7.9 Modifying Storage Maps 7–65
7.9.1 Creating Storage Maps for Tables That Contain Data 7–72

7.9.2 Moving Certain System Tables to Separate Storage Areas 7–73
7.10 Deleting Storage Maps 7–75
7.11 Reorganizing Databases 7–75
7.11.1 Reorganizing a Single-File Database into a Multifile Database 7–76
7.11.2 Reorganizing a Database for Special Use 7–79
7.11.3 Creating a Copy of the Database 7–82
7.11.4 Creating a Copy of an Empty Database 7–84
7.12 Moving Databases and Database Files 7–85
7.13 Deleting Databases, Database Files, and Repository Definitions 7–87

8 Modifying Database Elements

8.1 Modifying and Deleting Domains 8–1
8.2 Modifying and Deleting Tables 8–4
8.2.1 Deleting Tables 8–4
8.2.2 Deleting Tables Quickly 8–6
8.2.3 Modifying Tables That Are Used in Views or Have Indexes 8–6
8.2.4 Modifying Columns 8–7
8.2.5 Modifying Column Data Types 8–10
8.2.6 Modifying Columns That Include Date-Time Data Types 8–12
8.2.7 Adding, Modifying, and Dropping Default Values from a Column 8–13
8.2.8 Modifying the Name of a Table or the Name or Position of a Column 8–16
8.2.9 Modifying and Deleting Tables in Multischema Databases 8–17
8.3 Modifying and Deleting Constraints 8–18
8.4 Modifying and Deleting Triggers 8–19
8.5 Deleting Views 8–21
8.6 Deleting Schemas in Multischema Databases 8–22
8.7 Deleting Catalogs in Multischema Databases 8–23

9 Defining Database Protection

9.1 Planning for Database Security 9–1
9.2 Understanding Privilege Checking for SQL Statements 9–2
9.2.1 Introducing Access Control Entries (ACEs) 9–5
9.2.2 Introducing ACL-Style and ANSI/ISO-Style Privileges 9–6
9.2.3 Privileges Required for Data Manipulation and Data Definition 9–8
9.2.4 Building Access Control Lists 9–14
9.2.5 Putting the Access Control List in Order 9–17
9.3 Granting and Revoking Privileges 9–19
9.3.1 Defining Protection for Databases 9–21
9.3.2 Defining Protection for Tables 9–22
9.3.3 Defining Protection for Columns 9–24

9.3.4 Restricting Access to Tables by Using Views 9–25
9.3.5 Restricting Access to a Subset of Rows 9–27
9.3.6 Using Views to Maintain Role-Oriented Access 9–29
9.3.7 Defining Default Protection 9–30
9.4 Verifying Protection for a Database 9–31
9.4.1 Privileges with Override Capability 9–33
9.5 Understanding Privilege Checking for Oracle RMU Commands 9–35
9.5.1 Using Oracle RMU Privileges 9–35
9.5.2 Using Oracle RMU Privileges with Databases Created with Version 4.1 or Earlier 9–46
9.6 Restricting Database Creation 9–46
9.7 Securing Shareable Oracle Rdb Definitions in the Repository 9–47

10 Using Oracle Rdb with Oracle CDD/Repository

10.1 Overview of the Repository 10–1
10.1.1 Repository Naming Conventions 10–2
10.1.2 Using CDO 10–3
10.1.3 Criteria for Using the Repository with Oracle Rdb Databases 10–4
10.2 Deciding Whether to Require the Repository 10–5
10.3 Creating New Repository Definitions 10–6
10.4 Defining Record-Level Constraints in the Repository 10–14
10.5 Modifying Repository Definitions Using CDO 10–17
10.5.1 Using the CDO DEFINE Commands 10–17
10.5.2 Using the CDO CHANGE Commands 10–19
10.6 Modifying Repository Definitions and Database Files 10–21
10.7 Understanding How the Repository Is Updated 10–22
10.7.1 Automatically Updating the Repository and the Database File Using SQL 10–23
10.7.2 Receiving an Error on Updating the Repository 10–23
10.7.3 Storing Initial Definitions in the Repository but Updating Only the Database File 10–23
10.7.4 Not Storing Initial Definitions in the Repository and Updating Only the Database File 10–27
10.8 Updating Repository Definitions Using SQL 10–28
10.9 Integrating Domains and Tables Using Database Files 10–30
10.10 Updating the Database File Using Repository Definitions 10–31
10.11 Integrating Domains and Tables Using Repository Definitions 10–35
10.12 Using SQL to Delete Definitions 10–36
10.12.1 Removing All Links Between a Database and the Repository 10–36
10.12.2 Deleting Links with Database Definitions 10–37
10.12.3 Deleting Repository Definitions 10–40
10.13 Using CDO to Delete Repository Definitions 10–41

10.14 Changing the Database File Name Using the Repository 10–43

Index

Examples

2–1 Modeling a Read-Only Transaction 2–13
3–1 Creating the Database 3–7
3–2 Creating a Database and Specifying Subordinate Elements 3–10
3–3 Creating a Database Using Multiple Character Sets 3–11
3–4 Specifying How SQL Interprets the Length of Characters 3–25
3–5 Creating Domains Using the FROM Path-Name Clause 3–27
3–6 Creating Domains 3–29
3–7 Specifying Default Values 3–32
3–8 Specifying a Constraint for a Domain 3–35
3–9 Defining Fields and Records with Oracle CDD/Repository 3–36
3–10 Creating a Table Using the FROM Path-Name Clause 3–37
3–11 Creating a Table and Specifying the Data Type of Columns 3–40
3–12 Creating a Table with Columns Based on Domains 3–41
3–13 Creating a Table with One Column Based on Domains 3–41
3–14 Specifying Default Values for Columns 3–45
3–15 Creating Column Constraints 3–47
3–16 Creating Table Constraints Based on Other Tables 3–48
3–17 Displaying Table Constraints 3–49
3–18 Creating a Trigger to Delete All Information About an Employee . . . 3–55
3–19 Creating a Trigger to Prevent Deleting a Row 3–56
3–20 Calling External Functions from Triggers 3–59
3–21 Calling External Functions from Triggers to Reduce I/O 3–60
3–22 Creating a Sorted Index Using the RANKED keyword 3–62
3–23 Compressing Duplicate Index Entries 3–62
3–24 Creating a Sorted Index 3–63
3–25 Creating a Hashed Index 3–65
3–26 Creating a Global Temporary Table 3–75
3–27 Creating Stored Modules That Use Global Temporary Tables 3–76
3–28 Sharing Data in Global Temporary Tables 3–77
3–29 Creating Local Temporary Tables and Stored Modules That Use the Table 3–78

3–30	Isolating Data in Local Temporary Tables	3–79
3–31	Declaring a Local Temporary Table in Interactive SQL	3–79
3–32	Using Declared Local Temporary Tables in Stored Procedures	3–80
3–33	Creating the CURRENT_JOB View	3–87
3–34	Creating the CURRENT_SALARY View	3–88
3–35	Creating the CURRENT_INFO View	3–89
3–36	Creating a View That Contains Records for Employees with 15 or More Years of Service	3–90
4–1	Creating the Multifile personnel_db Database	4–2
4–2	Assigning a Table and an Index to a Storage Area	4–7
4–3	Partitioning the JOB_HISTORY Table	4–8
4–4	Partitioning a Hashed Index	4–9
4–5	Creating Indexes and Storage Maps Without Overflow Areas	4–9
4–6	Specifying Threshold Values for Uniform Areas	4–11
4–7	Enforcing Storage Map Partitioning	4–12
4–8	Creating Write-Once Storage Areas on WORM Optical Devices	4–17
4–9	Creating Storage Maps for Write-Once Storage Areas	4–18
4–10	Optimizing Performance for Range Retrieval Queries	4–19
4–11	Creating a Hashed Index	4–23
4–12	Clustering Related Rows from Two Tables	4–26
4–13	Partitioning the EMPLOYEES Table Vertically	4–28
4–14	Partitioning the EMPLOYEES Table Vertically and Horizontally	4–29
4–15	Finding the Sizes of Columns in a Table	4–49
4–16	Placing Rows and the Hashed Index in Separate Storage Areas and Not Using the Placement Clause	4–59
4–17	Placing Rows and the Hashed Index in Separate Storage Areas and Using the Placement Clause	4–59
4–18	Placing Rows and the Hashed Index in the Same Storage Area, Using the Placement Clause	4–60
4–19	Placing Parent and Child Rows in One Storage Area, Hashed Indexes in a Separate Area, Using the Placement Clause	4–60
4–20	Placing Parent Rows and Hashed Indexes in the Same Storage Area, Child Rows in a Separate Storage Area, Using the Placement Clause	4–62
4–21	Placing Parent and Child Rows and Hashed Indexes in the Same Storage Areas, Using the Placement Clause	4–63
5–1	Creating a Multischema Database	5–2
5–2	Creating a Catalog	5–3

5–3	Creating a Schema	5–3
5–4	Creating a Schema with Subordinate Elements	5–4
5–5	Displaying Stored Names	5–7
5–6	Specifying Stored Names	5–8
5–7	Using an Alias	5–9
5–8	Creating Domains in Multischema Databases	5–11
5–9	Creating Tables That Refer to Objects in Other Schemas	5–11
5–10	Creating Views That Refer to Tables in Other Schemas	5–13
5–11	Creating Triggers That Refer to Objects in Other Schemas	5–14
6–1	Using the PLACEMENT ONLY Clause to Extract Dbkeys from a Table	6–6
6–2	Loading the Sorted Data	6–11
6–3	BASIC Program That Calls an SQL Module to Load Data	6–21
6–4	Using an SQL Module to Load Data	6–22
6–5	COBOL Program That Calls an SQL Module to Load Data	6–25
6–6	Loading Data Using an SQL Module	6–27
6–7	Loading Data Using an SQL Precompiled C Program	6–29
6–8	Unloading a Table Using the RMU Unload Command	6–37
6–9	Loading Additional Rows into a Table Using the RMU Load Command	6–39
6–10	Loading Null Values from Empty Strings	6–41
6–11	Loading Null Values	6–41
6–12	Unloading Null Values	6–43
6–13	Loading a Table Using the RMU Load Command	6–44
6–14	Restructuring a Table Using the RMU Load and RMU Unload Commands	6–46
6–15	Creating a Command Procedure to Unload Data	6–48
6–16	Unloading Data Using the RMU Unload Command	6–49
6–17	Creating a Command Procedure to Load Data	6–49
6–18	Loading Data Using the RMU Load Command	6–52
6–19	Using Parallel Load	6–57
6–20	Generating a Plan File	6–58
6–21	Using a Plan File for Parallel Load	6–60
7–1	Disallowing Data Definition Changes	7–18
7–2	Displaying Settings for Database and Storage Area Parameters	7–20
7–3	Enabling After-Image Journaling	7–24
7–4	Adding Journal Files	7–25

7–5	Modifying the Allocation Value for .aij Files	7–26
7–6	Modifying the JOURNAL FAST COMMIT Attribute	7–27
7–7	Modifying Extent Values	7–27
7–8	Modifying Extent Options	7–28
7–9	Modifying the Maximum Number of Database Users	7–28
7–10	Modifying the Maximum Number of Cluster Nodes	7–29
7–11	Modifying Adjustable Lock Granularity	7–31
7–12	Disabling Carry-Over Locks	7–31
7–13	Setting the Lock Timeout Interval	7–32
7–14	Enabling Global Buffers by Node	7–33
7–15	Modifying the Number of Buffers	7–34
7–16	Modifying the Number of Recovery Buffers	7–34
7–17	Displaying Current Snapshot File Settings	7–35
7–18	Disabling Snapshot Files	7–36
7–19	Determining If Snapshot File Transactions Are in Progress	7–37
7–20	Specifying Deferred Snapshot Files	7–38
7–21	Modifying Extent Characteristics for Snapshot Files	7–38
7–22	Determining the Current Size of a Snapshot File	7–39
7–23	Increasing the Allocation Size of Snapshot Files	7–39
7–24	Using the DICTIONARY IS NOT REQUIRED Option	7–40
7–25	Modifying Storage Areas	7–41
7–26	Adding a Storage Area and Specifying Parameters	7–43
7–27	Using ALTER DATABASE, ALTER INDEX, and ALTER STORAGE MAP Statements	7–44
7–28	Using EXPORT and IMPORT Statements to Modify the RDB$SYSTEM Storage Area	7–47
7–29	Moving a Storage Area and Related Snapshot Files to a Different Disk Device	7–49
7–30	Moving a Storage Area and Related Snapshot Files to Two Different Disk Devices	7–49
7–31	Moving List Data from a Read/Write Storage Area to a Write-Once Storage Area	7–51
7–32	Moving List Data from a Write-Once Storage Area to a Read/Write Storage Area	7–52
7–33	Adding New List Data to a Write-Once Storage Area	7–53
7–34	Modifying a Read/Write Storage Area to Read-Only Access	7–54
7–35	Deleting a Storage Area Using the RESTRICT Keyword	7–56
7–36	Deleting a Storage Area Using the CASCADE Keyword	7–57

7–37	Attempting to Delete an Updatable Storage Area	7–57
7–38	Attempting to Delete a Storage Area	7–58
7–39	Modifying an Index Definition	7–60
7–40	Partitioning a Hashed Index Across Two Storage Areas	7–62
7–41	Adding Partitions to Indexes Without Overflow Areas	7–62
7–42	Disabling the Maintenance of an Index	7–64
7–43	Modifying the STORE Clause of the Storage Map Definition	7–69
7–44	Specifying the PLACEMENT VIA INDEX Option in a Storage Map Definition	7–69
7–45	Adding a Storage Area to a Storage Map Definition	7–70
7–46	Reorganizing Rows Across Old and New Storage Areas	7–70
7–47	Adding Partitions to Storage Maps Without Overflow Areas	7–70
7–48	Removing Overflow Partitions and Moving Existing Data	7–71
7–49	Specifying Threshold Values for New Areas	7–72
7–50	Creating a Storage Map for Tables Containing Data	7–73
7–51	Deleting a Storage Map	7–75
7–52	Creating an Interchange File Using the EXPORT Statement	7–77
7–53	Reorganizing a Database Using the IMPORT Statement	7–78
7–54	Creating an Interchange File	7–80
7–55	Using an IMPORT Statement to Reorganize a Database	7–80
7–56	Copying a Database	7–83
7–57	Copying the Database and Moving the RESUME_LISTS Storage Area to a WORM Optical Disk Device	7–83
7–58	Copying an Empty Database Using the EXPORT Statement	7–84
7–59	Copying an Empty Database Using the IMPORT Statement	7–84
7–60	Using the TRACE Clause in an IMPORT Operation to Check the Number of I/O Operations and CPU Time Required	7–85
7–61	Moving a Database	7–86
8–1	Modifying Domain Definitions to Add Default Values	8–2
8–2	Dropping the Default Value from a Domain	8–2
8–3	Modifying Domains to Change Domain Constraints	8–2
8–4	Deleting Tables	8–5
8–5	Using the TRUNCATE TABLE Statement to Delete Data from Tables	8–6
8–6	Modifying Tables That Contain Views and Indexes	8–6
8–7	Modifying and Deleting Columns	8–8
8–8	Changing Data Types in a Table	8–10

8–9	Modifying the Default Value of an Existing Column	8–13
8–10	Adding Columns with Default Values to Tables	8–14
8–11	Adding Columns Without Propagating Default Values to Previously Stored Rows	8–15
8–12	Dropping the Default Value from a Column	8–16
8–13	Modifying and Deleting Constraints	8–18
8–14	Modifying Constraints That Refer to Other Tables	8–19
8–15	Modifying and Deleting Triggers	8–20
9–1	Issuing SHOW PROTECTION Statements	9–15
9–2	Denying Privileges to a Group of Users	9–20
9–3	Defining Protection on a Database	9–21
9–4	Defining Protection on a Table	9–22
9–5	Defining Column Protection	9–24
9–6	Creating a View to Restrict Access to the Table	9–26
9–7	Restricting Access with View Definitions	9–26
9–8	Adding a Column and a Trigger to Track Users	9–27
9–9	Preventing Modification of a Column with a Trigger	9–28
9–10	Creating a View That Restricts Access to Certain Records	9–28
9–11	Revoking Protection on the Underlying Table	9–29
9–12	Creating a View to Check for Role-Oriented Privileges	9–30
9–13	Defining Default Protection	9–30
9–14	Verifying ACLs	9–32
9–15	Issuing the SHOW PRIVILEGES Statement	9–33
9–16	Displaying Oracle RMU Privileges	9–35
9–17	Setting Privileges for Oracle RMU Commands	9–36
10–1	Defining Shareable Fields	10–8
10–2	Checking Field Definitions	10–9
10–3	Defining Records	10–10
10–4	Using CDO Definitions to Create an Oracle Rdb Database with SQL	10–12
10–5	Creating Record-Level Constraints	10–14
10–6	Using the CDO DEFINE FIELD Command	10–17
10–7	Determining What Version of a Definition Is Used by a Database	10–19
10–8	Using the CDO CHANGE FIELD Command	10–20
10–9	Modifying Repository Definitions Using the INTEGRATE Statement with the ALTER DICTIONARY Clause	10–24
10–10	Storing Existing Database File Definitions in the Repository	10–28

10–11 Modifying a Domain Definition in the Repository Using the Definition from the Database 10–31
10–12 Updating the Database File Using the Repository Definitions 10–32
10–13 Modifying a Table Definition in the Database Using the Definition from the Repository 10–35
10–14 Removing Links to the Repository 10–36
10–15 Attempting to Drop a Domain Used by Another Database 10–38
10–16 Using the DROP DOMAIN Statement to Delete a Link with a Database 10–38
10–17 Using the ALTER TABLE Statement to Delete a Link with a Database 10–39
10–18 Deleting Definitions from the Repository Using the DROP PATHNAME Statement 10–41
10–19 Determining Owners of a Repository Field Definition 10–42
10–20 Using the CDO DELETE GENERIC Command 10–42
10–21 Changing the Database File Name in the Repository 10–43

Figures

1–1 EMPLOYEES Table 1–2
1–2 Separate Files of a Multifile Database 1–9
2–1 Entity-Relationship (E-R) Map 2–8
2–2 Transaction Paths for the Sample Database 2–11
2–3 Consolidated Transaction Map 2–12
3–1 Logical Model of the Sample Databases 3–2
3–2 Snapshot Transaction Time Line 3–17
4–1 Partitioning a Table Vertically and Horizontally 4–5
5–1 Multischema Database with Multiple Catalogs and Schemas 5–2
6–1 Using Parallel Load 6–54
9–1 Relationship Between Generic-Style Privileges and ACL- and ANSI/ISO-Style Privileges 9–3
9–2 Privileges to Access Oracle Rdb Databases 9–4
10–1 Centralized Design with the Repository 10–5
10–2 Shareable Fields in the Repository 10–37

Tables

1–1 The Entity: EMPLOYEES Table, Representing Data at the Logical Level 1–4
2–1 DEPARTMENTS Table 2–4
2–2 JOB_HISTORY Table 2–4
2–3 One-to-One and One-to-Many Relationships in the Sample Database 2–4
2–4 Volume Table for the Personnel Database 2–15
3–1 Calculating Memory Usage for Temporary Tables 3–82
4–1 Calculating the Fixed and Variable Overhead for a Page 4–42
4–2 Calculating the Size of Hashed Indexes 4–44
4–3 Calculating the Size of Hashed Indexes for the Mf_personnel Database 4–45
4–4 Column Sizes in the EMPLOYEES Table 4–48
4–5 Column Sizes in the JOB_HISTORY Table 4–48
4–6 Calculating the Fixed Overhead for a Page 4–50
4–7 Calculating the Data Row Overhead for a Page 4–50
4–8 Calculating the Page Size 4–51
4–9 Calculating the File Allocation Size 4–52
4–10 Calculating the File Allocation Size to Store 100 Data Pages 4–54
4–11 Calculating the SPAM Pages and Adding These Pages to the Estimated File Allocation Size 4–54
7–1 Adjusting Storage and Memory Use Parameters 7–4
7–2 Updating Data Definitions While Users Are Attached to the Database 7–10
7–3 Updating to Database-Wide Parameters While Users Are Attached to the Database 7–13
7–4 Columns and Keys for the JOB_ASSIGNMENTS Table 7–42
7–5 Summary of Modifying Storage Map Options and Effect on Rows (Moved/Not Moved) 7–67
7–6 Optional System Tables and Their Storage Map Names 7–73
7–7 Tables, Storage Areas, and Storage Maps for the Multifile mf_personnel Database 7–76
9–1 Privileges Required for DML and DDL Operations 9–9
9–2 Privilege Override Capability 9–34
9–3 Privileges Required for Oracle RMU Commands 9–38
9–4 Privileges Required for RMU Commands on Digital UNIX 9–43

10–1	Summary of CDO Pieces Tracking Commands	10–3
10–2	How CREATE DATABASE and ATTACH Statements Affect Repository Updates	10–22

Send Us Your Comments

Oracle Corporation welcomes your comments and suggestions on the quality and usefulness of this publication. Your input is an important part of the information used for revision.

You can send comments to us in the following ways:

- Electronic mail — nedc_doc@us.oracle.com
- FAX — 603-897-3334 Attn: Oracle Rdb Documentation
- Postal service

```
Oracle Corporation
Oracle Rdb Documentation
One Oracle Drive
Nashua, NH  03062
USA
```

If you like, you can use the following questionnaire to give us feedback. (Edit the online release notes file, extract a copy of this questionnaire, and send it to us.)

Name ______________________ Title ______________________

Company ______________________ Department ______________________

Mailing Address ______________________ Telephone Number ______________________

____

Book Title ______________________ Version Number ______________________

- Did you find any errors?
- Is the information clearly presented?
- Do you need more information? If so, where?

- Are the examples correct? Do you need more examples?
- What features did you like most about this manual?

If you find any errors or have any other suggestions for improvement, please indicate the chapter, section, and page number (if available).

Preface

Oracle Rdb is a general-purpose database management system based on the relational data model.

This manual describes how to design a relational database, how to use the data definition statements of the Oracle Rdb structured query language (SQL) to create and modify a database, and how to protect your database. In addition, it demonstrates how to load data into a database and how to use the repository.

Intended Audience

If you have not designed a database before, this manual will help you to analyze an information management problem and show you how to use your analysis to design a database.

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

How This Manual Is Organized

This manual contains the following chapters:

Chapter 1	Introduces concepts of the relational data model and database design.
Chapter 2	Describes techniques for creating a logical design of an Oracle Rdb database.
Chapter 3	Describes how to create a database, including database elements such as tables and domains.
Chapter 4	Describes various physical database designs for multifile databases.
Chapter 5	Describes how to create a multischema database.

Chapter 6	Demonstrates how to load data into an Oracle Rdb database using SQL programs and the RMU Load and RMU Unload commands, and describes data loading strategies and troubleshooting data loading operations.
Chapter 7	Describes how to modify databases and database characteristics such as journaling characteristics, storage areas, and storage maps.
Chapter 8	Describes how to modify database elements such as domains, tables, views, and columns.
Chapter 9	Shows how to create and remove privileges for database access using the SQL GRANT and REVOKE statements and how to control privileges for database maintenance operations with Oracle RMU privileges.
Chapter 10	Shows how to implement an Oracle Rdb database with shareable domains and tables using Oracle CDD/Repository.

Related Manuals

For more information on Oracle Rdb, see the other manuals in this documentation set, especially the following:

- *Oracle Rdb7 Guide to Database Performance and Tuning*
- *Oracle Rdb7 SQL Reference Manual*
- *Oracle RMU Reference Manual*

The *Oracle Rdb7 Release Notes* list all the manuals in the Oracle Rdb documentation set.

Conventions

In this manual, Oracle Rdb refers to Oracle Rdb for OpenVMS and Oracle Rdb for Digital UNIX software.

Oracle CDD/Repository software is referred to as the dictionary, the data dictionary, or the repository.

The SQL interface to Oracle Rdb is referred to as SQL. This interface is the Oracle Rdb implementation of the SQL standard ANSI X3.135-1992, ISO 9075:1992, commonly referred to as the ANSI/ISO SQL standard or SQL92.

OpenVMS means both the OpenVMS Alpha and the OpenVMS VAX operating systems.

This manual uses icons to identify information that is specific to an operating system or platform. Where material pertains to more than one platform or operating system, combination icons or generic icons are used. For example:

Digital UNIX	This icon denotes the beginning of information specific to the Digital UNIX operating system.
OpenVMS VAX OpenVMS Alpha	This icon combination denotes the beginning of information specific to both the OpenVMS VAX and OpenVMS Alpha operating systems.
♦	The diamond symbol denotes the end of a section of information specific to an operating system or platform.

The following conventions are also used in this manual:

. . . (vertical)	Vertical ellipsis points in an example mean that information not directly related to the example has been omitted.
. . .	Horizontal ellipsis points in statements or commands mean that parts of the statement or command not directly related to the example have been omitted.
e, f, t	Index entries in the printed manual may have a lowercase e, f, or t following the page number; the e, f, or t is a reference to the example, figure, or table, respectively, on that page.
< >	Angle brackets enclose user-supplied names.
[ ]	Brackets enclose optional clauses from which you can choose one or none.
$	The dollar sign represents the DIGITAL Command Language prompt in OpenVMS and the Bourne shell prompt in Digital UNIX.

Technical Changes and New Features

This section lists some of the new and changed features for Version 7.0 described in this manual.

The *Oracle Rdb7 Release Notes* provide information on all the new Version 7.0 features and technical changes. The *Oracle Rdb7 Release Notes* also describe current limitations or restrictions.

The major new features and technical changes that are described in this manual include the following:

- Freezing data definition changes

  You can ensure that the data definition of your database does not change by using the METADATA CHANGES ARE DISABLED clause of the ALTER DATABASE, CREATE DATABASE, or IMPORT statements. For more information, see Section 7.3.

- Modifying the database buffer size

  You can modify the database buffer size by using the BUFFER SIZE clause in the ALTER DATABASE statement. In previous versions, you could specify the clause only in the CREATE DATABASE statement. For more information, see Section 7.4.11.

- Specifying how a database opens when you create the database

  You can specify whether a database opens automatically or manually when you create the database. In previous versions, you could specify the OPEN IS clause only in the ALTER DATABASE statement. For more information, see Section 3.5.13.

- Increasing the fanout factor for adjustable lock granularity

  Adjustable lock granularity for previous versions of Oracle Rdb defaulted to a count of 3, meaning that the lock fanout factor was (10, 100, 1000). As databases grow larger, it is becoming necessary to allow these fanout factors to grow to reduce lock requirements for long queries. You can now change the fanout factor by specifying the

COUNT IS clause with the ADJUSTABLE LOCK GRANULARITY clause. For more information, see Section 7.4.8.

- New on-disk structure for B-tree (sorted) indexes

  You can specify that Oracle Rdb use a new on-disk structure for sorted indexes. The new structure allows better optimization of queries, particularly queries involving range retrievals. Oracle Rdb is able to make better estimates of cardinality, reducing disk I/O and lock contention. For more information, see Section 3.14.1.

- Duplicates compression

  If a sorted index allows duplicates, you can store many more records in a small space by using duplicates compression. When you do, Oracle Rdb uses byte-aligned bitmap compression to represent the dbkeys for the duplicate entries, instead of chaining the duplicate entries with uncompressed dbkeys. In addition to the savings in storage space, you minimize I/O, increasing performance. For more information, see Section 3.14.1.

- Creating a default storage area

  You can separate user data from the system data, such as the system tables, by using the DEFAULT STORAGE AREA clause of the CREATE DATABASE or IMPORT statements. This clause specifies that all user data and indexes that are not mapped explicitly to a storage area are stored in the default storage area. For more information, see Section 3.5.7.

- Extending the allocation of storage areas

  You can manually extend the storage area by using the ALLOCATION IS clause of the ALTER STORAGE AREA clause. For more information, see Section 7.6.

- Dropping a storage area with a cascading delete

  You can specify that Oracle Rdb drop a storage area with a cascading delete. When you do, Oracle Rdb drops database objects referring to the storage area. For more information, see Section 7.6.9.

- Vertical partitioning

  You can partition a table vertically as well as horizontally. When you partition a table horizontally, you divide the rows of the table among storage areas according to data values in one or more columns. When you partition a table vertically, you divide the columns of the table among storage areas. Consider partitioning a table vertically when you know that access to some of the columns in a table is frequent,

but access to other columns is occasional. For more information, see Section 4.2.

- Strict partitioning

  You can specify whether or not you can update a partitioning key for a storage map. If you specify that the key is not updatable, the retrieval performance improves because Oracle Rdb can use the partitioning criteria when optimizing the query. For more information, see Section 4.3.2.

- Quickly deleting data in tables

  If you want to quickly delete the data in a table, but you want to maintain the metadata definition of the table (perhaps to reload the data into a new partitioning scheme), you can use the TRUNCATE TABLE statement. For more information, see Section 8.2.2.

- Creating temporary tables

  You can create temporary tables to store temporary results for a short duration, perhaps to temporarily store the results of a query so that your application can act on the results of that query. The data in a temporary table is deleted at the end of an SQL session. For more information, see Section 3.15.

- Support for a parallel load operation

  You can specify a multiprocess RMU Load command (referred to as a parallel load). A parallel load operation can be used to increase the speed of a large load operation. For more information, see Section 6.7.

- Support for determining when and if constraints are evaluated when you use the RMU Load command

  The Constraints=Deferred and Noconstraints qualifiers let you determine when or if constraints are evaluated during a load operation. For more information, see Section 6.6.1.

- Support for deferring index updates when you use the RMU Load command

  The Defer_Index_Updates qualifier lets you specify that non-unique indexes, other than those that define the placement information for data in a storage area, will not be rebuilt until commit time. For more information, see Section 6.1 and Section 6.7.

- Support for generating and using a plan file when you use the RMU Load command

The List_Plan qualifier lets you generate a file containing all the information needed by Oracle RMU to execute a load procedure. This file is called a plan file. For more information, see Section 6.7.2.

The RMU Load Plan command allows you to execute the plan file. For more information, see Section 6.7.3.

- Support for specifying the number of rows sent between processes in a single I/O request when you use the RMU Load command

  A new option, Row_Count, allows you to specify the number of rows that are sent between processes in a single I/O request during a load operation. This option is primarily designed for use with Oracle Rdb for Digital UNIX databases. For more information, see Section 6.6.1.

- Support for storing null values when you use the RMU Load and Unload commands.

  A new option, Null, has been added to the Record_Definition qualifier. This option lets you load and unload null values. For more information, see Section 6.6.4 and Section 6.6.5.

- Removing the links with the repository

  You can remove the link between the repository and database but still maintain the data definitions in both places, using the DICTIONARY IS NOT USED clause of the ALTER DATABASE statement. For more information, see Section 10.12.1.

1 Designing a Relational Database

Effective database design makes data storage and retrieval as efficient as possible.

The main purpose of database design in a multiuser environment is to allow even conflicting needs of users to be supported by the same database system. The various techniques and tools presented in this chapter are the means by which you can create a database design that can satisfy the needs of many users.

Each database has a different set of design trade-offs. The concepts and terms presented in this chapter help you to decide among those trade-offs for the logical and physical design of a relational database.

1.1 Understanding Relational Concepts and Terminology

The concept of a relational database was introduced in the early 1970s by E.F. Codd. Based on the mathematical theory of sets, the relational database represents data in tables, *and only in tables*.

Tables are collections of rows (or records) that consist of columns (or fields). Data is presented in tables such as the one shown in Figure 1–1. The relational database designer structures these tables so that database users can:

- Efficiently access the data in the tables
- Read or write to the appropriate tables

Figure 1–1 EMPLOYEES Table

EMPLOYEE_ID	LAST_NAME	FIRST_NAME	MIDDLE_INITIAL	STATUS_CODE
00164	Toliver	Alvin	A	2
00165	Smith	Terry	D	1
00166	Dietrich	Rick	NULL	1
00167	Kilpatrick	Janet	NULL	1
00168	Nash	Norman	NULL	1
00169	Gray	Susan	O	1
00170	Wood	Brian	NULL	1
00171	D'Amico	Aruwa	NULL	1

Logically and physically, you describe data using *data definition language* (DDL) and access it using *data manipulation language* (DML). Both DDL and DML have syntax similar enough to be considered a single, comprehensive language. This comprehensive language ensures that all data and data describing data (metadata) are accessible in the same way and from the same place. You can define and access Oracle Rdb databases through the Oracle Rdb structured query language (SQL).

Metadata describes data and its uses. Defining metadata is one of the main tasks in the logical design of the database. Examples of metadata are the domains, tables, triggers, constraints, indexes, storage areas, and storage maps that you define and the system tables that Oracle Rdb defines. One way to view the metadata in an Oracle Rdb database is to attach to the sample multifile database, mf_personnel, and use the SQL SHOW statements to show a list of both user-defined and system-defined metadata. The SHOW statements that SQL provides include the following:

- SHOW ALL DOMAINS
- SHOW ALL TABLES
- SHOW TRIGGERS
- SHOW ALL TABLES (CONSTRAINTS) *
- SHOW ALL INDEXES
- SHOW STORAGE AREAS
- SHOW ALL STORAGE MAPS
- SHOW ALL OUTLINES

When you issue an SQL SHOW ALL TABLES statement, you see a list of metadata names; some have dollar signs in the names and others do not. For example, RDB$DATABASE is an example of a table that is system-defined metadata, and the EMPLOYEES table is an example of user-defined metadata.

1.2 Choosing a Design Method

Effective design follows a method. Several steps have to be followed to produce a database design for your application that will be accurate, simple, reliable, and productive. Implementing an existing, proven design method generally yields better results than less systematic approaches. Design methods vary from textbook to textbook, so it is important to choose a method that works for your particular application.

The examples of database design presented in this book adhere to the following model:

- Analyze requirements.
  - Interview key people to learn about the business, the nature of the application, how information is used and by whom, and expectations of end users.
  - Collect business documents such as personnel forms, invoice forms, order forms, and so forth to learn how information is used in the business.
  - Make the relational map that shows the natural groupings of information in table form and the relationship of the information among the tables; see Chapter 2 for more information.
- Create the database design.
  - Create the logical design.
  - Create the physical design.
- Implement the database application.
- Tune the database and database application.
- Maintain the database and database application.

The result of any selected design method is the set of Oracle Rdb tables, as well as the domains, views, constraints, triggers, indexes, storage area definitions, and storage maps associated with those tables.

The outcome of a good design is a database application that is complete, correct, and structured towards specific processing needs.

1.3 Understanding Logical and Physical Database Design

There are two aspects to database design:

- Logical design: understanding the logical relationships among all the objects in the database
- Physical design: implementing an effective way of storing and retrieving these database objects in files on a storage medium

1.3.1 Logical Design Concepts

The process of logical design involves arranging data objects into a series of logical relationships called entities and attributes. An **entity** in Oracle Rdb is a table. In Figure 1–1, an example of an entity is the entire table consisting of data items (columns) and a set of data values (rows). An **attribute** is a column in this table, such as the Employee ID column. A **row** is a set of data values, one value for each column in the row.

In a logical design, you represent these entities graphically, eliminate redundancies, and produce the most useful layouts of the tables that represent the data to the user. You also make models to understand which tables are accessed by which users in which sequences. These are referred to as transactions. It is important to understand what the typical transactions are, as well as which ones are most important.

Effective logical design considers the requirements of different users who need to own, access, and update data. All data at the logical level is represented explicitly by a set of data values in tables.

Table 1–1 shows two sets of data values (two rows) of five data items (five columns) in the EMPLOYEES table. Each row describes a set of values for a different employee.

Table 1–1 The Entity: EMPLOYEES Table, Representing Data at the Logical Level

			Data Items (Columns)		
	Employee ID	**Name**	**Address**	**Zip Code**	**Telephone Number**
Data Values	00165	Smith	10 Main Street	00111	(619) 555–1323
(Rows)	00166	Jones	234 Elm Street	00112	(619) 555–4321

It is important that you organize the data items (columns) in your database so that users with different needs can easily use them. For example, 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.

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, and list the parts of the organization that use it.

The database examples presented in this book assume the following background information:

- The personnel database of a large corporation has to be accessible to all the personnel people in domestic operations who update the employee records of the company on a daily basis.
- The MIS department makes a prototype database that includes all necessary data items (columns) in a single table, EMPLOYEES.
- Several groups provide input of how each would use the EMPLOYEES table.

Before determining the data items (columns) in the EMPLOYEES table and separating those columns into separate tables, the database designer listens to each group, and takes the needs of each into account.

For a discussion of logical database design, see Chapter 2.

1.3.2 Physical Design Concepts

Physical design consists of converting the information gathered during the logical design phase into a description of the physical database. This description optimizes the placement of the physical database structures that represent the logical design to attain the best performance. For example, knowing what the most important transactions are in your database application and the kinds of transactions they are (insert, modify, or delete), you can plan to physically place specific tables on certain disk devices to ensure optimal placement and guarantee the best performance.

Because the physical database description is the result of the integration of all the information about the tables and columns, relationships between tables, and transactions that may add or update rows for a database application, the description is created with a knowledge of how Oracle Rdb stores tables and other structures, such as indexes, on disk.

You can use Oracle Expert for Rdb to optimize the physical design of your Oracle Rdb database. Using Oracle Expert for Rdb, you specify information about the application workload, data volume, and system environment of the database. Oracle Expert applies its design rules (heuristics) to the database and to the information you have supplied. It generates several design reports, as well as a command procedure that (with minimal edits) you can run to create a new database with an optimal physical design. This procedure also unloads any existing data and reloads it in the new database.

You can obtain workload information using Oracle Trace. Oracle Trace collects and reports data and performance information from databases on an event basis (as opposed to products that collect on a timer basis). You can feed the information collected by Oracle Trace to Oracle Expert.

Using Oracle Rdb, it is possible to create and use a database without specifying values for many of the physical storage characteristics. In this case, Oracle Rdb provides default values for the characteristics. While such an approach can be quite useful and successful with smaller databases, it should not be used for larger, more complicated database designs where optimal performance is critical.

Chapter 4 describes the physical design decisions to consider if you design a large database to be used in a production environment with a high transaction throughput.

1.3.3 Oracle Rdb On-Disk Structures

The structure of the database on a disk varies according to the size and complexity of the database you choose to define. You can choose either to store all tables in one file or in separate files. A database that stores tables in one file (file type .rdb) is a **single-file** database. Alternately, you can have a database in which system information is stored in a database root file (file type .rdb) and the data and metadata are stored in one or more storage area files (file type .rda). You can place each table in a separate file, several tables in one file, or one table in separate files. A database with a root file and one or more storage area files is a **multifile** database.

The structure for a single-file database is:

Digital UNIX

- On Digital UNIX, a database directory with the file type .rdb. This directory contains the database root (.rdb) file and the snapshot (.snp) file. ♦
- A database root (.rdb) file, which contains all user data and information about the current status of all database operations. This file contains the system tables that hold metadata information about the structure of the user data stored in the database.
- A snapshot (.snp) file, which contains copies of rows (before-images) that are being modified by users updating the database. In read-intensive applications, use of the snapshot file results in good performance because the snapshot file is used by read-only transactions when that same requested row contained in the storage area file is being updated. This gives the read-only transaction access to the same row that is being updated and a consistent view of the database while not having to wait for update transactions to finish.

The structure for a multifile database is:

Digital UNIX

- On Digital UNIX, a database directory with the file type .rdb. This directory contains the database root (.rdb) file and the snapshot (.snp) file for the database root file. It can also contain storage area (.rda) files and additional snapshot files. ♦
- A database root (.rdb) file, which contains information about the current status of all database operations.
- A storage area (.rda) file, which is the storage area for database system tables. You can create this file by explicitly assigning it, RDB$SYSTEM, in a CREATE STORAGE AREA clause of a CREATE DATABASE statement (as was done with the multifile sample personnel database); or you can let Oracle Rdb create it by default.

  If you do not define a storage area file as RDB$SYSTEM, Oracle Rdb creates such a file with a default file name that is the same as the database root (.rdb) file name.
- One or more .rda files for user data. User data includes table rows and index tree structures. You can specify which tables will be stored in a given area, and, at the same time, you can specify the method of index structure that will be used to retrieve rows from that area.
- A snapshot (.snp) file for each .rda file and for the database root file. An .snp file contains copies of rows (before-images) that are being modified by users updating the database. The snapshot file is used by read-only transactions in read-intensive applications to improve performance. Read-only transactions always have a consistent view of the database for the

duration of their transaction even though some update transactions may be accessing the same rows.

Oracle Rdb also uses the following files:

- A recovery-unit journal (.ruj) file for each user attached to the database, which contains before-images of any modified rows. The .ruj file is used by the recovery process and user processes to roll back transactions or statements. All users who write to the database use the .ruj file except when they are performing batch-update transactions.
- One or more after-image journal (.aij) files (optional), which contain copies of modified rows and modified metadata (an after-image journal) of committed database transactions for a set time period. The .aij files are used to recover the database to its most consistent state (roll the database forward to the last completed transaction) following a database restore operation.
- Temporary files, which contain tables used to execute large, complex queries and to store intermediate results. For example, they are used to perform sort/merge processing on large tables (over 5000 rows) and during database recovery (rollforward) operations.

1.3.4 Storage Methods

When you define a multifile database and specify that data storage files are not on the same disk as the root file, it reduces the likelihood that the disk on which the root file is located will become an input/output (I/O) bottleneck for database operations. Oracle Rdb recommends that you use a multifile database, rather than a single-file database, so that you can take advantage of the performance options available with multifile databases.

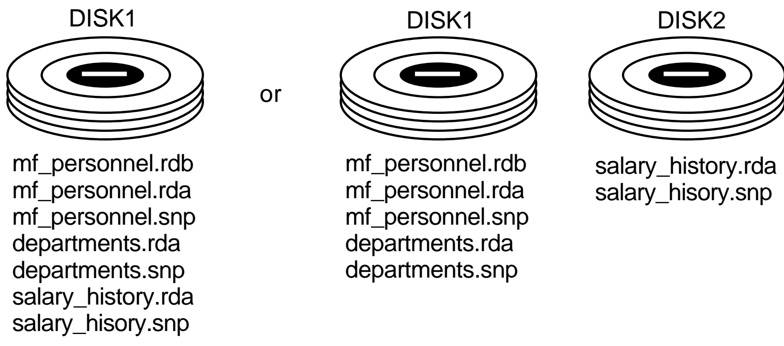
In a multifile database, the database root file contains information about the database storage areas. You can create separate .rda files to contain the data from one or more tables. For every .rda file you create, there is a corresponding .snp file. You can spread these files over more than one disk and you can divide the tables horizontally, vertically, or both.

Horizontal partitioning means that you partition a table so that groups of rows are stored in one storage area and other groups of rows are stored in other storage areas. **Vertical partitioning** means that you partition a table so that some columns are stored in one storage area and other columns are stored in other storage areas.

When you define storage areas, you control the placement of files on disks in the definition statement. In large and complicated databases, you can improve the performance by placing groups of files on different disks, thereby reducing disk I/O contention.

Figure 1–2 shows you can place all files in separate files on one disk or each table in separate files and the SALARY_HISTORY table on a separate disk.

Figure 1–2 Separate Files of a Multifile Database



1.3.5 Retrieval Methods

Oracle Rdb provides several methods for retrieving or accessing data. In the physical design of your database, consider that Oracle Rdb can use one or more of the following methods to retrieve the rows in a table:

- Sequential: locating a row or rows in sequence by retrieving data within a logical area
- Sorted index lookup with value retrieval: using the database key (dbkey) for the value from the index to retrieve the row
- Sorted index only: using data values in the index key pertinent to your query
- Hashed index retrieval: for retrieving exact data value matches
- Dbkey only: retrieving a row through its dbkey

You determine the retrieval method Oracle Rdb chooses by creating one or more sorted or hashed indexes.

Sorted index retrieval provides indexed sequential access to rows in a table. (A sorted index is also called a B-tree index.) By contrast, **hashed index** retrieval, also known as *hash-addressing*, provides direct retrieval of a specific row. Retrieval of a row is based on a given value of some set of columns in the row (called the **search key**).

Use a hashed index primarily for random, direct retrieval when you can supply the entire hashed key on which the hashed index is defined, such as an employee identification number (ID). For this kind of retrieval, input/output operations can be significantly reduced, particularly for tables with many rows and large indexes.

For example, to retrieve a row using a sorted index that is four levels deep, Oracle Rdb may need to do a total of five input/output operations, one for each level of the sorted index and one to retrieve the actual row. By using a hashed index, the number of input/output operations may be reduced to one or two because hashed index retrieval retrieves the row directly.

1.4 Introducing the Sample Databases

You can use a script, located in the sample directory, to create the sample personnel databases.

OpenVMS VAX OpenVMS Alpha

On OpenVMS, the sample directory is defined by the logical name SQL$SAMPLE. ♦

On Digital UNIX, the sample directory is:

```
/usr/lib/dbs/sql/vnn/examples
```

where vnn is the version number, for example, v70. ♦

For more information about creating the sample databases and the files used by the command procedure, see the online file about_sample_databases.txt in the sample directory.

After you create the sample database, you can attach to the database. The following example shows how you can attach to the sample multifile database from interactive SQL:

```
SQL> ATTACH 'FILENAME mf_personnel';
```

With the sample databases, you can generate your own tutorial examples in addition to the examples in this manual.

2 Making a Logical Database Design

This chapter describes techniques for the logical design of an Oracle Rdb database. The topics presented in this chapter include:

- Identifying business rules and collecting data
- Identifying key, non-key, and data items
- Identifying groupings of columns and tables and making a map of them
- Normalizing the columns and tables
- Identifying transaction paths and making a consolidated transaction map
- Determining an archiving strategy
- Making a volume table to allow for future growth

2.1 Analyzing Requirements

The process of identifying business rules, interviewing end users of the database, and manually collecting sample data to be used by the database is known as **requirements analysis**.

Begin by identifying the business requirements that the database will fulfill. These requirements reflect business activities, such as:

- Tracking candidates during the hiring process
- Performing daily updates of the employee master file
- Keeping track of employees' job histories
- Planning salary levels

The requirement that each employee must have a unique identification (ID) number is one such example. Requirements can come from existing policies and procedures or from business documents such as a personnel report. Such requirements help you decide what data to present, as well as when and where to present it.

As part of such data collection, you should:

- Interview end users to understand how business requirements are used
- Determine data relationships, volumes, and characteristics
- Identify what information has to be retained and for how long
- Cross-check all findings

Data collection can include report formats, inquiry formats, computer file layouts, account book formats, and other source documents. You should allow sufficient time to gather the supporting information before implementing and testing a database design.

Use these documents to cross-check the list of tables and columns you develop. The following requirements were utilized for designing the personnel database:

- An employee has a unique employee identification number (ID).
- Employee IDs are used only once.
- There are three classes of employees: hourly, salaried, and contract.
- There is only one start date per employee.
- Personnel groups must access the employee records every day to keep the files updated.
- Department heads need to read an employee's file several months before the anniversary of the employee's start date for salary planning.

Note that as business needs change, some rules and assumptions may change. Personnel management might decide, for example, to consider all employees in only two classes, hourly and salaried, because the third category is unnecessary. When business needs change you will need to update your requirements list and modify the existing logical design of the database.

2.2 Translating Requirements into Data Items

Data items can be categorized as *primary key, foreign key,* or *non-key* items.

Examine each source document (such as an employment application or an employee's personnel file) for a unique data item, that is, an identifying attribute. For example, the unique data item on the application for employment is an internal identification number (ID). The identifying attribute on the tab of each employee's personnel file is an employee ID number. These unique attributes could become the unique columns used to identify the rows of a database table.

Unique data items (ID number, department code) should be used as **primary keys**. In some cases, it may be necessary to consider several data items to arrive at a unique key. Each data item may not be unique, but the combination of items is unique. These several items together can be used to form a compound primary key, or **multisegmented key**. For example, an employee ID number and an educational degree can form a multisegmented key to find employees holding specific degrees.

Consider whether or not each document is the source for a table. If the document is, in fact, a source document, assign the table a meaningful name based on the document. The employee file, for example, is the source document for the EMPLOYEES table. Every significant data item, like Last Name and First Name, gets translated into a column name.

Data items that become column names in one source document may themselves be primary keys in another document. These columns establish a relationship among the tables. The **foreign key** is a column in one table whose data values are required to match a unique value, such as the primary key, of some other table. For example, Department Code exists as a foreign key in the employee file folder, but it serves as a primary key in the list of departments.

2.3 Mapping Relationships Among Columns and Tables

The connections among tables and columns are known as **relationships**. Mapping relationships involves:

- Listing the columns: the primary keys, the foreign keys, and the non-keys
- Identifying the logical connections between tables and columns
- Recording tables and columns on a single table or graphic representation of the database. This graphic representation is known as the entity-relationship (E-R) map.

Begin by identifying the primary key, foreign key, and non-keys for each table. Table 2–1 shows the DEPARTMENT_CODE column as a primary key of the DEPARTMENTS table and MANAGER_ID is a foreign key. The DEPARTMENT_NAME column is a non-key of the DEPARTMENTS table.

Table 2–1 DEPARTMENTS Table

DEPARTMENT_CODE	(primary key)
DEPARTMENT_NAME	(non-key)
MANAGER_ID	(foreign key)
BUDGET_PROJECTED	(non-key)
BUDGET_ACTUAL	(non-key)

Next, identify the tables with which the DEPARTMENTS table has relationships. For example, because each row in the JOB_HISTORY table uses the DEPARTMENT_CODE column as a foreign key, a one-to-many relationship exists between the JOB_HISTORY and DEPARTMENTS tables. That is, one employee can work in only one department (at a time); one department has many employees who have worked in it. The DEPARTMENT_CODE column is a foreign key in the JOB_HISTORY table, as shown in Table 2–2.

Table 2–2 JOB_HISTORY Table

EMPLOYEE_ID	(foreign key)
JOB_CODE	(foreign key)
JOB_START	(non-key)
JOB_END	(non-key)
DEPARTMENT_CODE	(foreign key)
SUPERVISOR_ID	(foreign key)

Further analysis of the sample personnel database turns up the one-to-one (1:1) and one-to-many (1:M) relationships shown in Table 2–3.

Table 2–3 One-to-One and One-to-Many Relationships in the Sample Database

1:1 Relationships	1:M Relationships
EMPLOYEES to DEPARTMENTS	EMPLOYEES to JOB_HISTORY
EMPLOYEES to CANDIDATES	EMPLOYEES to SALARY_HISTORY
	EMPLOYEES to DEGREES

(continued on next page)

Table 2–3 (Cont.) One-to-One and One-to-Many Relationships in the Sample Database

1:1 Relationships	1:M Relationships
	WORK_STATUS to EMPLOYEES
	JOBS to JOB_HISTORY
	DEPARTMENTS to JOB_HISTORY
	COLLEGES to DEGREES

These relationships can be determined in the following way:

- 1:1 Relationships
  - EMPLOYEES to DEPARTMENTS

    One employee can manage one and only one department, and one department can be managed by one and only one employee.
  - EMPLOYEES to CANDIDATES

    One employee can be a candidate for one and only one position at a time, and one candidate can be hired as an employee for one and only one position.
- 1:M Relationships
  - EMPLOYEES to JOB_HISTORY

    An employee's job history information is current only for some period of time; each employee, upon a change in job code, department, job end date, or supervisor, can be associated with additional up-to-date job history information.
  - EMPLOYEES to SALARY_HISTORY

    An employee's salary history information is current only for some period of time; each employee, upon a change in salary, salary start, and salary end dates, can be associated with additional up-to-date salary history information.
  - EMPLOYEES to DEGREES

    One employee can have more than one degree; one particular degree (with the associated information about the year granted, college, degree field, and employee ID) can be achieved by only one employee.
  - WORK_STATUS to EMPLOYEES

One particular work status can be assigned to many employees; one employee can have one and only one work status (at a time).

- JOBS to JOB_HISTORY

  One job can have many employees with that job code; one employee (job history) can only be assigned to one job (at a time).

- DEPARTMENTS to JOB_HISTORY

  One department can have many employees who have worked in it; one employee (job history) can work in only one department (at a time).

- COLLEGES to DEGREES

  One college can have many degrees that can be awarded from it; one particular degree (with the associated information about the year granted, college, degree field, and employee ID) can be awarded by only one college (at a time).

During the later stages of the logical design process, other information needs to be added to the E-R map. The kind of information to keep includes record volumes or the number of rows anticipated to be stored in each table, frequency of use or the retrieval frequency of rows stored in each table, and retention periods or the length of time rows will be stored in each table before being archived. For more details about database volumes, see the *Oracle Rdb7 Guide to Database Performance and Tuning*.

All pertinent business documents obtained in the requirements analysis should be used to check the relationships identified in the E-R map. You may have to revise the E-R map repeatedly until you obtain all of the necessary information to understand the logical design from the user's point of view.

Once you have identified the different kinds of relationships, diagram them to further understand the ways tables are tied together.

A one-to-many relationship is the most common way of tying tables together. Adding foreign keys establishes one-to-many relationships, while deleting foreign keys removes one-to-many relationships. A one-to-many relationship is typically represented as shown in the following figure:

A One-to-Many Relationship

ONE : EMPLOYEES ——————————> **MANY : DEGREES**

A one-to-one relationship can be formed between two tables if you can say for each row in the table that there is one and only one corresponding row in the other table. For example, one employee can manage one and only one department and one department can be managed by only one employee at

a time. A one-to-one relationship is typically represented as shown in the following figure:

A One-to-One Relationship

ONE : EMPLOYEES ——————————> ONE : DEPARTMENTS

The EMPLOYEES and DEPARTMENTS tables are considered to have a one-to-one relationship.

Complex data models usually include additional kinds of relationships, such as:

- Many-to-many relationships

  For example, any instance of an employee's salary history can be associated with more than one instance of that employee's job history; any instance of an employee's job history can be associated with more than one instance of that employee's salary history. You can represent this relationship as a bidirectional arrow, as shown in the following figure:

  A Many-to-Many Relationship

  MANY : SALARY_HISTORY <————————-> MANY : JOB_HISTORY

- Reflexive relationships

  Reflexive relationships can be one-to-one, one-to-many, or even many-to-many. For example, two tables that have mutual one-to-many relationships form a **reflexive** one-to-one relationship. The relationship is reflexive when one employee manages other employees, as shown in the following figure:

  A One-to-One Reflexive Relationship

  ONE : EMPLOYEES ————————> ONE : EMPLOYEES (REFLEXIVE)

Record these relationships on the E-R map. The simplest form for such a map is a simple table or graphic representation. Figure 2–1 shows an E-R map for the personnel database. The figure shows only a few of the relationships between tables, not all the relationships.

Figure 2–1 Entity-Relationship (E-R) Map

Employee Personal Data
Employee ID number
Last name
First name
Middle initial
Building, apt.
Street address
City
State
Postal code
Sex
Birth date
Work status code
Resume

Employee Job History
Employee ID number
Job code
Date of job start
Date of job end
Department code
Supervisor ID number

Job Descriptions
Job code
Wage class
Job title
Minimum salary
Maximum salary

Employee Salary History
Employee ID number
Salary amount
Date of salary start
Date of salary end

Department Descriptions
Department code
Department name
Department manager ID
Projected annual budget
Actual annual budget

Candidates Data
Last name
First name
Middle initial
Candidate status code
Resume

Employee Degrees
Employee ID number
College code
Year of degree
Type of degree
Degree field

College Information
College code
College name
City
State
Postal code

Work Status Descriptions
Work status code
Status description
Status type

NU–2073A–RA

2.4 Normalizing Tables

The process of removing redundant data from the tables of a relational database is called **normalization**. If used appropriately, normalization is the best and easiest way to arrive at an effective logical organization of the tables of a relational database. When you convert data into normalized form, you:

- Reduce a database structure to its simplest form
- Remove redundant columns from tables
- Identify all data that is dependent on other data

There are five recognized degrees of normalization, but data in the third normal form is said to be *fully normalized*. However, the degree of normalization of data depends on the particular application. If data is not sufficiently normalized, many facts repeat in the tables, and you must search through this repeating information for the significant facts and update anomalies that may occur. If data is overnormalized, it takes longer to access the data as more tables must be joined. Generally, each row of the table should present one significant fact.

Effective normalization consists of the following tasks:

- Remove repeating columns from the table to attain the first normal form. For example, if salary history information was stored in the EMPLOYEES table, this information would represent repeating columns.
- Remove data not completely dependent on a primary key from the table to attain the second normal form. That is, if the table contains a multisegmented key and some columns are completely determined by only part of the multisegmented key, move those columns and the part of the key on which they are dependent to a separate table.
- Remove data that does not belong in the table (items wholly dependent on another column or foreign key) to put data in the third normal form.
- Test that each row of the table shows one significant fact.
- Bring back a certain amount of redundancy into the table to improve performance by avoiding the need to do cross-table joins. For example, when you add an employee's FIRST_NAME and LAST_NAME columns to the SALARY_HISTORY table, you are increasing data redundancy. Even though these two columns add overhead to the table, overall performance gains may offset the added redundancy.

2.5 Analyzing Transactions

Transaction analysis brings together the results of requirements analysis and data analysis; it involves translating requirements into data items, mapping relationships among columns and tables, and normalizing tables. During this process, you check to see that the logical database model can support the transactions required by users of the system.

Requirements analysis should yield the following information about each transaction:

- A sequence of steps that shows which tables and columns are accessed and whether they are updated

- Response time and throughput goals
- Times when access to the database is required

Data analysis generates the following information about the logical database model:

- A list of tables, columns, keys, and relationships
- An E-R map

Transaction analysis provides answers to the following questions:

- What transaction paths are required by each transaction?
- What entry points are required?
- What relationships are needed but not yet defined?
- What relationships are defined but not used?
- Which transaction paths sustain the greatest rates of activity?
- Which transaction paths are used repeatedly to get only a single column?

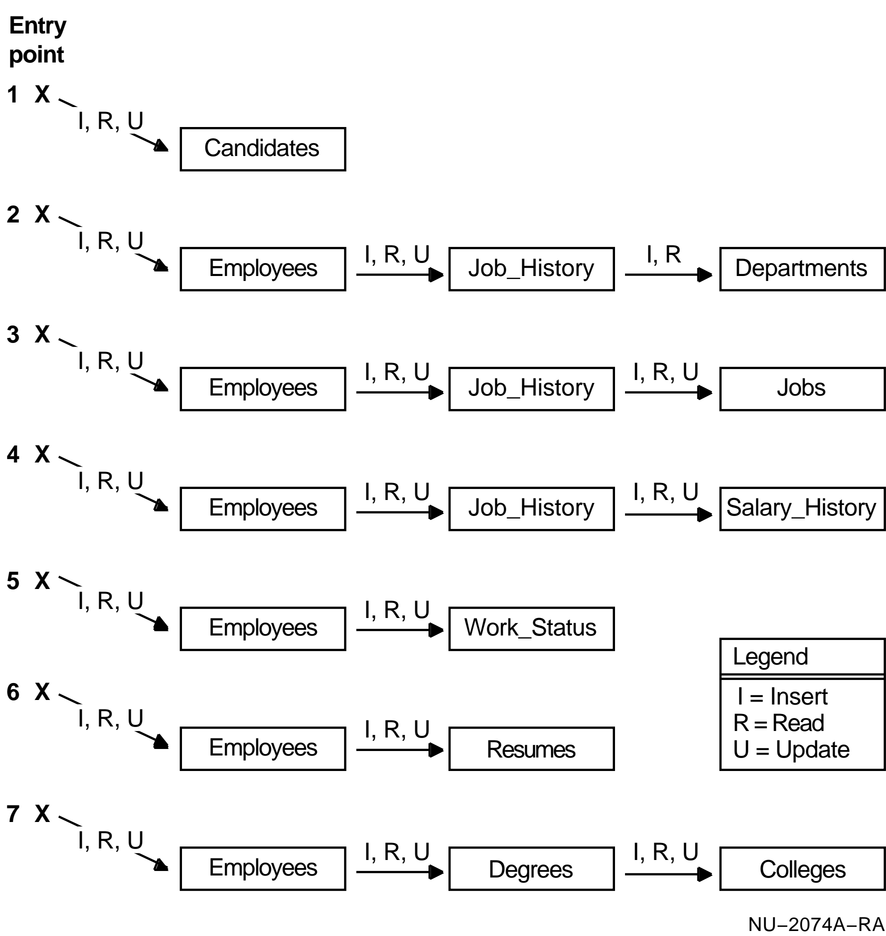
There is one more type of transaction to consider: archiving. Users tend not to consider this issue, but database administration must plan for archiving inactive or obsolete rows. Archiving is a transaction that occurs at specific times in the life of a database. Section 2.6 discusses archiving in more detail.

2.5.1 Tracing Transaction Paths Through the Logical Model

Trace the path of each transaction through the map of the relationships and columns. Figure 2–2 shows the map of the personnel database as revised after data analysis. You may find the following method of building a transaction map helpful. To map transactions:

1. Assign a number to each transaction.
2. Make copies of the third normal form relationships without the arrows between the tables.
3. Draw each transaction path on a separate piece of paper using the following conventions:
   - Mark entry points to the database with an "X."
   - Label each path between tables as R (read), U (update), I (insert), or D (delete).

Figure 2–2 Transaction Paths for the Sample Database



It is helpful to map every transaction, but you gain the greatest benefit by mapping the most critical transactions. In the personnel database, there are six transactions, plus the archiving transaction. The three most critical transactions are:

- Adding (inserting) a new employee's row
- Updating the employee's row with hiring and job history information
- Querying (reading) the employee's row

Each transaction uses one entry point to the database, using key access to a row. From the entry point, the map shows the path of the row access using relationships established in the database. For example, you look for an EMPLOYEES row using the EMPLOYEE_ID key. To add a salary history record, first check for the existence of a particular EMPLOYEE_ID in the EMPLOYEES row. If that employee's row is present, then add a new job history row, if needed, to the JOB_HISTORY table based on that EMPLOYEE_ID, and finally add a new salary history row to the SALARY_HISTORY table based on the same EMPLOYEE_ID. This information can be retrieved in a separate transaction after the transaction that adds these records to the JOB_HISTORY and SALARY_HISTORY tables is committed. Figure 2–3 consolidates these transactions into a single map.

Figure 2–3 Consolidated Transaction Map

Candidates
Archival
Entry point X
Degrees
Colleges
Employees
Work_Status
Jobs
Job_History
Salary_History
Departments
Resumes
NU–2075A–RA

Transaction analysis provides valuable information for database implementation. Further, such analysis can help to identify potential input/output bottlenecks that should be adjusted to improve performance.

2.5.2 Prototype Transactions in SQL

Use your transaction map as a flow diagram to implement a prototype version of your transaction in Oracle Rdb.

Example 2–1 shows a procedure that creates a prototype of the read-only transaction to query an employee's salary history.

Example 2–1 Modeling a Read-Only Transaction

```
$ SQL
SQL> ATTACH 'FILENAME mf_personnel';
SQL> @PROTO1
-- prototype: CURRENT_SALARY transaction
SELECT LAST_NAME, FIRST_NAME, EMPLOYEE_ID, SALARY_START, SALARY_AMOUNT
       FROM EMPLOYEES E NATURAL JOIN SALARY_HISTORY SH
       WHERE SALARY_END IS NULL;
 E.LAST_NAME      E.FIRST_NAME   EMPLOYEE_ID   SH.SALARY_START
   SH.SALARY_AMOUNT
 Toliver          Alvin          00164         14-Jan-1983
        $51,712.00

 Smith            Terry          00165          1-Jul-1982
        $11,676.00

 Dietrich         Rick           00166          7-Aug-1982
        $18,497.00
   .
   .
   .
```

This procedure:

- Demonstrates that the prototype database can handle the transaction
- Confirms with a user that the transactions reflect the way the organization conducts its business (a check on the requirements analysis)
- Serves as guide for the actual data manipulation statements in the production application

You should create a prototype (or model) of each transaction to verify each transaction map. Then you can consolidate the transaction maps to discover the most heavily used transaction paths in the logical database.

2.6 Archiving Information

Archiving is essential for maintaining a reasonable size for the database and for saving needed information. If you never archived rows in a database similar to the sample personnel database, the database would eventually outstrip the capacity of the existing disks. It is better to develop an archiving strategy in the design phase than to wait for the day when your database runs out of space.

Different kinds of database tables grow at different rates. Some tables retain a fairly constant number (or volume) of rows, while other tables grow without limit. You may begin to notice that database rows seem to fall into two categories:

- Reference rows
- Event rows

Reference rows are rows in tables that may become fairly stable in number over time and represent information that exists as a reference through time. The EMPLOYEES and DEPARTMENTS tables are examples of tables that contain reference rows. The number of EMPLOYEES rows added in any given month is fairly small and constant (assuming that the company is not experiencing rapid growth). **Event rows** like those in the JOB_HISTORY table, by contrast, are highly volatile, representing a shorter period in the life of some information that may relate to a reference row. The main goal of archiving is to preserve information about old events by removing them from the active database. But if you remove only the event rows, you may lose the reference information that provides a context for these events.

While developing a strategy for archiving rows, you incur additional transactions on the database. Take each archiving transaction and trace its path through the logical database model. When you understand the behavior of all transactions in the database, you can make further tradeoffs in their physical design to improve their performance.

The archiving strategy for the sample personnel database completes the information necessary to calculate the volume of its tables.

2.7 Developing a Volume Table

To further predict the behavior of your design, develop a volume table that allows you to determine the size of your database. By convention, a volume table indicates the state of a database at rest: how many rows or instances of each table occur during any given time period. Much of this information comes from the user requirements. But you also need the archiving strategy to understand how long volatile event rows will stay in the system.

Requirements analysis assumes that our sample company will double its size in 5 years. Assume a current size of 600 employees, yielding 1200 employee rows at some peak volume.

The largest volume of event rows is in the JOB_HISTORY table as shown in Table 2–4. The archiving strategy for the table is to retain employees' rows in the JOB_HISTORY table for one year and then archive them to magnetic tape.

Table 2–4 Volume Table for the Personnel Database

Table	Volume
EMPLOYEES	1200
WORK_STATUS	1200
RESUMES	1200
DEGREES	1800
JOB_HISTORY	3600
SALARY_HISTORY	1200
JOBS	150
DEPARTMENTS	10
COLLEGES	250
CANDIDATES	500

Now that a logical prototype of the database has been designed to accommodate the present needs of end users and to anticipate the future needs of archival storage and growth, you are ready to implement the physical design.

3

Defining a Database

After you design the logical data model, you can create the physical database. That is, you define the database and its elements. This chapter describes how to define a database and how to use the repository when creating a database. This chapter also:

- Provides an overview of database definition
- Summarizes the database definitions you can create with Oracle Rdb
- Explains the options for executing these definitions
- Explains how you can use the repository when you create a database
- Guides you through the process of using SQL statements to define a simple multifile database, including database elements such as domains, tables, triggers, indexes, and views

This chapter provides only introductory information about storage characteristics for multifile database. For detailed information about storage areas and how to assign table data, snapshot files, and indexes to different storage areas, see Chapter 4.

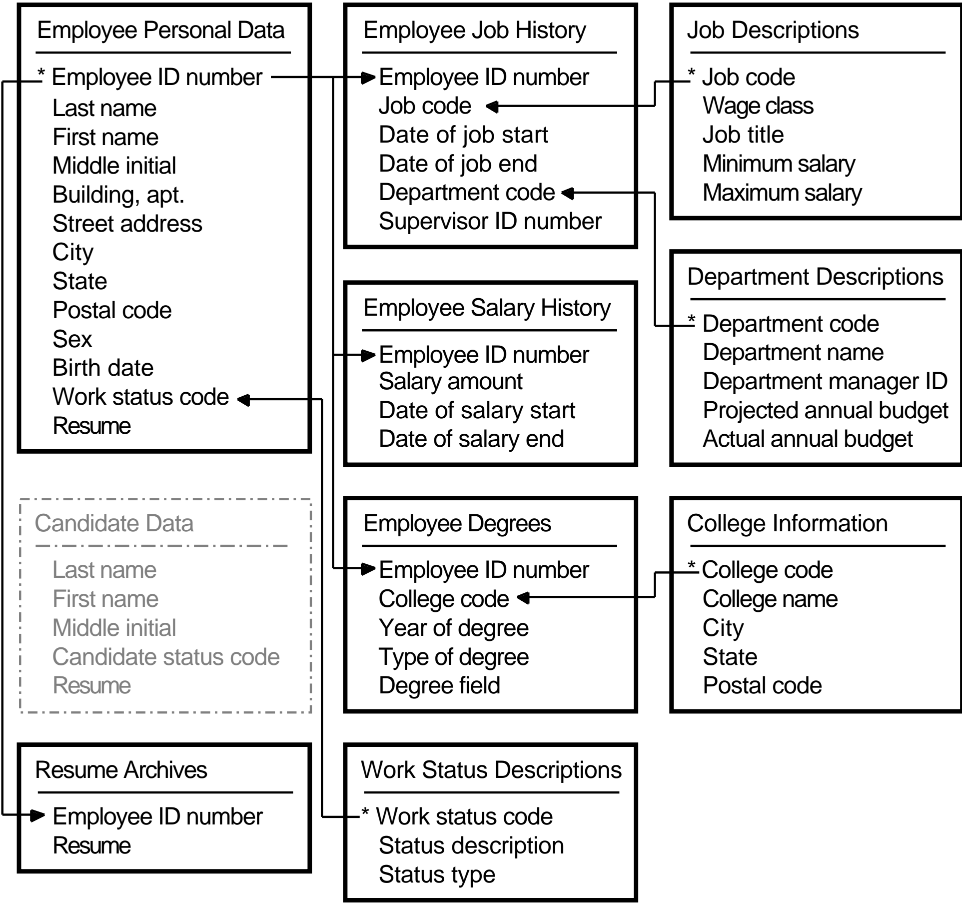
3.1 Overview of Database Definition

A logical data model of a relational database specifies sets of data elements. In SQL, these elements are called tables, columns, and constraints. **Tables** are made up of columns and rows. **Columns** are the vertical dimensions of a table. Columns have a name and a data type, and all values in a column have that same data type. **Constraints** check the validity of table and column values.

The final form of your logical data model should include a separate table for each information category that you identify. See Chapter 2 for further information on logical database design.

Figure 3–1 illustrates the logical data model. The logical definitions shown in this figure are the same for the sample single-file personnel database and the sample multifile mf_personnel database referred to in this chapter.

Figure 3–1 Logical Model of the Sample Databases



After you understand the logical design of the tables and columns, you can proceed with the physical definition. All of the definitions you create for the database make up the physical database definition.

3.2 Summary of Database Elements

A **database** consists of physical data storage characteristics, such as root file and storage area specifications; metadata definitions, such as tables, views, and domains; and user data.

With SQL, you define the following elements for a database:

- The database itself, which defines characteristics that apply to the database as a whole
- Each table in the database, including columns for the table, characteristics of each column (including whether it references a column in another table), and constraints that apply to column values or the entire table

In addition, you can define the following elements:

- **Domains**, which specify a set of values for columns by associating a data type with a domain name
- **Views**, which combine columns from one or more tables, but do not store any data
- **Indexes**, which are structures based on one or more columns to provide direct access to table rows
- **Triggers**, which cause one or more actions to occur when a specified operation (using an INSERT, DELETE, or UPDATE statement) is performed
- A **collating sequence**, which determines both the method for sorting rows when columns are used as sort keys and the behavior of operators that compare two columns or a column with a literal value

  Collating sequences are useful when the data in the database is not in English or when the user's primary language is not English. You define a collating sequence to be used for all columns in the database with the COLLATING SEQUENCE IS clause of the CREATE DATABASE statement.
- **Storage areas**, which specify files to which you may assign table rows or indexes, or both
- **Storage maps**, which determine the arrangement of rows, columns, index nodes, and snapshot areas within or among storage areas
- **Stored routines**, which are SQL language procedures and functions that are stored within the database

You define a stored procedure or function with the CREATE MODULE statement. For more information about creating stored procedures and functions, see the *Oracle Rdb7 Guide to SQL Programming* and the *Oracle Rdb7 SQL Reference Manual*.

- **External routines**, which are functions or procedures that are external to the database and which SQL executes using an SQL statement

  External functions and procedures extend the capability of SQL statements to a nearly infinite variety of programs.

  You can define an external routine and store the definition in the database with the Create Routine statement. For more information about external routines, see Section 3.13.

- **Query outlines**, which are overall plans for how queries can be implemented

  For information about query outlines, see the *Oracle Rdb7 Guide to Database Performance and Tuning*.

When you define a multischema database, in addition to the elements listed previously, you define the following elements:

- **Schemas**, which consist of metadata definitions such as domains, tables, views, constraints, collating sequences, indexes, storage maps, triggers, and the privileges for each of these. Each schema can contain one or more or these elements.
- **Catalogs**, which organize groups of schemas within one database.

Chapter 5 explains how to create a multischema database.

If you do not use the multischema option, your database contains a single schema and no catalogs.

You can define database protection for these elements. Defining database protection is discussed separately in Chapter 9.

3.3 Options for Executing Statements That Define a Database

This chapter shows how to define a database in the interactive SQL environment by typing definition statements at the SQL prompt (SQL>). As alternatives, you can use the following options:

- Use an editor to create a script (command procedure) with the .sql file type. The script can contain all SQL definition statements required to create the database. This method is efficient if you are familiar with the SQL syntax.

- Use the EDIT statement editor in the SQL interactive environment. If you use this method, you can still 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 and then exit the editor to execute a corrected statement. This method is useful if you are unfamiliar with the syntax of the statements.
- Embed the SQL definition statements in a program.

If you use a script to define the database, you execute the script by typing an at sign (@) followed by the file name of the script. For example, to execute a script named create_db.sql in your working directory, you would type:

```
SQL> @create_db.sql
```

It is particularly useful to create database definitions using scripts and to keep these scripts for future use. For example, views cannot be changed. If, in the future, you want to make certain kinds of changes to tables on which views are based, you must first delete the views. You can edit, if necessary, and then execute scripts to create the views again after changing the tables. Similar scripts are often required for other definitions (indexes, constraints, storage maps) that are dependent on a table definition you want to change.

3.4 Using the Repository When You Define a Database

OpenVMS VAX OpenVMS Alpha

You can specify repository access in one of two directions when you define a database and its subordinate elements:

- Into the repository

  You can create a repository node for the database by specifying a PATHNAME clause in a CREATE DATABASE statement. In this case, definitions in the CREATE DATABASE statement are stored both in a database file and in the repository. If you include the DICTIONARY IS REQUIRED clause in your CREATE DATABASE statement, you ensure that any subsequent changes to the database are made in the repository as well.

  If you detach from the database and later want to define, change, or delete definitions in the database, you enter an ATTACH statement that includes a PATHNAME clause with the name of the repository node established by the CREATE DATABASE statement.

- From the repository

You can create definitions for domains and tables based on field and record definitions already stored in the repository. (You can define a table or domain using the FROM path-name clause only if the definition in the repository was originally created using the Common Dictionary Operator (CDO) utility.) In this case, you specify in the CREATE TABLE or CREATE DOMAIN statement a FROM path-name clause to identify the path name of the existing repository definition.

SQL does not require that you use the repository; however, storing database definitions in the repository provides a central source of definitions and allows you to use other information management products with your database. For example, when both the database and subordinate element definitions are in the repository, application programmers can easily include them in host language programs to define program variables that match columns, or to define new databases.

See Chapter 10 for more information on creating and using databases with the repository. ♦

3.5 Creating the Database and Specifying Its Characteristics

If you are creating a corporate database rather than a database for private use, you should probably create all database elements using an account specifically set up for database administration. Chapter 9 explains how to protect the elements in the database.

When you create a database, you first establish characteristics that apply throughout the database and allocate resources for the database. If the repository is installed on your system, you can create a database using definitions from the repository, as explained in Section 3.4.

The CREATE DATABASE statement shown in Example 3–1 creates database files, specifies their names, and determines the physical characteristics for the mf_personnel_test database.

Example 3–1 Creating the Database

```
$ SQL
SQL>  CREATE DATABASE FILENAME mf_personnel_test
cont>        ALIAS MF_PERS
cont>        RESERVE 6 JOURNALS
cont>        RESERVE 15 STORAGE AREAS
cont>        DEFAULT STORAGE AREA default_area
cont>        SYSTEM INDEX COMPRESSION IS ENABLED
cont>     CREATE STORAGE AREA default_area FILENAME default_area
cont>     CREATE STORAGE AREA RDB$SYSTEM FILENAME pers_system_area;
```

The CREATE DATABASE statement shown in Example 3–1 explicitly performs the following operations:

Digital UNIX

- On Digital UNIX, creates a database directory named mf_personnel_test.rdb

  This directory contains the database root (.rdb) file. In this example, it also contains the system storage area file, the default storage area file, and the snapshot files for those storage areas. ♦

- Creates a database root file

  Digital UNIX

  On Digital UNIX, Oracle Rdb creates the database root file, named rdb_system.rdb, in the database directory. ♦

  OpenVMS VAX OpenVMS Alpha

  On OpenVMS, Oracle Rdb creates a database root file, named mf_personnel_test.rdb, in the process default directory. ♦

  The database root file (file type .rdb) contains pointers to storage area files (file type .rda).

- Defines MF_PERS as the alias for mf_personnel_test, which is necessary if you need to access more than one database at a time (for more information about aliases, see Section 3.5.3 and the *Oracle Rdb7 Introduction to SQL*)
- Reserves 6 slots in the database root file for pointers to after-image journal (.aij) files

  By storing all database transaction activity in a common file, .aij files provide a method to roll forward all transactions since the last backup operation. When you reserve slots for journal files, you make it possible to add .aij files while the database is on line. Note that reserving slots does not enable after-image journaling. See Section 3.5.4 for more information about reserving slots for .aij files.

- Reserves 15 slots in the database root file for pointers to storage areas

When you reserve slots for storage areas, you make it possible to add storage areas while the database is on line. See Section 3.5.5 for more information about reserving storage areas.

- Specifies that the default storage area is default_area

  The default storage area stores data from tables and indexes that are not mapped to specific storage areas.

  For information about specifying default storage areas, see Section 3.5.7.

- Specifies that Oracle Rdb compress the system indexes

  See Section 3.5.9 for more information about system index compression.

- Creates the default storage area

  OpenVMS VAX OpenVMS Alpha

  On OpenVMS, Oracle Rdb creates the file, named default_area.rda, in the process default directory. ♦

  Digital UNIX

  On Digital UNIX, Oracle Rdb creates the file, named default_area.rda, in the database directory. ♦

  For information about specifying and creating default storage areas, see Section 3.5.7.

- Creates the RDB$SYSTEM storage area file

  OpenVMS VAX OpenVMS Alpha

  On OpenVMS, Oracle Rdb creates the file, named pers_system_area.rda, in the process default directory. ♦

  Digital UNIX

  On Digital UNIX, Oracle Rdb creates the file, named pers_system_area.rda, in the database directory. ♦

  The RDB$SYSTEM area contains database definitions which are stored as special-purpose tables. These special-purpose tables are called **system tables**.

  If the CREATE DATABASE statement in Example 3–1 did not include the DEFAULT STORAGE AREA clause, the RDB$SYSTEM storage area would include data from tables and indexes in addition to the system tables.

  For more information about storage areas, see Section 3.5.6, Section 3.5.7, Section 3.5.8, Section 4.2, and Section 7.6.

In addition to these explicit operations, the CREATE DATABASE statement shown in Example 3–1 implicitly performs the following operations:

- Creates snapshot files named pers_system_area.snp and default_area.snp

  On OpenVMS, the files are created in the process default directory; on Digital UNIX, they are created in the database directory. The snapshot file (file type .snp) is a work file that improves database performance when retrieval and update transactions simultaneously access the database.

- Uses the default physical storage parameters for the .rdb, .rda, and .snp files
- Specifies that users are not required to update the repository when creating, changing, or deleting definitions from the database
- Specifies that the database contains only one schema and no catalogs
- Uses the default of 50 for the limit for the number of simultaneous database users
- Specifies that database key (dbkey) values are valid only until the end of a transaction
- Defines database protection by creating a default access privilege set for the database itself

  If you use a CREATE DATABASE statement that includes table and view definitions, Oracle Rdb creates a default access privilege set for each table and view. If you have enabled the multischema option, you can define protection for each schema. Chapter 9 explains the default access privilege sets and how you can change them.
- Uses the default DEC_MCS character set as the database default, national, and identifier character sets
- Attaches to mf_personnel_test as the default database

3.5.1 Specifying a Database with Subordinate Elements

In most of this chapter, the CREATE DATABASE statement defines a database without subordinate element definitions. (Chapter sections following this one illustrate using independent statements to define elements such as domains, tables, and views.)

You can create a database using a CREATE DATABASE statement with subordinate elements. That is, you can create storage areas, storage maps, catalogs, schemas, domains, tables, indexes, and views using one CREATE DATABASE statement. In this case, the CREATE DATABASE statement contains only one semicolon (;), which is at the end of the statement. Definitions for the subordinate elements do not end with a semicolon when contained in a CREATE DATABASE statement. Example 3–2 shows how to create a database using a CREATE DATABASE statement with subordinate elements.

Example 3–2 Creating a Database and Specifying Subordinate Elements

```
SQL>  CREATE DATABASE FILENAME mf_personnel_test1
cont>        CREATE STORAGE AREA RDB$SYSTEM FILENAME pers1_system_area
cont> -- Create domains.
cont> CREATE DOMAIN JOB_CODE_DOM CHAR(4)
cont> CREATE DOMAIN WAGE_CLASS_DOM CHAR(1)
cont> CREATE DOMAIN JOB_TITLE_DOM CHAR(20)
cont>
cont> -- Create a table.
cont>   CREATE TABLE JOBS
cont>        (
cont>        JOB_CODE         JOB_CODE_DOM,
cont>        WAGE_CLASS       WAGE_CLASS_DOM,
cont>        JOB_TITLE        JOB_TITLE_DOM
cont>        )
cont>
cont> -- End CREATE DATABASE statement.
cont>    ;
```

The execution of a CREATE DATABASE statement with subordinate definitions starts a read/write transaction to write definitions to the newly created database files. If all definitions execute successfully, the CREATE DATABASE statement automatically commits the transaction. Failure of any part of the CREATE DATABASE statement automatically rolls back any element definitions that have been created prior to the error and deletes any repository node or database files that the statement may have created. Therefore, if you try to execute a COMMIT or ROLLBACK statement immediately following a CREATE DATABASE statement, Oracle Rdb returns an error to tell you that there is no active transaction.

In addition, if you try to execute a DROP DATABASE or DROP PATHNAME statement in response to failure of a CREATE DATABASE statement, you will encounter an error to tell you that the path name or file name does not exist.

3.5.2 Creating Databases Using Multiple Character Sets

Oracle Rdb lets you create databases and database elements using character sets in addition to the DEC Multinational character set. This feature provides support for the Multivendor Integration Architecture (MIA). Oracle Rdb provides support for the following:

- Several character sets in addition to the default DEC Multinational character set (specified in syntax as DEC_MCS.) For the list of supported character sets, see the *Oracle Rdb7 SQL Reference Manual*.
- Using multiple character sets in one database.

- Specifying character sets for database objects, identifiers, literals, and character data type parameters.
- Specifying character lengths and offsets in characters, rather than octets. For more information, see Section 3.8.
- Using the SET DIALECT statement to set the character set. For more information, see the *Oracle Rdb7 Guide to SQL Programming* and the *Oracle Rdb7 SQL Reference Manual*.

When you create a database, you can specify the database default, identifier, and national character sets, as shown in Example 3–3. See the *Oracle Rdb7 SQL Reference Manual* for a description of these character sets.

Example 3–3 Creating a Database Using Multiple Character Sets

```
SQL> -- Before you create a database using the character set clauses,
SQL> -- you must set the dialect to MIA or SQL92.
SQL> --
SQL> SET DIALECT 'SQL92';
SQL> --
SQL> -- Create a database with DEC_KANJI as the database default and
SQL> -- identifier character set and KANJI as the national character set.
SQL> --
SQL> CREATE DATABASE FILENAME mia_char_set
cont>   DEFAULT CHARACTER SET DEC_KANJI
cont>   NATIONAL CHARACTER SET KANJI
cont>   IDENTIFIER CHARACTER SET DEC_KANJI;
SQL> SHOW DATABASE RDB$DBHANDLE
Default alias:
    Oracle Rdb database in file mia_char_set
        Multischema mode is disabled
        Default character set is DEC_KANJI
        National character set is KANJI
        Identifier character set is DEC_KANJI
        Number of users:              50
   .
   .
   .
```

If you do not specify the database default, national, or identifier character sets, SQL uses the character sets specified by the session if the dialect has been set to SQL92 or MIA. If the dialect has not been set to SQL92 or MIA, SQL uses DEC_MCS for the database default, national, and identifier character sets. Note that, even if you choose DEC_MCS for the database default, national, and identifier character sets, you may store strings qualified by any character set within the database. The *Oracle Rdb7 Guide to SQL Programming* describes how to specify the character set for an SQL session.

For more information about specifying the character sets for a database, see the list of supported character sets and the CREATE DATABASE statement in the *Oracle Rdb7 SQL Reference Manual*. For information about specifying character sets for domains, see Section 3.10.2.1.

3.5.3 Specifying an Alias

When you create a database, you can specify an alias using the ALIAS clause in the CREATE DATABASE statement. An **alias** is a name that identifies the database during an attach. You can think of an alias as a database handle. The ALIAS clause is optional.

If you are creating a database when you have already declared another database as your default, you must specify an alias other than the default alias, RDB$DBHANDLE. In this case, statements that follow the CREATE DATABASE statement and either define or refer to already defined elements in the newly created database must include the alias as a qualifier before element names. For example, suppose you specify ALIAS PERS in a CREATE DATABASE statement that executes successfully. If a subsequent statement creates or refers to a table named EMPLOYEES in the newly created database, you must refer to the table as PERS.EMPLOYEES.

However, note that regardless of what you specify for an alias, you never need to qualify the names of any elements that you define within the CREATE DATABASE statement itself.

For a discussion of using aliases when you are creating a multischema database, see Section 5.6. For a more complete discussion and an example of using aliases, see the *Oracle Rdb7 Introduction to SQL*.

3.5.4 Reserving Slots for After-Image Journal Files

When you create a database, you should consider whether you will want to use a single extensible after-image journal (.aij) file or multiple fixed-size .aij files (circular .aij files). Oracle Rdb manages the multiple .aij files as a single seamless .aij file. As one .aij file fills, Oracle Rdb automatically switches to the next available .aij file with users being unaware that any change has taken place.

Although you cannot create .aij files when you create a database, if you choose to use multiple fixed-size .aij files, you can reserve slots in the database root file for pointers to .aij files.

Consider the following:

- A single extensible .aij file

This option lets you specify the number of blocks by which the .aij file should be extended when it is full. However, this option has the following drawbacks:

  – Because you cannot limit the number of times the .aij file extends, you can run out of disk space on the .aij device.
  – While Oracle Rdb is extending the .aij file, database activity may be interrupted.
  – The RMU Backup After_Journal command stalls if it encounters an active .aij file.

- Multiple .aij files of a fixed size

  This option lets you create any number of .aij files. Oracle Rdb writes to only one .aij file at a time. When the current .aij file is full, Oracle Rdb automatically switches to the next available .aij file. This option provides the following benefits:

  – Because each .aij file is a fixed size, you are less likely to run out of disk space. You can specify a different allocation size for each .aij file.
  – If you reserve enough slots for .aij files, you can add .aij files while the database is on line without interrupting database activity.
  – Because you can back up the .aij files that are not currently in use, back up operations are faster. In addition, you can take advantage of automated backup options. For more information about the backup options you can use with multiple .aij files, see the *Oracle Rdb7 Guide to Database Maintenance*.

You use the RESERVE JOURNAL clause to reserve slots for .aij files. Because after-image journaling provides a way to recover from failure of the hardware or software, each journal file should be on a separate disk and should not be on the same disk as other database files. Therefore, the maximum number of slots you should reserve equals the number of disk drives available minus the number of disk drives used for database files. To calculate the number of slots to reserve, allocate one slot for the root file and one for each storage area and add a number for future growth.

When you create a database and do not specify the RESERVE JOURNAL clause, Oracle Rdb reserves one slot.

Because reserving slots takes few resources, it is better to reserve too many slots than to reserve too few. If you have to reserve additional slots, the database must be off line; that is, you must have exclusive access to the database.

For more information about creating .aij files and enabling journaling, see Section 7.4.1 and Section 7.4.2.

3.5.5 Reserving Slots for Storage Areas

When you create a multifile database, you should consider how many storage areas your database may need in the future and reserve slots for at least that number of storage areas. When you reserve slots, you reserve slots in the database root file for pointers to storage areas. If you reserve a sufficient number of slots for storage area files, you can add storage areas while the database is on line without interrupting database activity.

You use the RESERVE STORAGE AREAS clause of the CREATE DATABASE statement to reserve slots for storage areas, as shown in Example 3–1. If you do not specify a RESERVE STORAGE AREAS clause, Oracle Rdb does not reserve any slots.

3.5.6 Specifying Storage Areas for Multifile Databases

When you create a multifile database, you can define more than one storage area and you can specify how tables and indexes are stored in the different areas.

The mf_personnel_test database in this chapter is defined as a multifile database with two storage areas, the RDB$SYSTEM area and the default storage area. The statement illustrates a fairly simple multifile database. Usually, multifile databases include more than one or two storage areas, along with storage maps to specify how tables and indexes are stored in the storage areas.

Because the number and identity of database area files is a characteristic of the database as a whole, you must create storage areas by using the CREATE STORAGE AREA clause in a CREATE DATABASE or IMPORT statement or the ADD STORAGE AREA clause in an ALTER DATABASE statement.

You use CREATE STORAGE MAP statements to assign a table to a particular storage area, partition a table across multiple storage areas, or cluster records that are likely to be accessed together so that one input/output operation retrieves all those records. The STORE clause in a CREATE INDEX statement gives you similar storage control for index records. You can adjust storage area assignments using ALTER STORAGE MAP and ALTER INDEX statements.

3.5.7 Creating a Default Storage Area

To keep user data separate from the system data, such as the system tables, use the DEFAULT STORAGE AREA clause of the CREATE DATABASE statement. This clause specifies that all user data and indexes that are not mapped explicitly to a storage area are stored in the default storage area.

In addition to user data and indexes, Oracle Rdb stores the following system tables in the default storage area:

- RDB$INTERRELATIONS
- RDB$MODULES
- RDB$ROUTINES
- RDB$PARAMETERS
- RDB$QUERY_OUTLINES
- Optional system tables, such as for multischema databases and the workload collection tables

Although you can specify either MIXED or UNIFORM page format for the default storage area, Oracle Rdb recommends using UNIFORM page format to improve performance of sequential retrieval.

Because the DEFAULT STORAGE AREA clause must refer to an existing storage area, you must create the storage area using the CREATE STORAGE AREA clause in the same CREATE DATABASE statement as the DEFAULT STORAGE AREA clause. You cannot drop or change the default storage area. However, you can include the DEFAULT STORAGE AREA clause in an IMPORT statement.

3.5.8 Creating Several Storage Areas in Parallel

You can specify whether Oracle Rdb creates all storage areas in parallel (asynchronously), creates a specified number in parallel, or creates areas serially, using the MULTITHREADED AREA ADDITIONS clause of the CREATE DATABASE statement.

Because you can specify how many storage areas are created in parallel, you may save time when you create a database.

However, if you specify a large number of storage areas and many areas share the same device, multithreading may cause excessive disk head movement, which may result in the storage area creation taking longer than if the areas were created serially. In addition, if you specify a large number of storage areas, you may exceed process quotas, resulting in the database creation failing. For example, a database with 200 storage areas requires a file open

quota of at least 400 files because Oracle Rdb creates a snapshot file for every storage area.

If you specify the ALL AREAS option, Oracle Rdb creates all areas in parallel. If you want to limit the number of areas created at one time, you can use the LIMIT TO n AREAS option. The following example shows how to limit the number of storage areas created at one time to 100 areas:

```
SQL> CREATE DATABASE FILENAME pers_test
cont>       MULTITHREADED AREA ADDITIONS
cont>         (LIMIT TO 100 AREAS)
cont>       CREATE STORAGE AREA STOR1 FILENAME stor1
cont>       CREATE STORAGE AREA STOR2 FILENAME stor2
cont>       CREATE STORAGE AREA STOR3 FILENAME stor3
            .
            .
            .
cont>       CREATE STORAGE AREA STOR200 FILENAME stor200;
```

For more information on the topic of storage area definition in a multifile database, see Chapter 4.

3.5.9 Compressing System Indexes

When you create a database, you can specify that Oracle Rdb compress the system indexes. To do so, use the SYSTEM INDEX COMPRESSION clause of the CREATE DATABASE statement.

For system indexes, Oracle Rdb uses run-length compression, which compresses a sequence of space characters from text data types and binary zeros from nontext data types. It compresses any sequences of two or more spaces for text data types or two or more binary zeros for nontext data types.

Compressing system indexes results in reduced storage and improved I/O. Unless your applications often perform data definition concurrently, you should use compressed system indexes.

Oracle Rdb implicitly enables system index compression when you use the METADATA CHANGES ARE DISABLED clause in a CREATE DATABASE statement.

3.5.10 Choosing Among Snapshot File Options

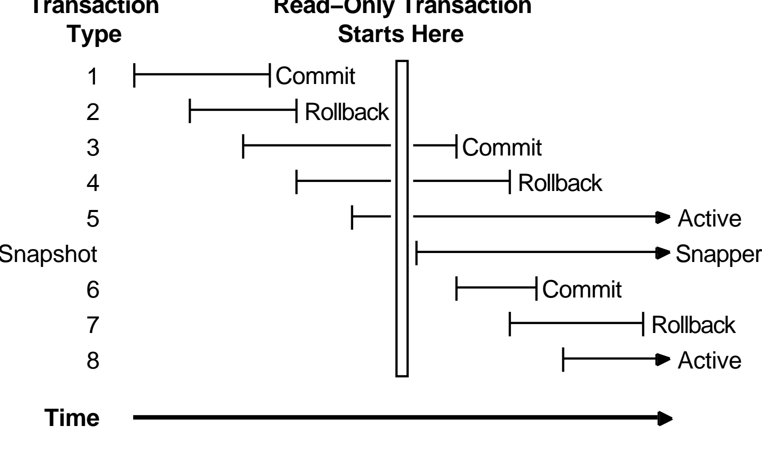
Many transactions only read the database with no intent to update any information. The only requirement for a read-only transaction is that it see a consistent copy of the database. One way to achieve this is to disallow updates to the data being read until after the read-only transaction completes. This can be done with locking, but the cost is a potentially large loss of concurrency. That is, if you lock the database to disallow updates, other users will not be able to update the database at the same time you are reading the data in the database.

Snapshot files allow the read-only transaction to see a consistent view of the database while other transactions update the database. The previous versions of rows are written to the snapshot files by the transactions that update the rows.

The snapshot file does not reflect any updates from update transactions that are still active (uncommitted), nor does it see any updates to the database that start after the snapshot transaction begins. Read-only and exclusive update transactions cannot run concurrently.

Figure 3–2 shows a snapshot transaction time line and how various types of transactions are viewed by the snapshot file.

Figure 3–2 Snapshot Transaction Time Line



When you define multiple data storage areas and enable snapshot files for your database, each storage area has an associated snapshot file.

Snapshot files are enabled by default. Using a CREATE DATABASE or ALTER DATABASE statement, you may choose the following options:

- Enable snapshot files regardless of whether or not a read-only transaction is active (SNAPSHOT ENABLED IMMEDIATE)
- Enable snapshot files only when read-only transactions are active (SNAPSHOT ENABLED DEFERRED)

  The SNAPSHOT ENABLED DEFERRED option reduces unnecessary write operations for read/write transactions that update data, but the SNAPSHOT ENABLED DEFERRED option can make it more difficult for read-only transactions to get started.
- Disable snapshot files for your entire database to conserve disk space (SNAPSHOT DISABLED)

  Read Section 7.4.14 to find out the implications of selecting the SNAPSHOT DISABLED option.

Remember that what you specify for snapshot files applies to all data areas. You cannot, for example, disable snapshot files for some areas and enable snapshot files for other areas. However, you can make adjustments to the physical characteristics of a snapshot file on a per storage area basis.

Note

In this section, the phrase "update transactions" refers to all update transactions except those labeled exclusive update.

The major benefit of the snapshot mechanism is the increased concurrency between read-only transactions and update transactions. With snapshot files, readers and writers to the database can access the same data and still not conflict with each other. There may be some contention between them, but that contention is minimal and transitory. The snapshot mechanism assures that they never interfere with each other.

Snapshot transactions have the following traits:

- A read-only transaction always sees a consistent database.
- A database user can start a read-only transaction at any time.
- A read-only transaction aborts if it cannot successfully access an area. This can happen if another user reserved the area for exclusive update.

- An update transaction (except exclusive update) is not obstructed because of a read-only transaction. Locks held by the read-only transaction are transitory; a read-only transaction holds a lock on a record only long enough to get the record.
- A read-only transaction is never involved in a deadlock.

For more information about using snapshot files, see Section 7.4.14.

3.5.11 Allocating Disk Space and Memory

SQL provides a number of CREATE DATABASE options that determine how disk space and memory are allocated. The options you can specify include the following:

- How much disk space is initially allocated for database files (PAGE SIZE IS, ALLOCATION IS, SNAPSHOT ALLOCATION IS)
- How much disk space is added to database files each time they need to be extended (EXTENT IS, MINIMUM, MAXIMUM, PERCENT GROWTH, and SNAPSHOT EXTENT IS)
- How many pages at a time are held in memory (BUFFER SIZE IS)
- Whether pages are written to disk or held in memory (PAGE TRANSFER IS)
- Whether or not frequently used rows are held in memory (ROW CACHE IS ENABLED/DISABLED)

  If you specify ROW CACHE IS ENABLED, frequently used rows remain in memory, even when the associated page has been flushed back to disk, resulting in reduced database page reads and writes. By default, Oracle Rdb gives you one cache slot. To add more, you must first reserve cache slots with the RESERVE CACHE SLOTS clause of the CREATE or ALTER DATABASE statement. Then, create cache areas with the CREATE CACHE or ADD CACHE clauses.

  For more information, see the *Oracle Rdb7 Guide to Database Performance and Tuning*.
- The maximum number of users that can simultaneously access the database (NUMBER OF USERS IS)

  (Oracle Rdb considers each attach to the database a user.)
- How many buffers are available to each user accessing the database (NUMBER OF BUFFERS IS)
- Whether to use local buffers or global buffers

Local buffers are the default. If you use global buffers (GLOBAL BUFFERS ARE ENABLED), you can specify the following:

- How many buffers are available to all the users accessing the database from a single node (NUMBER IS number-of-global-buffers)
- The maximum number of global buffers that can be allocated to any one user (USER LIMIT IS)

When you use **local buffers**, Oracle Rdb maintains a buffer pool for each user. If more than one user uses the same page, each user must read the page into his or her own buffer pool.

When you use **global buffers**, Oracle Rdb maintains one buffer pool on each node for each database. If more than one user uses the same page, only one user must read the page into the global buffer pool, and other users can read the page from the buffer pool. Global buffers improve performance because input and output are reduced and memory is better utilized.

For information on how to determine whether to use local or global buffers and how to determine the buffer size, number of buffers, and user limit, see the *Oracle Rdb7 Guide to Database Performance and Tuning*.

The CREATE STORAGE AREA clause also has subordinate clauses that let you specify allocation, page size, extent, snapshot allocation, snapshot extent, minimum, maximum, and percent growth. In multifile databases, it is important to examine the parameters set by these clauses in the context of each storage area. For example, the ALLOCATION clause is critical for areas where records are distributed according to hashed index values. In areas where you cluster related rows on a page, set the PAGE SIZE clause large enough to accommodate all the rows you expect to cluster on a page. Even if you do not cluster multiple rows on a page, set the PAGE SIZE so that an entire row can fit on one page.

Therefore, for file allocation and page size values, you rarely rely on default values set by the database system or set your own defaults in the CREATE DATABASE statement. Instead, you need to explicitly specify file allocation and page size values in each CREATE STORAGE AREA clause.

Chapter 4, Chapter 7, and the *Oracle Rdb7 Guide to Database Performance and Tuning* explain how you determine if you should change the defaults for database storage and memory.

3.5.12 Setting Database Key (Dbkey) Scope

If you are writing a program to create a database, load it with data, and then use that data, you may want to retrieve and use database key (dbkey) values for rows. The CREATE DATABASE statement includes a DBKEY SCOPE clause because you may want to specify a dbkey scope option for the database invocation that the CREATE DATABASE statement makes on your behalf. For more information, see the *Oracle Rdb7 SQL Reference Manual*.

3.5.13 Specifying Who Can Open a Database

To specify that a database must be opened explicitly by a privileged user, include the OPEN IS MANUAL clause in the CREATE DATABASE statement. The default is OPEN IS AUTOMATIC, which means that Oracle Rdb automatically opens database files when the first privileged or nonprivileged user attaches to the database.

Use the OPEN IS MANUAL clause for the following reasons:

- To ensure that no one accesses the database while you are performing regular maintenance operations. You must have the SQL DBADM privilege to attach to the database. You must use the RMU Open command to explicitly open the database.
- To control which node in the cluster handles lock requests by opening the database first on that node.

Use the OPEN IS AUTOMATIC clause for the following situations:

- When regular maintenance operations are complete. When you set the database for automatic opening, the first attach operation opens the database.
- When you do not need to control which node handles locking. The first user to attach to the database determines the node on which locks are handled (at least until a cluster transition).

In an environment where users frequently attach and detach from the database, use the RMU Open command to open the database manually and leave it open rather than allow the first attached user process to absorb the additional overhead when it attaches to the database. Leaving a database open does not use additional system overhead.

When you specify OPEN IS AUTOMATIC, you can use the WAIT FOR CLOSE clause to specify the amount of time that Oracle Rdb waits before automatically closing a database.

For more information on the use of the SQL OPEN IS clause or the RMU Open and RMU Close commands, see the *Oracle Rdb7 Guide to Database Maintenance*.

3.5.14 Looking for More Detailed Information About Database Definition

The preceding sections about creating a database showed you how to define a simple database and introduced you to some options you can apply to a database. Chapter 4 provides an overview of your options for creating a database with more complex requirements for user-defined storage. It also provides information about default database characteristics and when it is appropriate to change these. Chapter 7 describes how to change many database and storage area characteristics.

If you are designing a complex database that many users will access, you will not find all the information you need in this manual. Refer to the section about the CREATE DATABASE statement in the *Oracle Rdb7 SQL Reference Manual* for detailed information about all syntax options that you can include in a CREATE DATABASE statement. In addition, consult the *Oracle Rdb7 Guide to Database Performance and Tuning* to find more information about the database options and how to change these to improve performance.

For information about creating a multischema database, see Chapter 5.

The following sections describe how to define database elements such as domains, tables, and columns.

3.6 Naming Database Elements

When you create database elements, you supply names (identifiers) for the elements. You can use one of the following types of names:

- Identifiers

  Identifiers are user-supplied names that can include alphanumeric characters such as letters, underscores (_), and digits. You can use uppercase or lowercase, but SQL converts all identifiers to uppercase.

- Delimited identifiers

  Delimited identifiers are user-supplied names that are enclosed in double quotation marks ("). They can contain alphanumeric characters, special characters, control characters, and spaces. Trailing spaces are not significant. The alphabetic characters can be uppercase or lowercase.

  To use delimited identifiers, you must enable ANSI/ISO quoting. One method is to use the SET QUOTING RULES 'SQL92' statement.

For more information about valid characters for user-supplied names, see the *Oracle Rdb7 SQL Reference Manual*.

3.7 Using Data Types

When you create database elements such as domains or columns in a table, you must specify a data type. Some of the data types that you can specify include the following:

- CHAR

  The CHAR data type is a fixed-length data type. Use it for text, such as names and labels. The CHAR data type is also useful for identification numbers that are not used in calculations, for example, room numbers. The size of the domain or column should be sufficient to hold the longest string of text characters that you anticipate storing in the column. If any string is shorter than the size of the domain or column, Oracle Rdb pads the column with blank characters.

- VARCHAR

  The VARCHAR data type is a text data type that you use when the length of a text string may vary. For example, use it for a column that holds comments about a job candidate's status. Oracle Rdb maintains information about the length of each string stored in a VARCHAR column and passes that information to the application.

- SMALLINT

  The SMALLINT data type (a signed 16-bit word) is a fixed-point numeric data type. The SMALLINT data type can store a range of values from –32768 to 32767.

- INTEGER

  The INTEGER data type (a signed 32-bit longword) is a fixed-point numeric data type. It can store a range of values from –2**31 to (2**31)–1. You can specify a scale factor, which indicates the number of places to the right of the decimal point. To store money values such as $10,229.85, use the INTEGER (2) data type.

- BIGINT

  The BIGINT data type (a signed 64-bit quadword) is a fixed-point numeric data type. The BIGINT data type can store a range of values from –2**63 to (2**63)–1.

- LIST OF BYTE VARYING

The LIST OF BYTE VARYING data type allows you to store large amounts of data in a single column. You can store large amounts of text, long strings of binary input from a data collecting device, or graphics data in a list. Each column of the LIST OF BYTE VARYING data type consists of segmented lists. Each segment of a list can store up to 64K bytes of data. The list itself can grow to an infinite size, limited only by disk capacity.

Oracle Rdb provides three on-disk formats for lists:

- Chained format
- Indexed format
- Single-segment format

For more information about the formats, see the *Oracle Rdb7 SQL Reference Manual*.

- DATE ANSI

  The DATE ANSI data type specifies the year, month, and day. DATE ANSI is the default date format if you use the SET DIALECT 'SQL92' statement or the SET DEFAULT DATE FORMAT 'SQL92'. You can perform arithmetic operations on the columns with this data type and you can extract an individual date-time field from a column.

- TIME

  The TIME data type specifies military time containing the hour, minute, and second. You can perform arithmetic operations on the columns with this data type and you can extract an individual date-time field from a column.

- INTERVAL

  The INTERVAL data type specifies the difference between two absolute dates, times, or timestamps. Intervals can be year-month intervals or day-time intervals. You can perform arithmetic operations on the columns with this data type and you can extract an individual date-time field from a column.

- TIMESTAMP data type

  The TIMESTAMP data type specifies the year, month, and day as well as 24-hour military time containing the hour, minute, and second. You can perform arithmetic operations on the columns with this data type and you can extract an individual date-time field from a column.

- DATE VMS

The DATE VMS data type is a quadword value that specifies a timestamp containing the year to the second. Many OpenVMS languages and utilities use this DATE data type for specifying dates. By default, SQL interprets DATE as DATE VMS. However, you cannot perform date arithmetic on columns of the DATE VMS data type without first converting it to a DATE ANSI, TIME, or TIMESTAMP data type.

For more information about the data types that SQL supports, refer to the *Oracle Rdb7 SQL Reference Manual*.

3.8 Specifying the Length of Characters in Octets or Characters

SQL lets you determine whether you want to specify the length of a character data type using octets or characters. An **octet** is a group of eight bits. Specifying the character length in octets or characters is especially important when you use multi-octet character sets such as Kanji.

You can use the SET CHARACTER LENGTH statement to control whether you specify the length of elements, such as domains and columns, in octets or characters. Example 3–4 illustrates controlling how SQL interprets the length of characters.

Example 3–4 Specifying How SQL Interprets the Length of Characters

```
SQL> ATTACH 'FILENAME mia_char_set';
SQL> -- Set the length to CHARACTERS and create a domain that is 3
SQL> -- characters (6 octets) in length.  Because a character set is not
SQL> -- specified, SQL uses the default character set (DEC_KANJI) for the
SQL> -- domain in this example.
SQL> --
SQL> SET CHARACTER LENGTH 'CHARACTERS';
SQL> CREATE DOMAIN TEST_DOM CHAR(3);
SQL> --
SQL> -- SQL displays the size of the domain in characters
SQL> --
SQL> SHOW DOMAIN TEST_DOM
TEST_DOM                        CHAR(3)
SQL> -- Set the length to OCTETS.  SQL displays the size of the domain in
SQL> -- octets.
SQL> --
SQL> SET CHARACTER LENGTH 'OCTETS';
SQL> SHOW DOMAIN TEST_DOM
TEST_DOM                        CHAR(6)
SQL>
```

If you specify the SET CHARACTER LENGTH 'OCTETS' statement before creating domains or tables, you must make sure that you specify an appropriate size for domains and columns. For example, because the KANJI character set uses two octets for each character, you must specify a character data type length that is a multiple of two, as the following example demonstrates.

```
SQL>  SET CHARACTER LENGTH 'OCTETS';
SQL>  CREATE DOMAIN TEST_DOM1 CHAR(5) CHARACTER SET KANJI;
%SQL-F-CHRUNIBAD, Number of octets is not an integral number of characters
SQL>  CREATE DOMAIN TEST_DOM1 CHAR(6) CHARACTER SET KANJI;
SQL> SHOW DOMAIN TEST_DOM1
TEST_DOM1                       CHAR(6)
         KANJI 3 Characters,  6 Octets
SQL> ROLLBACK;
SQL> EXIT
```

You can also control how character length is specified with the SET DIALECT statement. For information on the number of octets used by each character set and more information on setting the character length, see the *Oracle Rdb7 SQL Reference Manual*.

3.9 Including Comments in Definitions of Elements

You can include comments that explain the purpose of certain database elements by specifying the COMMENT ON statement. You can use the COMMENT ON statement for domains, columns, tables, and indexes.

The quoted string you provide in the COMMENT ON statement is displayed for the specified element when interactive SQL users enter SHOW statements.

For example, you include comments about the domain definition by entering a COMMENT ON DOMAIN statement:

```
SQL> CREATE DOMAIN ID_DOM CHAR(5);
SQL> COMMENT ON DOMAIN ID_DOM IS
cont> 'Standard definition of employee ID';
SQL> SHOW DOMAIN ID_DOM
ID_DOM                          CHAR(5)
 Comment:       Standard definition of employee ID
```

Example 3–6 includes COMMENT ON DOMAIN statements for some domain definitions.

3.10 Creating Domains

After defining characteristics for the database as a whole, you can create domains for the database. A domain is analogous to a user-defined data type in some programming languages. A **domain** specifies a set of values for columns by associating a data type with a domain name.

There are two ways you can create a domain using the CREATE DOMAIN statement:

- You can specify a domain name, data type, and, optionally, other characteristics that can be implemented by more than one column in more than one table.
- You can create a domain by specifying the FROM clause and a repository path name. SQL creates the domain using the definition from this repository field.

3.10.1 Creating Domains Based on Repository Fields

OpenVMS VAX OpenVMS Alpha

You can create domains that are based on repository field definitions. In this case, you include a FROM path-name clause in your CREATE DOMAIN statement, as shown in Example 3–5. For Example 3–5 to work, you must have created the mf_personnel_test database using the PATHNAME clause.

Example 3–5 Creating Domains Using the FROM Path-Name Clause

```
$ REPOSITORY
Welcome to CDO V2.3
The CDD/Repository V5.3 User Interface
Type HELP for help
CDO> !
CDO> ! Create the field using CDO:
CDO> !
CDO> DEFINE FIELD DOMAIN_TEST DATATYPE IS SIGNED QUADWORD.
CDO> EXIT
$ SQL
SQL> -- Attach to the database.
SQL> --
SQL> ATTACH 'ALIAS MF_PERS PATHNAME mf_personnel_test';
SQL> --
SQL> -- Use the FROM path-name clause to create the domain:
```

(continued on next page)

Example 3–5 (Cont.) Creating Domains Using the FROM Path-Name Clause

```
SQL> --
SQL> CREATE DOMAIN FROM DOMAIN_TEST;
SQL> --
SQL> -- Now look at the domain definition:
SQL> --
SQL> SHOW DOMAIN DOMAIN_TEST
DOMAIN_TEST                    QUADWORD
 CDD Pathname:  SYS$COMMON:[CDDPLUS]DEPT32.FIELDMAN.DOMAIN_TEST;1
SQL> COMMIT;                                        ♦
```

3.10.2 Specifying Characteristics of Domains

If you do not use the repository definitions to define domains, you can specify characteristics for domains using the CREATE DOMAIN statement.

You can create a domain to specify a standard storage format and length for a specific kind of name, such as job title, that may be a column in one or more tables. Or, you can create a more generic domain to specify a standard storage format and length for all nonpersonal names such as job titles, department names, company names, project names, and so forth.

Although you do not have to explicitly create domains for a database, Oracle Rdb recommends that you either create all domains for your database or let SQL create all of them for you. Domain characteristics include:

- A name for the domain

  Because domain can be used in more than one table, you should make domain names as general as possible. For example, several columns might contain date values and use the same domain. Give these domains a generic name, such as STANDARD_DATE_DOM.
- A data type
- A character set

  If you do not specify a character set, SQL uses the default character set.
- An optional default value for the domain

  Specify a data value to be stored in a column if the row that is inserted does not include a data value for that column.
- A collating sequence
- Optional formatting clauses that allow you to modify data displays or query characteristics for interactive SQL users.
- An optional constraint that applies to columns based on the domain

Specify a value or a range of values to restrict the data values that can be stored in columns that are based on the domain.

Example 3–6 creates some of the domains in the mf_personnel_test database. (Other domains are created later in this section.)

Example 3–6 Creating Domains

```
SQL> -- This example assumes that you have created the mf_personnel_test
SQL> -- database.  If you have, but it is not your default database now,
SQL> -- enter the ATTACH statement before continuing with this example.
SQL> --
SQL>  ATTACH 'FILENAME mf_personnel_test';
SQL> --
SQL> -- Create the domains.  This example shows how to add comments,
SQL> -- but does not include comments for all domains.
SQL> CREATE DOMAIN ID_DOM CHAR(5);
SQL> COMMENT ON DOMAIN ID_DOM IS
cont> 'Standard definition of employee ID';
SQL> --
SQL> CREATE DOMAIN LAST_NAME_DOM CHAR(14);
SQL> COMMENT ON DOMAIN LAST_NAME_DOM IS
cont> 'Standard definition of last name';
SQL> --
SQL> CREATE DOMAIN FIRST_NAME_DOM CHAR(14);
SQL> COMMENT ON DOMAIN FIRST_NAME_DOM IS
cont> 'Standard definition of first name';
SQL> --
SQL> CREATE DOMAIN ADDRESS_DATA_1_DOM CHAR(25);
SQL> --
SQL> CREATE DOMAIN CITY_DOM CHAR(20);
SQL> --
SQL> CREATE DOMAIN STATE_DOM CHAR(2);
SQL> --
SQL> CREATE DOMAIN POSTAL_CODE_DOM CHAR(5);
SQL> --
SQL> CREATE DOMAIN SEX_DOM CHAR(1);
SQL> --
SQL> CREATE DOMAIN STANDARD_DATE_DOM DATE;
SQL> --
SQL> CREATE DOMAIN DEPARTMENT_CODE_DOM CHAR(4);
SQL> --
```

```
SQL> CREATE DOMAIN JOB_CODE_DOM CHAR(4);
SQL> COMMIT;
```

When you create the table itself, you can base column definitions on the domain definitions. The following example shows that the column definitions in the JOB_HISTORY table are based on the domain definitions:

```
SQL> CREATE TABLE JOB_HISTORY
cont>       (EMPLOYEE_ID      ID_DOM,
cont>        JOB_CODE         JOB_CODE_DOM,
cont>        JOB_START        STANDARD_DATE_DOM,
cont>        JOB_END          STANDARD_DATE_DOM,
cont>        DEPARTMENT_CODE  DEPARTMENT_CODE_DOM );
SQL> ROLLBACK;
```

The following example shows how to specify a variety of data types:

```
SQL> -- Use the SET DIALECT 'SQL92' statement so that the default date
SQL> -- format is DATE ANSI.
SQL> --
SQL>  SET DIALECT 'SQL92';
SQL> --
SQL> CREATE DOMAIN CANDIDATE_STATUS_DOM VARCHAR(255);
SQL> --
SQL> CREATE DOMAIN SALARY_DOM INTEGER
cont>    EDIT STRING IS '$$$$,$$9.99';
SQL> --
SQL>  CREATE DOMAIN YEAR_DOM SMALLINT;
SQL> --
SQL> CREATE DOMAIN RESUME_DOM LIST OF BYTE VARYING;
SQL> --
SQL> CREATE DOMAIN DATE_DOM DATE;
```

Consult the *Oracle Rdb7 SQL Reference Manual* for a complete list of the data types and the characteristics for each data type.

The next sections describe how to specify character sets, collating sequences, optional default values, optional support clauses, and edit strings. For more information about these topics, see the CREATE DOMAIN statement and the chapter on SQL language and syntax elements in the *Oracle Rdb7 SQL Reference Manual*.

3.10.2.1 Specifying Character Sets for Domains

You can specify a character set for a domain by qualifying the data type with a character set or by using the NCHAR or NCHAR VARYING data type, as shown by the following example:

```
SQL> ATTACH 'FILENAME mia_char_set';
SQL> --
SQL> SET CHARACTER LENGTH 'CHARACTERS';
SQL> SET DIALECT 'SQL92';
SQL> --
SQL> -- Specify the character set.
SQL>  CREATE DOMAIN MCS_DOM CHAR(8) CHARACTER SET DEC_MCS;
SQL> --
SQL> -- If you use the NCHAR or NCHAR VARYING data type, SQL uses the
SQL> -- national character set of the database. In this example, KANJI
SQL> -- is the national character set.
SQL> --
SQL> CREATE DOMAIN KANJI_DOM NCHAR(4);
SQL> SHOW DOMAINS
User domains in database with filename mia_char_set
KANJI_DOM                      CHAR(4)
         KANJI 4 Characters,  8 Octets
MCS_DOM                        CHAR(8)
         DEC_MCS 8 Characters,  8 Octets
```

If you do not specify a character set, SQL uses the default character set, as shown in the following example:

```
SQL> CREATE DOMAIN DEC_KANJI_DOM CHAR(8);
SQL> --
SQL> -- If the domain has the same character set as the session's default
SQL> -- character set, SQL does not display the character set.
SQL> --
SQL> SHOW DOMAINS
User domains in database with filename mia_char_set
DEC_KANJI_DOM                  CHAR(8)
KANJI_DOM                      CHAR(4)
         KANJI 4 Characters,  8 Octets
MCS_DOM                        CHAR(8)
         DEC_MCS 8 Characters,  8 Octets
SQL>
```

For more information on specifying character sets for domains, see the *Oracle Rdb7 SQL Reference Manual*.

3.10.2.2 Specifying Default Values for Domains

SQL lets you specify a default value for a domain. The default value clause specifies what data value is stored in columns based on the domain if an insert or update operation on a row does not specify a data value for that column. For example, you can use a default value such as NULL or “Not Applicable” that clearly highlights that no data was inserted into a column based on that domain, or you can use a default value such as “full-time” for work status if a column would usually contain a particular value.

You can use the following as default values:

- Literals

  Literals must be the same data type as the domain.
- NULL keyword
- The SQL built-in functions, including CURRENT_DATE, CURRENT_TIME, CURRENT_TIMESTAMP, and the functions that return the user name, such as CURRENT_USER, SESSION_USER, SYSTEM_USER, and USER

If you do not specify a default value, SQL assigns NULL as the default value. You can also specify a default value for a column in a table, but when you do, it overrides any default value specified for the domain on which the column is based.

Example 3–7 shows how to specify default values for domains.

Example 3–7 Specifying Default Values

```
SQL> ATTACH 'FILENAME mf_personnel_test';
SQL> --
SQL> -- If no middle initial is entered, use NULL.
SQL> --
SQL> CREATE DOMAIN MIDDLE_INITIAL_DOM CHAR(1)
cont>       DEFAULT NULL;
SQL> --
SQL> SHOW DOMAIN MIDDLE_INITIAL_DOM
MIDDLE_INITIAL_DOM              CHAR(1)
 Oracle Rdb default: NULL
SQL> --
```

(continued on next page)

Example 3–7 (Cont.) Specifying Default Values

```
SQL> -- If the street address takes only one line,
SQL> -- use 'None' for the default for the second line.
SQL> --
SQL> CREATE DOMAIN ADDRESS_DATA_2_DOM CHAR(20)
cont>       DEFAULT 'None';
```

If you do not specify a data value for a column that has a default value defined, Oracle Rdb stores the default value in the database. If you do not specify a data value for a column that has no default value defined, Oracle Rdb stores nothing in that column and sets an internal null flag.

One implication of the way in which Oracle Rdb handles default values is that if you add or change the default value for a domain or column, it has no effect on any existing data in the database; that is, rows stored with columns that contain the *old* default value are not changed.

For more information on specifying a default value, see the sections on the CREATE DOMAIN, ALTER DOMAIN, CREATE TABLE, and ALTER TABLE statements in the *Oracle Rdb7 SQL Reference Manual*.

3.10.2.3 Specifying Collating Sequences

A **collating sequence** determines how rows are sorted when the column is used as a sort key and it determines the behavior of operators that compare items of this domain type.

By default, Oracle Rdb uses the ASCII collating sequence for all sorting and Boolean operations; you can override this default by specifying one of the following:

- One of the language-specific collating sequences supplied by the OpenVMS National Character Set (NCS) utility
- A user-defined collating sequence using NCS

To specify a collating sequence, you must first identify one using the CREATE COLLATING SEQUENCE statement. Then, you use the COLLATING SEQUENCE clause of the CREATE DOMAIN statement to specify the collating sequence for the domain, as shown in the following example:

```
SQL> CREATE COLLATING SEQUENCE SP SPANISH FROM NCS$LIBRARY;
SQL> CREATE DOMAIN SP_NAMES_DOM CHAR(20) COLLATING SEQUENCE IS SP;
```

If you want to specify a collating sequence for all columns in the database, use the CREATE DATABASE . . . COLLATING SEQUENCE statement. If you want to specify a collating sequence for only some columns or domains, use the CREATE COLLATING SEQUENCE statement.

3.10.2.4 Specifying SQL Formatting Clauses

If you intend to access the database with interactive SQL, you may wish to specify SQL formatting clauses to enhance how data is displayed. You can use formatting clauses, including the following, to modify the data display:

- EDIT STRING

  In this clause, you specify an edit string to control the display of data values in columns that are based on the domain. You might specify an edit string to include, for example, a dollar sign ($) before values that represent money or to include a period (.) after values that represent initials. You can specify how dates are displayed, as shown in the following example:

  ```
  SQL> CREATE DOMAIN DATES_DOM DATE
  cont>      EDIT STRING IS 'DD-MMM-YYYY';
  ```

- QUERY HEADER

  In this clause, you specify a character string that in data displays replaces the names of columns that are based on the domain. You can use the QUERY HEADER clause to replace a column name with one that is easier to read. In addition, when column data values include fewer display characters than column names, you can specify a column header that requires the same, or less, line space as the data values do. For some tables, this may reduce or eliminate line wrapping in data displays.

You can also specify EDIT STRING and QUERY NAME clauses for the columns you define in table and view definitions. When you specify these clauses in a domain definition, they supply defaults for associated columns in tables, but not for any associated columns in views.

3.10.2.5 Specifying Domain Constraints

To limit which values can be stored in columns based on a domain, you can specify a domain constraint when you create a domain. **Domain constraints** specify that columns based on the domain contain only certain data values or that data values can or cannot be null.

You use the CHECK clause to specify that a value must be within a specified range or that it exactly matches a list of values. When you specify a CHECK clause for a domain, you ensure that all values stored in columns based on the domain are checked consistently.

Example 3–8 shows how to create domain constraints.

Example 3–8 Specifying a Constraint for a Domain

```
SQL> -- The SET DIALECT 'SQL92' statement makes the NOT DEFERRABLE clause
SQL> -- optional and sets the default date format to the ANSI/ISO standard
SQL> -- format.
SQL>  SET DIALECT 'SQL92';
SQL> --
SQL> -- The following domain checks that the values inserted into any columns
SQL> -- based on the STATUS_CODE_DOM domain match the list of values in the
SQL> -- CHECK clause.  Note that the list of values includes the value
SQL> -- 'N', which is the default value for the domain.
SQL> --
SQL> CREATE DOMAIN STATUS_CODE_DOM CHAR(1)
cont>       DEFAULT 'N'
cont>       CHECK (VALUE IN ('0', '1', '2', 'N'))
cont>         NOT DEFERRABLE;
SQL> --
SQL> -- The example explicitly uses the NOT DEFERRABLE clause for clarity.
SQL> --
SQL> -- The following domain ensures that any dates inserted into the database
SQL> -- are later than January 1, 1900:
SQL> --
SQL> CREATE DOMAIN DATE_DOM DATE
cont>       DEFAULT NULL
cont>       CHECK (VALUE > DATE'1900-01-01' OR
cont>              VALUE IS NULL)
cont>           NOT DEFERRABLE;
SQL>
SQL> -- The following example creates a table with one column based on the
SQL> -- domain DATE_DOM:
SQL> CREATE TABLE DOMAIN_TEST
cont>       (DATE_COL   DATE_DOM);
SQL> --
SQL> -- SQL returns an error if you attempt to insert data that does not
SQL> -- conform to the domain constraint:
SQL> --
SQL> INSERT INTO DOMAIN_TEST
cont>       VALUES (DATE'1899-01-01');
%RDB-E-NOT_VALID, validation on field DATE_COL caused operation to fail
```

Keep in mind the following points:

- Domain constraints must be NOT DEFERRABLE. You can explicitly specify the NOT DEFERRABLE clause or use the SET DIALECT 'SQL92' statement, which sets the mode of domain constraints to NOT DEFERRABLE.
- You can specify only one constraint for each domain.

• Domain constraints are not applied to variables and parameters.

3.11 Creating Tables

After you create domains, you can create tables where data is stored. There are two ways you can create a table. You can specify a table name, column names, the data type of each column, and various options in the CREATE TABLE statement. Alternatively, you can specify a FROM clause in the CREATE TABLE statement to define a table from a record definition stored in the repository.

3.11.1 Creating Tables Based on Repository Definitions

You can use the FROM path-name clause of the CREATE TABLE statement to create a table that is based on a record definition in the repository. You can create a table using the FROM path-name clause *only* if the definition in the repository was originally created using CDO.

Example 3–9 shows how to define fields and records in the repository using CDO.

Example 3–9 Defining Fields and Records with Oracle CDD/Repository

```
$ ! First, use CDO to define the fields and record in the
$ ! repository:
$ !
$ REPOSITORY
Welcome to CDO V2.3
The CDD/Repository V5.3 User Interface
Type HELP for help
CDO> DEFINE FIELD ONE DATATYPE IS TEXT 4.
CDO> DEFINE FIELD TWO DATATYPE IS TEXT 4.
CDO> !
CDO> ! Look at the fields using the SHOW FIELDS statement.  If you
CDO> ! defined the field DOMAIN_TEST when reading the section on
CDO> ! domains, DOMAIN_TEST also appears in your display:
CDO> !
CDO> SHOW FIELDS
Definition of field DOMAIN_TEST
|   Datatype                signed quadword

Definition of field ONE
|   Datatype                text size is 4 characters

Definition of field TWO
|   Datatype                text size is 4 characters
```

(continued on next page)

Example 3–9 (Cont.) Defining Fields and Records with Oracle CDD/Repository

```
CDO> !
CDO> ! Define a record called NUMBERS using the fields ONE and TWO
CDO> ! that you just defined:
CDO> !
CDO> DEFINE RECORD NUMBERS.
cont> ONE.
cont> TWO.
cont> END NUMBERS RECORD.
CDO> EXIT
```

Note that you cannot specify column constraints and other optional clauses for columns when you create a table from a record definition stored in the repository. However, you can create table constraints. For more information, see Section 10.4.

Example 3–10 shows you how to use the FROM path-name clause of the CREATE TABLE statement to create a table that is based on a record definition in the repository.

Example 3–10 Creating a Table Using the FROM Path-Name Clause

```
$ SQL
SQL> -- Attach to the database by path name:
SQL> --
SQL> ATTACH 'PATHNAME mf_personnel_test';
SQL> --
SQL> -- Now, use the FROM path-name clause to create a table
SQL> -- based on the record NUMBERS:
SQL> --
SQL> CREATE TABLE FROM SYS$COMMON:[CDDPLUS]FIELDMAN.NUMBERS;
SQL> --
SQL> -- The SHOW TABLE statement shows the columns in the newly
SQL> -- created table NUMBERS.  Notice that SQL automatically uses
SQL> -- table, domain, and column names that are the same as the
SQL> -- field and record names that you specified with CDO:
```

(continued on next page)

Example 3–10 (Cont.) Creating a Table Using the FROM Path-Name Clause

```
SQL> --
SQL> SHOW TABLE (COLUMNS) NUMBERS
Information for table NUMBERS

Columns for table NUMBERS:
Column Name                     Data Type        Domain
-----------                     ---------        ------
ONE                             CHAR(4)          ONE
TWO                             CHAR(4)          TWO

SQL> -- Use ROLLBACK so that the table does not become part of the database.
SQL> ROLLBACK;
```

To make the definitions of the tables part of the database, you could have entered a COMMIT statement instead of a ROLLBACK statement.

Suppose you attach to a database by file name and sometime later in your session create a table whose definition should be in the repository. You can use the SHOW ALIAS statement to find out how you attached to the database. If the database is identified by file name in the resulting display, you attached to the database by file name. In this case, subsequent CREATE statements do not store definitions in the repository. After rolling back the table definition, you can attach to the database by path name, enter the table definition again, and then commit the definition. Alternatively, you can use the INTEGRATE statement, which is described in Chapter 10. ♦

3.11.2 Specifying Elements of Tables

If you do not want to define a table using the repository definitions, you can define the elements of a table using the CREATE TABLE statement.

A table definition includes:

- A name for the table
- A name for each column in the table
- The definition of each column in the table

  The definition can specify many characteristics, such as the data type or constraints, or it can specify that the column is a computed value and describe how to evaluate the value.

  If the column is a computed value, you specify a COMPUTED BY clause. Otherwise, you can specify the following characteristics:

  – The data type for each column in the table

You can explicitly specify the data type, length, and (for some numeric columns) scale factor of the data stored in each column, or you can specify a domain upon which to base the column.

- The character set for each column in the table

  You can explicitly specify the character set for each column of the character data type. If you specify a domain upon which to base the column, SQL uses the character set of the domain. If you do not specify a character set or use a domain, SQL uses the default character set.

- An optional default value for each column
- Optional constraints that apply to column data values or to the entire table

  Constraints specify that columns contain only certain data values, contain primary key values, contain unique data values, or that data values cannot be null. You can also create foreign key constraints, which specify that columns contain only the data values that correspond to values in another table.

  If you do not supply a name for a constraint, SQL supplies one on your behalf. However, because constraint names appear in SHOW TABLE displays and error messages, and you may need to refer to them by name in ALTER TABLE statements, you may prefer to explicitly specify your own names for constraints.

- Optional clauses for columns, including EDIT STRING and QUERY HEADER, that affect data displays and query characteristics for interactive SQL

  See Section 3.10.2.4 for introductory information on these optional formatting clauses. Note that domain definitions can specify formatting clauses; specifying formatting clauses for columns in the table definition overrides the counterpart clauses in domain definitions.

The examples in the following sections assume that your default database is mf_personnel_test and that you have created the domains for this database. (See Section 3.5 and Section 3.10 on creating the database and creating domains.)

If you created the mf_personnel_test database and its domains, but are no longer attached to it, enter an ATTACH statement to attach to it again.

The chapter on SQL statements in the *Oracle Rdb7 SQL Reference Manual* contains a section on the CREATE TABLE statement. That section provides additional rules and guidelines for this statement. The chapter on language and syntax elements in the *Oracle Rdb7 SQL Reference Manual* contains sections on data types, names, and optional formatting clauses for columns.

3.11.2.1 Specifying the Data Type of Columns

When you create a table and name the columns in the table, you specify the data type of the column. You can specify the data type either by explicitly defining the data type, length, and (for some numeric columns) the scale factor, or by naming a domain you have already defined.

Example 3–11 creates a table with columns that are not based upon user-defined domains; in other words, the data types are explicitly specified.

Example 3–11 Creating a Table and Specifying the Data Type of Columns

```
SQL> CREATE TABLE WORK_STATUS
cont>      ( STATUS_CODE     CHAR (1),
cont>        STATUS_NAME     CHAR (8),
cont>        STATUS_TYPE     CHAR (14)
cont>      );
SQL> --
SQL> SHOW TABLE (COLUMNS) WORK_STATUS
Information for table WORK_STATUS

Columns for table WORK_STATUS:
Column Name                     Data Type        Domain
-----------                     ---------        ------
STATUS_CODE                     CHAR(1)
STATUS_NAME                     CHAR(8)
STATUS_TYPE                     CHAR(14)

SQL> -- At this point, your table definition is still temporary.
SQL> -- Make the table definition a permanent addition to the
SQL> -- database with an explicit COMMIT statement.
SQL> --
SQL> COMMIT;
```

Example 3–12 illustrates the recommended practice of basing columns on domains. (The example assumes that you created the sample domains in Section 3.10.)

Example 3–12 Creating a Table with Columns Based on Domains

```
SQL> CREATE TABLE EMPLOYEES
cont>    (
cont>    EMPLOYEE_ID      ID_DOM,
cont>    LAST_NAME        LAST_NAME_DOM,
cont>    FIRST_NAME       FIRST_NAME_DOM,
cont>    MIDDLE_INITIAL   MIDDLE_INITIAL_DOM,
cont>    ADDRESS_DATA_1   ADDRESS_DATA_1_DOM,
cont>    ADDRESS_DATA_2   ADDRESS_DATA_2_DOM,
cont>    CITY             CITY_DOM,
cont>    STATE            STATE_DOM,
cont>    POSTAL_CODE      POSTAL_CODE_DOM,
cont>    SEX              SEX_DOM,
cont>    BIRTHDAY         DATE_DOM,
cont>    STATUS_CODE      STATUS_CODE_DOM
cont>    );
```

Note the domains supplied by SQL for columns in the WORK_STATUS table (Example 3–11) and your own domains for columns in the EMPLOYEES table (Example 3–12). When defining columns for your own tables, it is best to implement user-defined domains consistently. Specifying an explicit data type for STATUS_CODE in the WORK_STATUS table and a user-defined domain for STATUS_CODE in the EMPLOYEES table has resulted in different domains being the basis for the same kind of column.

Example 3–13 shows how to correct this problem.

Example 3–13 Creating a Table with One Column Based on Domains

```
SQL> -- Drop the WORK_STATUS table so that you can create it again.
SQL> --
SQL> DROP TABLE WORK_STATUS;
SQL> --
SQL> -- Create the table and base the STATUS_CODE column on the
SQL> -- STATUS_CODE_DOM domain.
SQL> --
SQL> CREATE TABLE WORK_STATUS
cont>       ( STATUS_CODE     STATUS_CODE_DOM,
cont>         STATUS_NAME     CHAR (8),
cont>         STATUS_TYPE     CHAR (14)
cont>       );
```

3.11.2.2 Assigning Character Sets to Columns

When you create a table, you can specify the character set for a column by qualifying the data type with a character set name or by using NCHAR or NCHAR VARYING. The following example shows how to specify different character sets for different columns in the same table:

```
SQL> ATTACH 'FILENAME mia_char_set';
SQL> --
SQL> SET CHARACTER LENGTH 'CHARACTERS'
SQL> --
SQL> CREATE TABLE COLOURS
cont>     (ENGLISH   MCS_DOM,
cont>      FRENCH    MCS_DOM,
cont> --
cont> -- The JAPANESE column uses the national character set defined by
cont> -- the KANJI_DOM domain.
cont>      JAPANESE  KANJI_DOM,
cont> --
cont> -- The ROMAJI column uses the database default character set defined
cont> -- by the DEC_KANJI_DOM domain.
cont> --
cont>      ROMAJI    DEC_KANJI_DOM);
SQL> SHOW TABLE (COLUMNS) COLOURS;
Information for table COLOURS

Columns for table COLOURS:
Column Name                     Data Type        Domain
-----------                     ---------        ------
ENGLISH                         CHAR(8)          MCS_DOM
         DEC_MCS 8 Characters,  8 Octets
FRENCH                          CHAR(8)          MCS_DOM
         DEC_MCS 8 Characters,  8 Octets
JAPANESE                        CHAR(4)          KANJI_DOM
         KANJI 4 Characters,  8 Octets
ROMAJI                          CHAR(8)          DEC_KANJI_DOM
SQL> COMMIT;
```

For more information, see the CREATE TABLE statement in the *Oracle Rdb7 SQL Reference Manual*.

3.11.2.3 Specifying the COMPUTED BY Clause

You can use the COMPUTED BY clause to create a column that holds a computed value. For example, if the social security rate for an employee is 6.10 percent of his starting salary, and the group insurance rate is 0.7 percent, you can define SS_AMT and GROUP_RATE fields as shown in the following example:

```
SQL> ATTACH 'FILENAME mf_personnel_test';
SQL> --
SQL> CREATE TABLE PAYROLL_DETAIL
cont>       (SALARY_CODE CHAR(1),
cont>        STARTING_SALARY SMALLINT(2),
cont>        SS_AMT
cont>          COMPUTED BY (STARTING_SALARY * 0.061),
cont>        GROUP_RATE
cont>          COMPUTED BY (STARTING_SALARY * 0.007));
```

When you use this type of definition, you only have to store values in the SALARY_CODE and STARTING_SALARY columns. The social security and group insurance deduction columns are computed automatically.

You can use a select expression in a COMPUTED BY clause. The following example shows how to use the COMPUTED BY clause to count the number of current employees of a particular department:

```
SQL> ATTACH 'FILENAME mf_personnel';
SQL> CREATE TABLE DEPTS1
cont>       (DEPARTMENT_CODE DEPARTMENT_CODE_DOM,
cont>         DEPT_COUNT   COMPUTED BY
cont>          (SELECT COUNT (*) FROM  JOB_HISTORY JH
cont>             WHERE JOB_END IS NULL
cont>             AND
cont> --
cont> -- Use correlation names to qualify the DEPARTMENT_CODE columns.
cont>             DEPTS1.DEPARTMENT_CODE = JH.DEPARTMENT_CODE),
cont>         DEPARTMENT_NAME DEPARTMENT_NAME_DOM)
cont> ;
SQL> -- For this example, load data from the DEPARTMENTS table.
SQL> INSERT INTO DEPTS1 (DEPARTMENT_CODE, DEPARTMENT_NAME)
cont>     (SELECT DEPARTMENT_CODE, DEPARTMENT_NAME FROM DEPARTMENTS);
26 rows inserted
SQL> --
SQL> SELECT * FROM DEPTS1 WHERE DEPARTMENT_CODE = 'ADMN';
 DEPARTMENT_CODE    DEPT_COUNT   DEPARTMENT_NAME
 ADMN                        7   Corporate Administration
1 row selected
SQL> ROLLBACK;
```

When you use the ANSI/ISO standard date-time data types, DATE ANSI, TIMESTAMP, or TIME, you can calculate date and time intervals using the COMPUTED BY clause. For example, you can create a column that calculates an employee's age based upon the difference between the current date and the employee's birth date.

```
SQL> ATTACH 'FILENAME mf_personnel_test';
SQL> --
SQL> -- By default, SQL interprets the DATE data type as DATE VMS.
SQL> -- To change the interpretation to DATE ANSI, use the
SQL> -- SET DIALECT 'SQL92' or SET DEFAULT DATE FORMAT 'SQL92' statement.
SQL> --
SQL> SET DIALECT 'SQL92';
SQL> --
SQL> CREATE TABLE EMPLOYEE_INFO
cont>      (EMPLOYEE_ID    CHAR(5),
cont>       LAST_NAME      CHAR(20),
cont>       FIRST_NAME     CHAR(14),
cont>       MIDDLE_INITIAL CHAR(1),
cont>       BIRTHDAY       DATE,          -- Interpreted as DATE ANSI
cont>       AGE
cont> --  Compute the employee's age to the year and month.
cont>          COMPUTED BY (CURRENT_DATE - BIRTHDAY) YEAR TO MONTH
cont>      );
```

The preceding example uses the built-in CURRENT_DATE function, which returns the current date. SQL also provides the CURRENT_TIME and CURRENT_TIMESTAMP functions.

3.11.2.4 Specifying Default Values for Columns

You can specify a default value for a column in a table. The default value for a column overrides any default value specified for the domain on which the column is based.

The default value of a column is the data value that is stored in the database if an insert operation on a row does not specify a data value for that column. You might have any of several possible reasons for specifying a default value for a column; for instance, you may want to store the most commonly used data value, or you may want to store a data value that highlights the fact that no value was stored.

For example, most employees in the sample personnel database live in New Hampshire (state code NH). If you want to store NH in the STATE column of an EMPLOYEES row whenever a data entry clerk does not enter a data value, specify a default value of NH. On the other hand, if you want to treat the absence of a STATE entry as an unusual condition (for example, if the data entry clerk does not know the new employee's state of residence when initially entering the information, but should find it out later), you could specify a default value of "?".

Example 3–14 specifies that the default value for the SUPERVISOR_ID column is a question mark (?).

Example 3–14 Specifying Default Values for Columns

```
SQL> CREATE TABLE JOB_HISTORY
cont>      (EMPLOYEE_ID      ID_DOM,
cont>       JOB_CODE         JOB_CODE_DOM,
cont>       JOB_START        DATE_DOM,
cont>       JOB_END          DATE_DOM,
cont>       DEPARTMENT_CODE  DEPARTMENT_CODE_DOM,
cont>       SUPERVISOR_ID    ID_DOM
cont>                       DEFAULT '?');
SQL> ROLLBACK;
```

3.11.2.5 Creating Constraints

To ensure the integrity of your data, you can define constraints. Applied correctly, constraints help to maintain the integrity of the database. For example, for every data value of a foreign key in one table, you want a matching data value in the primary key column of another table. When no such constraint checking is performed, it is possible to add a data value to the foreign key column in one table that does not match the primary key value in another table. Therefore, even though your database design is normalized, you want to ensure that the links between the foreign key in one table and a primary key in another are secure.

You can use constraints also to check another table for the presence of specific data values. Section 3.12 shows how to enhance the referential integrity of a database by using columns that refer to columns in another table and by using triggers.

Constraints that are specified as part of column definitions are called **column constraints**. Constraints that are specified independently of column definitions are called **table constraints**.

You can place constraints on columns and the entire table in the following ways:

- With the CHECK constraint clause in a CREATE TABLE, ALTER TABLE, CREATE DOMAIN, or ALTER DOMAIN statement

  A CHECK constraint clause checks the validity of the value in a table. The CHECK constraint clause consists of a predicate that column values inserted into the table must satisfy and an optional CONSTRAINT clause that specifies a name for the constraint.

  A CHECK constraint, as long as it does not refer to other columns in the table, can be included in a column definition. You can also specify a CHECK constraint independently of the column definition by separating the constraint definition from definitions of other elements in the table using a comma.

- By specifying NOT NULL or UNIQUE as column constraints in a CREATE TABLE or ALTER TABLE statement

  The NOT NULL constraint restricts the data values in the column to data values that are not null. The UNIQUE constraint specifies that the data values must be unique.

- By specifying UNIQUE as a table constraint in a CREATE TABLE or ALTER TABLE statement

  The UNIQUE constraint specifies that the combination of data values of the columns in a row must be unique.

- By specifying PRIMARY KEY as a column constraint or table constraint

  When you specify PRIMARY KEY as a column or table constraint, you declare the column or columns as the primary key. Because SQL requires that data values in a primary key column be unique and not null, you do not need to specify UNIQUE and NOT NULL for the primary key column.

- By specifying FOREIGN KEY as a table constraint

  When you specify FOREIGN KEY as a table constraint, you name the column or columns you want to declare as a foreign key. You specify foreign key constraints with the REFERENCES clause.

Note

You cannot specify constraints on columns of the LIST OF BYTE VARYING data type.

When you create a constraint, you can use the following clauses to specify when SQL evaluates the constraint:

- DEFERRABLE

  When a constraint has the DEFERRABLE attribute, it can be evaluated at any point including when the data is committed.

- NOT DEFERRABLE

  When a constraint has the NOT DEFERRABLE attribute, Oracle Rdb evaluates it when the statement executes.

You can also specify the default constraint evaluation by using the SET DIALECT statement. If you specify a dialect of SQLV40, the default constraint evaluation is deferrable. If you specify a dialect of SQL92, the default constraint evaluation is not deferrable. See the *Oracle Rdb7 Guide to SQL Programming* for more information about controlling constraint evaluation.

Example 3–15 illustrates creating column constraints when you create a table.

Example 3–15 Creating Column Constraints

```
SQL> CREATE TABLE EMPLOYEES
cont>   (
cont>   -- Employee identification (ID) number (EMPLOYEE_ID) uniquely
cont>   -- identifies rows in this table and therefore is a primary key.
cont>   -- The CONSTRAINT clause, which is optional but recommended,
cont>   -- specifies the name of the constraint.  Note that placement of
cont>   -- the comma indicates that the PRIMARY KEY constraint is defined
cont>   -- as part of the column definition:
cont>   --
cont>   EMPLOYEE_ID      ID_DOM
cont>     CONSTRAINT     EMP_EMPLOYEE_ID_PRIMARY_KEY
cont>     PRIMARY KEY
cont>     NOT DEFERRABLE,
cont>   LAST_NAME        LAST_NAME_DOM,
cont>   FIRST_NAME       FIRST_NAME_DOM,
cont>   MIDDLE_INITIAL   MIDDLE_INITIAL_DOM,
cont>   ADDRESS_DATA_1   ADDRESS_DATA_1_DOM,
cont>   ADDRESS_DATA_2   ADDRESS_DATA_2_DOM,
cont>   CITY             CITY_DOM,
cont>   STATE            STATE_DOM,
cont>   POSTAL_CODE      POSTAL_CODE_DOM,
cont>   --
cont>   -- A CHECK constraint limits the data values in the SEX column.
cont>   -- The constraint named EMP_SEX_VALUES is a column constraint.
cont>   --
cont>   SEX              SEX_DOM
cont>     CONSTRAINT     EMP_SEX_VALUES
cont>     CHECK          (SEX IN ('M', 'F') )
cont>     NOT DEFERRABLE,
cont>   BIRTHDAY         DATE_DOM,
cont>   STATUS_CODE      STATUS_CODE_DOM
cont>   );
```

You can create constraints that check column values by referring to data values stored in other tables. If a constraint on a table refers to more than one of its columns you must define it as a table constraint (that is, precede the constraint definition with a comma to separate it from the column definition).

In Example 3–16, the foreign key constraint named JH_EMPLOYEE_ID_IN_EMP specifies that to be valid for a JOB_HISTORY table row, an EMPLOYEE_ID value must already exist in the EMPLOYEES table. Similarly, the table constraint named JH_SUP_ID_IN_EMP specifies that to be valid the value in the SUPERVISOR_ID column must exist in the EMPLOYEE_ID column of the EMPLOYEES table.

Note that when you apply constraints that refer to other tables, you imply that the tables in your database are loaded with data in a certain order. For example, because of the JH_EMPLOYEE_ID_IN_EMP constraint, you cannot load an employee's job history record into the JOB_HISTORY table unless that employee already has a record in the EMPLOYEES table.

Example 3–16 Creating Table Constraints Based on Other Tables

```
SQL> CREATE TABLE JOB_HISTORY
cont>    (
cont>    EMPLOYEE_ID      ID_DOM,
cont>    -- The constraint JH_EMPLOYEE_ID_IN_EMP is a table constraint
cont>    -- because the preceding line terminates in a comma.
cont>    --
cont>      CONSTRAINT     JH_EMPLOYEE_ID_IN_EMP
cont>      FOREIGN KEY (EMPLOYEE_ID)
cont>      REFERENCES EMPLOYEES(EMPLOYEE_ID)
cont>      NOT DEFERRABLE,
cont>    JOB_CODE         JOB_CODE_DOM,
cont>    JOB_START        DATE_DOM,
cont>    JOB_END          DATE_DOM,
cont>    -- The table constraint JOB_END_AFTER_START specifies that the value
cont>    -- of JOB_END must be greater than JOB_START.
cont>       CONSTRAINT    JOB_END_AFTER_START
cont>       CHECK (JOB_END > JOB_START)
cont>       NOT DEFERRABLE,
cont>    DEPARTMENT_CODE  DEPARTMENT_CODE_DOM,
cont>    -- The table constraint JH_SUP_ID_IN_EMP specifies that the
cont>    -- value of SUPERVISOR_ID must exist in the EMPLOYEES table.
cont>    SUPERVISOR_ID    ID_DOM,
cont>      CONSTRAINT     JH_SUP_ID_IN_EMP
cont>      FOREIGN KEY    (SUPERVISOR_ID)
cont>      REFERENCES EMPLOYEES(EMPLOYEE_ID)
cont>      NOT DEFERRABLE
cont>    );
```

Constraint definitions can be modified as the needs of the physical definition change. However, if you modify a constraint in a table that contains data, Oracle Rdb evaluates the constraint at definition time to make sure that the table does not contain any rows that violate the constraint.

You can use the SHOW TABLES statement to find out which constraints have been created and how they were defined.

Enter a SHOW TABLES statement to display characteristics of two of the tables you just created, as shown in Example 3–17.

Example 3–17 Displaying Table Constraints

```
SQL> SHOW TABLES EMPLOYEES, JOB_HISTORY;
Information for table EMPLOYEES

Columns for table EMPLOYEES:

Column Name                     Data Type        Domain
-----------                     ---------        ------
EMPLOYEE_ID                     CHAR(5)          ID_DOM
 Primary Key constraint EMP_EMPLOYEE_ID_PRIMARY_KEY
   .
   .
   .
SEX                             CHAR(1)          SEX_DOM
BIRTHDAY                        DATE ANSI        DATE_DOM
STATUS_CODE                     CHAR(1)          STATUS_CODE_DOM

Table constraints for EMPLOYEES:
EMP_EMPLOYEE_ID_PRIMARY_KEY
 Primary Key constraint
 Column constraint for EMPLOYEES.EMPLOYEE_ID
 Evaluated on UPDATE, NOT DEFERRABLE
 Source:
        EMPLOYEES.EMPLOYEE_ID PRIMARY KEY

EMP_SEX_VALUES
 Check constraint
 Column constraint for EMPLOYEES.SEX
 Evaluated on UPDATE, NOT DEFERRABLE
 Source:
        CHECK  (SEX IN ('M', 'F') )

JH_EMPLOYEE_ID_IN_EMP
 Foreign Key constraint
 Table constraint for JOB_HISTORY
 Evaluated on UPDATE, NOT DEFERRABLE
 Source:
        FOREIGN KEY (EMPLOYEE_ID)
           REFERENCES EMPLOYEES(EMPLOYEE_ID)

JH_SUP_ID_IN_EMP
 Foreign Key constraint
 Table constraint for JOB_HISTORY
 Evaluated on UPDATE, NOT DEFERRABLE
```

(continued on next page)

Example 3–17 (Cont.) Displaying Table Constraints

```
 Source:
        FOREIGN KEY    (SUPERVISOR_ID)
          REFERENCES EMPLOYEES(EMPLOYEE_ID)
    .
    .
    .
Information for table JOB_HISTORY

Columns for table JOB_HISTORY:
Column Name                   Data Type        Domain
-----------                   ---------        ------
EMPLOYEE_ID                   CHAR(5)          ID_DOM
JOB_CODE                      CHAR(4)          JOB_CODE_DOM
JOB_START                     DATE ANSI        DATE_DOM
JOB_END                       DATE ANSI        DATE_DOM
DEPARTMENT_CODE               CHAR(4)          DEPARTMENT_CODE_DOM
SUPERVISOR_ID                 CHAR(5)          ID_DOM

Table constraints for JOB_HISTORY:
JH_EMPLOYEE_ID_IN_EMP
 Foreign Key constraint
 Table constraint for JOB_HISTORY
 Evaluated on UPDATE, NOT DEFERRABLE
 Source:
        FOREIGN KEY (EMPLOYEE_ID)
          REFERENCES EMPLOYEES(EMPLOYEE_ID)

JH_SUP_ID_IN_EMP
 Foreign Key constraint
 Table constraint for JOB_HISTORY
 Evaluated on UPDATE, NOT DEFERRABLE
 Source:
        FOREIGN KEY    (SUPERVISOR_ID)
          REFERENCES EMPLOYEES(EMPLOYEE_ID)

JOB_END_AFTER_START
 Check constraint
 Table constraint for JOB_HISTORY
 Evaluated on UPDATE, NOT DEFERRABLE
 Source:
        CHECK (JOB_END > JOB_START)
    .
    .
    .
```

Section 8.3 describes how to modify and delete constraints.

Using constraints affects performance in several important ways. SQL must place locks on one or more tables to check column values. As a result, SQL may 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 tables. For more information about join operations, see the *Oracle Rdb7 Introduction to SQL*. For information about whether to use constraints or indexes, see Section 3.14.3.

3.11.2.6 Implementing a UNIQUE OR NULL Constraint

You may want to implement a constraint that checks whether the data is either UNIQUE or NULL. There is no specific SQL syntax to define this type of constraint although you can implement the constraint easily using the CHECK constraint syntax. However, the choice of implementation can affect the efficiency of such a constraint.

Consider this problem: You want a order entry system to allow a credit card number to be NULL (that is, the customer is not paying by credit card) or not null. If it is not null, you want to ensure that no other customer has the same credit card number. That is, the number is UNIQUE. The following CHECK constraint implements this rule:

```
CHECK ((CC_NUMBER IS NULL) OR
       (NOT EXISTS (SELECT *
                    FROM CUSTOMERS C
                    WHERE CUSTOMERS.CC_NUMBER = C.CC_NUMBER
                          AND CUSTOMERS.DBKEY <> C.DBKEY)))
```

The NOT EXISTS predicate uses a subquery to find any matching credit card numbers. If any exist, the constraint fails. Because the check must avoid the current row being inserted, the constraint uses a dbkey comparison to filter the current row during the check of the table.

The presence of the OR operator forces the optimizer to disable dbkey retrieval so that correct semantics of the OR can be guaranteed. A sorted or hashed index on the column CC_NUMBER in the CUSTOMERS table would improve the retrieval times for this constraint because Oracle Rdb could perform an index-only retrieval on each of the retrievals:

```
~S: Constraint "UNIQUE_CC_NUMBER" evaluated
Cross block of 2 entries
  Cross block entry 1
    Index only retrieval of relation CUSTOMERS
      Index name  CC_INDEX [0:1]
  Cross block entry 2
    Conjunct        Aggregate-F1    Conjunct        Conjunct
    Index only retrieval of relation CUSTOMERS
      Index name  CC_INDEX [1:1]
```

You can use a CHECK constraint that uses less I/O than the previous example. The following CHECK constraint uses a different type of query to check the same things, but avoids an OR expression and so enables dbkey lookup for the IS NULL check.

```
CHECK ((SELECT COUNT(*)
        FROM CUSTOMERS C
        WHERE C.CC_NUMBER = CUSTOMERS.CC_NUMBER) <= 1)
```

This CHECK constraint counts all the credit card numbers which match the current value:

- If the value is one, this is the current row being inserted
- If the value is zero, the matching row must contain NULL. (Equality with NULL returns unknown, not TRUE, and so the row will not be counted.)
- Any other value indicates that this is a duplicate value and the constraint fails.

The resulting strategy performs less I/O than the initial attempt. This constraint performs correctly, as shown in the following example:

```
SQL> CREATE TABLE CUSTOMERS
cont>     ( CC_NUMBER BIGINT
cont>         CONSTRAINT UNIQUE_CC_NUMBER
cont>         CHECK((SELECT COUNT(*)
cont>                FROM CUSTOMERS C
cont>                WHERE C.CC_NUMBER=CUSTOMERS.CC_NUMBER) <= 1)
cont>         NOT DEFERRABLE,
cont>     -- Other customer field would exists in a real table
cont>     );
SQL> CREATE INDEX CC_INDEX ON CUSTOMERS (CC_NUMBER);
SQL>
SQL> INSERT INTO CUSTOMERS (CC_NUMBER)
cont>       VALUE (1234567890123456);
1 row inserted
```

```
SQL> -- The following statement fails because it is a duplicate.
SQL> INSERT INTO CUSTOMERS (CC_NUMBER)
cont>       VALUE (1234567890123456);
%RDB-E-INTEG_FAIL, violation of constraint UNIQUE_CC_NUMBER caused operation
to fail
-RDB-F-ON_DB, on database USER1:[TEST]SCRATCH.RDB;1
SQL> INSERT INTO CUSTOMERS (CC_NUMBER)
cont>       VALUE (NULL);
1 row inserted
```

The following example shows the optimizer strategy for this constraint evaluation. The SQL statement SET FLAGS 'strategy,request_name' was used to generate this output:

```
~S: Constraint "UNIQUE_CC_NUMBER" evaluated
Cross block of 2 entries
  Cross block entry 1
    Conjunct        Firstn  Get     Retrieval by DBK of relation CUSTOMERS
  Cross block entry 2
    Conjunct        Aggregate       Index only retrieval of relation CUSTOMERS
      Index name  CC_INDEX [1:1]
1 row inserted
```

3.12 Enforcing Referential Integrity Through Constraints and Triggers

Referential integrity refers to the consistency of related pieces of information across multiple tables in a database. Normally this involves ensuring the integrity of primary and foreign key relationships. That is, every value in a foreign key column must match a value in the primary key column it references, or it must be null. The following sections discuss how to ensure the referential integrity of a database using:

- Constraints that reference columns in other tables

  A column in a table can be defined as "referencing" a column in another table (often the primary key in the other table); such a definition establishes a constraint that prevents you from deleting a row that has rows in another table dependent upon it, or from adding or modifying rows without a corresponding matching row in another table.

- Triggers

  A **trigger** causes one or more actions to be performed before or after a particular write operation is performed (using an INSERT, DELETE, or UPDATE statement).

  Triggers are often used to enforce referential integrity.

3.12.1 Using Constraints to Enforce Referential Integrity

Constraints that establish references between columns in tables help to preserve the referential integrity of the database, ensuring that no changes are made that would violate certain dependencies among tables. A common use of such constraints is to preserve the integrity of relationships between a primary key and its associated foreign keys.

For example, assume that in the sample personnel database, you defined a constraint by which the EMPLOYEE_ID column in the SALARY_HISTORY table must reference the EMPLOYEE_ID column in the EMPLOYEES table. This is what the column definition from the CREATE TABLE statement would look like:

```
EMPLOYEE_ID      ID_DOM
  CONSTRAINT SALARY_HISTORY_EMPLOYEE_ID_REF
  REFERENCES EMPLOYEES (EMPLOYEE_ID),
```

This definition establishes the constraint that any employee ID entered in a SALARY_HISTORY row must match an existing employee ID in the EMPLOYEES table, and that no row can be deleted from the EMPLOYEES table as long as there are any rows in the SALARY_HISTORY table with that person's employee ID. In Example 3–18, which shows how to create a trigger, SALARY_HISTORY rows are deleted *before* the associated EMPLOYEES row is deleted. Any statement you enter that violates this constraint will fail (at definition time, execution time, or commit time, depending on when constraint evaluation is performed). See the *Oracle Rdb7 Guide to SQL Programming* for more information about constraint evaluation.

For further information about creating constraints that reference columns in other tables, see the CREATE TABLE section in the *Oracle Rdb7 SQL Reference Manual*.

3.12.2 Using Triggers to Enforce Referential Integrity

Triggers are often defined to cause one or more actions to be taken automatically before or after a particular write operation (using an INSERT, DELETE, or UPDATE statement) is performed. The particular operation causes the *triggered action* to take place, affecting columns or even entire rows in other tables or in the same table.

You can also use triggers to track changes to your database based on timestamps for a particular action. Example 3–18 includes a trigger that stores the employee ID of the employee being deleted, the name of the user doing the deletion, and the date and time of the deletion in the EMPLOYEE_ERASE_TAB table.

Example 3–18 defines a trigger that implements a *cascading delete* triggered by the deletion of an employee row. The trigger ensures that before an employee row is erased from the EMPLOYEES table, all of the employee's rows in the JOB_HISTORY and SALARY_HISTORY tables will also be erased. Additionally, the trigger ensures that if the employee in question is a department manager, the MANAGER_ID column for that department will be set to null.

Example 3–18 Creating a Trigger to Delete All Information About an Employee

```
SQL> ATTACH 'FILENAME mf_personnel';
SQL> -- If an employee is terminated, remove all associated rows
SQL> -- from the JOB_HISTORY  and SALARY_HISTORY tables,
SQL> -- and record event in EMPLOYEE_ERASE_TAB table.
SQL> --
SQL> -- Create table EMPLOYEE_ERASE_TAB to track deletions from the EMPLOYEES
SQL> -- table.
SQL> --
SQL> CREATE TABLE EMPLOYEE_ERASE_TAB
cont>       (NAME CHAR(31),
cont>        DELETE_ID ID_DOM,
cont>        ERASE_TIME DATE);
SQL> --
SQL> CREATE TRIGGER EMPLOYEE_ID_CASCADE_DELETE
cont>       BEFORE DELETE ON EMPLOYEES
cont>          (DELETE FROM JOB_HISTORY JH
cont>              WHERE JH.EMPLOYEE_ID = EMPLOYEES.EMPLOYEE_ID)
cont>          FOR EACH ROW
cont>          (DELETE FROM SALARY_HISTORY SH
cont>              WHERE SH.EMPLOYEE_ID = EMPLOYEES.EMPLOYEE_ID)
cont>          FOR EACH ROW
cont> --
cont> -- If an employee is terminated and that employee is
cont> -- the manager of a department, set the MANAGER_ID to null
cont> -- for that department.
cont> --
cont>          (UPDATE DEPARTMENTS D  SET D.MANAGER_ID = NULL
cont>              WHERE D.MANAGER_ID = EMPLOYEES.EMPLOYEE_ID)
cont>          FOR EACH ROW
cont> --
```

```
cont> -- Track all deletions in EMPLOYEE_ERASE_TAB table.
cont>          (INSERT INTO EMPLOYEE_ERASE_TAB
cont>              (NAME, DELETE_ID, ERASE_TIME)
cont>              VALUES (USER, EMPLOYEE_ID, CURRENT_TIMESTAMP))
cont>          FOR EACH ROW;
```

If you want to prevent deletion of a row in one table if a certain value exists in a column in another table, but you do *not* want to limit the rows in the second table to those that match a value in the first table, you can use a trigger instead of a constraint.

For example, in the sample personnel database, the DEGREES table contains the DEGREES_FOREIGN2 foreign key constraint, which is based on the COLLEGE_CODE primary key. The DEGREES_FOREIGN2 constraint prevents users from deleting a row in the COLLEGES table if the value in COLLEGE_CODE is used in the DEGREES table. In addition, it prevents users from inserting a row in the DEGREES table unless the COLLEGE_CODE value exists in the COLLEGES table. When you want to prevent users from deleting a row in the COLLEGES table if the COLLEGE_CODE is used in the DEGREES table, but you want to let them insert a row in the DEGREES table even if the COLLEGE_CODE does not exist in the COLLEGES table, use a trigger instead of the foreign key constraint.

Assume that the DEGREES table does not contain the DEGREES_FOREIGN2 foreign key constraint. In this case, users can insert into the DEGREES table a COLLEGE_CODE value that does not exist in the COLLEGES table. To prevent users from deleting a college from the COLLEGES table if that college is referred to in the DEGREES table, create a trigger as shown in Example 3–19.

Example 3–19 Creating a Trigger to Prevent Deleting a Row

```
SQL> CREATE TRIGGER NO_DELETE_COLLEGE_IF_IN_DEG
cont>       BEFORE DELETE ON COLLEGES
cont>          WHEN (EXISTS (SELECT DEGREES.COLLEGE_CODE FROM DEGREES
cont>                   WHERE DEGREES.COLLEGE_CODE = COLLEGES.COLLEGE_CODE))
cont>       (ERROR) FOR EACH ROW;
```

To avoid ambiguous syntax when you use the WHEN clause, use parentheses around the predicate.

Note

The execution of a triggered action is *not* guaranteed to occur at any specific point within the transaction; the only guarantee is that the

cumulative impact of any triggered actions will be in effect when the transaction is committed.

You should not assume that any specific triggered action will be executed immediately after the statement triggering it.

For example, assume that the following trigger has been defined to calculate the next sequence number to be assigned (by adding 1 to the count of orders):

```
SQL> CREATE TRIGGER SEQUENCE_NUM_TRIG
cont>    AFTER INSERT ON ORDERS_TABLE
cont>       (UPDATE SEQ_TABLE
cont>        SET SEQ_TABLE.NUMBER = (SELECT COUNT (*) FROM ORDERS_TABLE) + 1)
cont>   FOR EACH ROW;
```

Assume that the ORDERS_TABLE table contains 99 rows, and the value of SEQ_TABLE.NUMBER is 100. Your application stores 10 new rows in ORDERS_TABLE within a single transaction. Under the current implementation, each row insertion causes the SEQ_TABLE.NUMBER value to be updated; thus, after the hundredth ORDERS_TABLE row is inserted, NUMBER is set to 101; after the ORDERS_TABLE row number 101 is inserted, NUMBER is set to 102; and so forth.

However, this implementation may change in the future, so that the triggered actions are performed at the end after all 10 insertions, thus causing the value of SEQ_TABLE.NUMBER to increase from 100 to 110 only when the transaction is committed. Therefore, be sure to design applications so they do not depend on a particular timing of triggered actions within a transaction.

You should make sure that a trigger does not fire at the wrong time. For example, if you have an update trigger and you set the value to the same value as is currently stored in the column, the trigger will fire. Use the NEW and OLD context values of the WHEN predicate to prevent the execution of the trigger action if the actual column values did not change during the update.

Caution

Triggers can be very powerful mechanisms. They can make application development more efficient and enhance database consistency; however, if you are not careful, triggers can produce consequences you may not want. Triggers should be used within a carefully designed system of update procedures, to ensure reproducible and consistent update and retrieval operations.

To use the example of the EMPLOYEE_ID cascading deletion trigger, if you want to save certain information about employees *before* they terminate their employment, be sure to store the necessary information in an archival file or database *before* performing the deletion from the EMPLOYEES row that triggers the deletion of all the other related information.

You can cause infinite loops or inconsistent results to be returned if you define triggers that update rows or add rows to the trigger subject table. For example, if you have the following two conditions, you can encounter problems:

- A BEFORE UPDATE trigger on the EMPLOYEES table that inserts a row into the EMPLOYEES table
- An UPDATE statement that affects all the rows in the EMPLOYEES table

In these circumstances, the UPDATE statement will loop until all resources are consumed because, for each row updated, a new row will be added, which in turn will be updated, and so forth.

When subject table rows are being retrieved using an index, a triggered action operating on the same table could affect the index (by changing index key values or adding new keys) such that the triggering statement behaves in a different manner than when no trigger is involved.

You should construct any such triggers to operate only on rows that are either the current subject table row, or that will never be selected by the triggering statement. Another, more difficult method is to restructure triggering statements such that they never select a row that could have been updated or added by a trigger action. Some circumstances require a combination of these methods.

Triggers, either alone or in conjunction with other features, ensure the referential integrity of a database. For more information on triggers, see the CREATE TRIGGER section in the *Oracle Rdb7 SQL Reference Manual*. For information about the privileges required to create a trigger, see Table 9–1.

3.13 Creating Triggers to Invoke External Functions

You can use external functions in trigger definitions to perform processing that is external to the database. The external functions can respond to specific changes to the data no matter which application caused those changes.

External functions can invoke operating system services (for example, spawn a process or perform file operations), system library routines (for example, OpenVMS run-time library routines), or user-defined routines. Thus, by using the external functions in trigger definitions you can monitor trends, maintain

derived data outside the database, as well as provide external notification mechanisms.

For example, you can use an external function called from a trigger to send electronic mail to notify users of specific data changes in the database. In Example 3–20, when the number of items in the inventory reaches a low threshold, Oracle Rdb invokes the trigger NOTIFY_LOW_INVENTORY. The trigger adds items to a reorder list and invokes an external C function NOTIFY, which sends electronic mail to the user about the items that need to be reordered.

Example 3–20 Calling External Functions from Triggers

```
SQL>  -- Create the function.
SQL>  CREATE FUNCTION NOTIFY(
cont>                   INTEGER BY VALUE,
cont>                   INTEGER BY VALUE)
cont>                   RETURNS INTEGER;
cont>                   EXTERNAL LOCATION 'notify$exe'
cont>       LANGUAGE C
cont>       GENERAL PARAMETER STYLE VARIANT;
SQL>
SQL> -- Create the trigger.
SQL>  CREATE TRIGGER NOTIFY_LOW_INVENTORY
cont>       AFTER UPDATE OF NUM_ITEMS ON ITEMS
cont>       REFERENCING OLD AS OITEM
cont>                   NEW AS NITEM
cont>       WHEN (NITEM.NUM_ITEMS < ITEM_LOW)
cont>            (INSERT INTO REORDER_LIST
cont>             VALUES (NITEM.ID, NOTIFY(NITEM.ID, NITEM.NUM_ITEMS)))
cont>   FOR EACH ROW;
```

The function NOTIFY sends mail to a user named in the routine. The first argument in the external function definition is the identification number of the inventory item and second argument is the number of items remaining in stock.

Even if you do not want to insert or update data, you can use a trigger to call an external function. Example 3–21 uses an external function in the WHEN clause to conditionally activate an error condition. Because the function result always causes the error action to be avoided, you can call the function without any addition I/O to the database.

Example 3–21 Calling External Functions from Triggers to Reduce I/O

```
SQL> ATTACH 'FILENAME mf_personnel_test';
SQL> -- Create the function.
SQL> CREATE FUNCTION SEND_MAIL
cont>               (CHAR(32), CHAR(32), CHAR(256))
cont>       RETURNS INTEGER;
cont>       EXTERNAL NAME SEND_MAIL
cont>       LOCATION 'SUPPORT_FUNCTIONS'
cont>       LANGUAGE C  GENERAL PARAMETER STYLE;
SQL>
SQL> -- Create the trigger.
SQL> CREATE TRIGGER EMPLOYEE_DELETE_NOTIFY
cont>    BEFORE DELETE ON EMPLOYEES
cont>    WHEN SEND_MAIL ('db_administrator',
cont>                    'Employee ' || EMPLOYEES.EMPLOYEE_ID || ' deleted.',
cont>                    'User ' || CURRENT_USER || ' deleted employee ' ||
cont>                    EMPLOYEES.EMPLOYEE_ID )
cont>                      <> 0
cont>    (ERROR)
cont>    FOR EACH ROW;
```

In Example 3–21, the first argument is the name of the user to whom the mail is sent, the second argument is the subject line of the mail message and the third argument is the text of the mail message.

For information about the CREATE FUNCTION statement, see the *Oracle Rdb7 SQL Reference Manual*. For a detailed description of defining and using external functions, see the *Oracle Rdb7 Guide to SQL Programming*.

3.14 Creating Indexes

Indexes are special structures added to the database to speed searching for selected rows. You use the CREATE INDEX statement to specify one or more columns on which you want to base an index for a table. Then, when you perform an operation that requires searching or joining tables by that column, SQL can use the associated index to find rows directly, without a sequential scan of all rows in the table.

You can define the following types of indexes:

- Sorted index

  A sorted index can improve the performance of a variety of queries that search for rows by specifying a *range of values* in the column (or columns) on which the index is based.

If you want to perform ascending and descending scans on a table, you do not need to define two indexes. Oracle Rdb can perform ascending and descending scans using the same index. See the *Oracle Rdb7 Guide to Database Performance and Tuning* and the *Oracle Rdb7 SQL Reference Manual* for more information.

- Hashed index

  A hashed index can improve performance of queries that search for rows by specifying an *exact match of values* in the column or columns on which the index is based.

3.14.1 Creating Sorted Indexes

A sorted index can improve the performance of a variety of queries that search for rows by specifying a *range of values* in the column (or columns) on which the index is based. A **sorted index** is a tree structure of nodes. The database system navigates a sorted index by reading nodes on progressively lower levels of the tree until it finds the entry that contains the location of a particular row.

A sorted index improves the performance of queries that compare values using range operators, not exact match operators. A sorted index, unlike a hashed index, improves the performance of:

- Range retrieval operators, such as BETWEEN, greater than (>), and less than (<)
- Built-in functions, such as SUM, MIN, MAX, and AVG
- Retrieval operations that use the ORDER BY clause

  If the sorting criteria of the ORDER BY clause is the same as the index, a sorted index may prevent an extra step in the sorting process.

If the preceding operators compare values with columns on which a sorted index is based, the database system can often process column values directly from the index without having to retrieve the rows.

You can specify two types of structures for sorted indexes:

- A nonranked B-tree structure
- A ranked B-tree structure

  The ranked B-tree structure allows better optimization of queries, particularly queries involving range retrievals. Oracle Rdb is able to make better estimates of cardinality, reducing disk I/O and lock contention.

Oracle Rdb recommends that you use ranked sorted indexes. To specify that Oracle Rdb create an index with this structure, use the RANKED keyword of the TYPE IS SORTED clause, as shown in Example 3–22.

Example 3–22 Creating a Sorted Index Using the RANKED keyword

```
SQL> CREATE UNIQUE INDEX COLL_CODE_IND
cont>       ON COLLEGES (COLLEGE_CODE)
cont>       TYPE IS SORTED RANKED;
```

Example 3–22 creates a sorted index named COLL_CODE_IND for the COLLEGES table. It is based on the column COLLEGE_CODE. Because the index definition specifies the UNIQUE clause, it allows no duplicate values to be stored in the index.

If a sorted ranked index allows duplicate entries, you can store many more records in a small space when you compress duplicates, using the DUPLICATES ARE COMPRESSED clause of the CREATE INDEX statement. When you compress duplicates, Oracle Rdb uses byte-aligned bitmap compression to represent the dbkeys for the duplicate entries instead of chaining the duplicate entries together with uncompressed dbkeys. In addition to the savings in storage space, you minimize I/O, increasing performance.

Example 3–23 shows how to create the index JH_EMP_ID, which allows duplicates (by omitting the UNIQUE keyword) and compress duplicate entries.

Example 3–23 Compressing Duplicate Index Entries

```
SQL> CREATE INDEX JH_EMP_ID
cont>       ON JOB_HISTORY  (EMPLOYEE_ID)
cont>       TYPE IS SORTED RANKED
cont>       DUPLICATES ARE COMPRESSED;
```

The DUPLICATES ARE COMPRESSED clause is the default when you use the RANKED keyword. The clause is optional.

You cannot use the DUPLICATES ARE COMPRESSED clause when you create nonranked indexes.

Example 3–24 illustrates how to create a nonranked sorted index. The index, EMP_EMPLOYEE_ID, is based on the column EMPLOYEE_ID. It allows no duplicate values to be stored in the index.

When you create unique indexes, you should support the index definition by defining a column constraint of NOT NULL.

Example 3–24 Creating a Sorted Index

```
SQL> CREATE UNIQUE INDEX EMP_EMPLOYEE_ID
cont>       ON EMPLOYEES (EMPLOYEE_ID);
```

To allow duplicate values in an index, simply omit the UNIQUE keyword.

For more information about sorted indexes, see the following sections:

- Section 3.14.6, which describes the types of index compression you can use
- Section 4.7, which describes how to set the characteristics of sorted indexes to improve performance
- Chapter 4, especially Section 4.6.1, which discusses how to achieve optimal performance for range retrieval

3.14.2 Creating Hashed Indexes

A hashed index can improve performance of queries that search for rows by specifying an *exact match of values* in the column or columns on which the index is based. A **hashed index** is a set of records called hash buckets, each of which contains dbkey entries for all rows from the table that are stored on a specific page of a storage area. The hashing algorithm stores records scattered randomly throughout the file. If there is not sufficient space on a page to store all the rows from the table, the database system stores the rows on adjacent pages. The database system uses a hashing algorithm, which uses values in the column (or columns) on which the index is based, to locate which hash bucket contains the dbkey for a row.

When a table is large, row retrieval using a hashed index can be faster than row retrieval using a sorted index; however, the database system uses a hashed index only for exact match searches. A query that compares values using the equal (=) operator selects rows by exact match.

A hashed index is effective only if you specify the full key values directly; it cannot process a range of column values.

Even for exact match row retrieval, the performance advantage of a hashed index over a sorted index is not likely to be apparent when a table stores only a few hundred or a thousand rows. In this case, the database system may require one or very few input/output operations to read an entire sorted index into memory. Remember this if you plan to test the performance benefits of

indexes by creating a database prototype with only a subset of the rows that you will load in the production database.

To realize any performance benefits from hashed indexes, you must pay careful attention to storage parameters. In particular, the page format, page size, and initial file allocation for a storage area are critical parameters for hashed index performance. See Chapter 4 for more discussion of sorted and hashed indexes and how indexes are related to the storage design for your database.

When you create a hashed index, you can specify one of the following hashing algorithms:

- HASHED SCATTERED

  The HASHED SCATTERED algorithm is appropriate in most situations, such as when the index key values are not evenly distributed across a range of values or when you cannot predict the index key values. The algorithm scatters the data across a storage area. As a result, the record distribution pattern is not usually uniform; some pages are chosen as targets more often than others.

  Use the HASHED SCATTERED algorithm unless your data meets the criteria for the HASHED ORDERED algorithm.

- HASHED ORDERED

  The HASHED ORDERED algorithm is ideal for applications where the key values are uniformly distributed across a range. That is, the HASHED ORDERED algorithm should be used when an application has a range of index key values, and each key value occurs the same number of times. An application with a range of sequential index key values between 1 and 100,000 with no duplicate values is an example of an application that would benefit from using the HASHED ORDERED algorithm.

  Use the HASHED ORDERED option *only* when all of the following are true:

  - The index keys are integer, date, timestamp, or interval values.
  - The index is defined as an ASCENDING index.
  - You do not use the MAPPING VALUES clause when defining the index (so the index does not compress all-numeric index key values).

  If the index key values are not evenly distributed, records could be distributed very unevenly and the HASHED SCATTERED option would be a better choice.

For more information about the HASHED SCATTERED and HASHED ORDERED algorithms, see the *Oracle Rdb7 Guide to Database Performance and Tuning*.

Example 3–25 demonstrates creating a hashed index based on the column EMPLOYEE_ID.

Example 3–25 Creating a Hashed Index

```
SQL> CREATE INDEX EMP_HASH_INDEX ON EMPLOYEES
cont>       (EMPLOYEE_ID)
cont>       TYPE IS HASHED
cont>       STORE USING (EMPLOYEE_ID)
cont>             IN PERSONNEL_1 WITH LIMIT OF ('00399')
cont>             IN PERSONNEL_2 WITH LIMIT OF ('00699')
cont>             IN PERSONNEL_3 WITH LIMIT OF ('10000');
```

For more information about hashed indexes, see Chapter 4, especially Section 4.6.2, which discusses how to achieve optimal performance for exact match retrieval.

3.14.3 Deciding Between an Index and a Constraint to Enforce Unique Column Values

Example 3–24 defines an index that does not allow duplicate values. It should be supported by a NOT NULL constraint on the associated column. Alternatively, you could forego the UNIQUE keyword in the index definition entirely and specify PRIMARY KEY or NOT NULL and UNIQUE constraints on the associated column. To keep users from storing duplicate values in a column, either strategy will work. Each has its advantages and disadvantages.

Most database administrators implement the unique value requirement through the index definition and specify only NOT NULL as a supporting column constraint. If you implement unique values through the index definition and users attempt to store a duplicate value in a column, users get specific information about the kind of error they made. (A constraint violation can encompass a variety of problems and the error text is general.)

However, implementing uniqueness as a column constraint allows you to defer checking whether or not values are unique until commit time. (Commit-time checking of constraints provides performance benefits when a transaction processes many rows.)

Use the method of ensuring unique values that benefits the users and applications at your site. In general, avoid enforcing uniqueness through both a constraint definition and an index definition. The performance overhead you incur by asking a database system to check a value twice (both for the row change or commit operation and for update of the index) makes it worthwhile to choose one and only one way to enforce unique column values. In addition, a dual strategy for enforcing uniqueness adds complexity to error-handling strategies in programs. If you enforce uniqueness both in the index and as a column constraint, the expected error returned to the program depends on when constraints are evaluated. If constraints are evaluated at the time a row is stored, a general constraint violation is returned to the program. If constraints are evaluated at commit time, a duplicate value (index) violation would be encountered before the constraint violation.

3.14.4 Deciding When Indexes Are Beneficial

Generally, indexes improve the performance of data retrieval and degrade the performance of data update. Like most general rules, however, the preceding one suffers from oversimplification.

A data retrieval task, for example, must search for rows or order rows in a table using data values in the indexed column to derive any benefit from the index. In addition, certain complex searches, particularly those involving join operations, may not use an index defined for one of the tables being searched because the database system finds it more efficient to search that particular table sequentially.

Conversely, operations that update or delete rows must retrieve rows before writing to the database, and can benefit from an index that speeds searches for the rows being changed or deleted. The performance degradation caused by an index occurs only when users change data values in columns on which an index is based. The INSERT and DELETE statements always update all indexes defined for a table. The UPDATE statement causes changes in an index only if it changes data values in indexed columns of the table.

Deciding which indexes to define for your database is often a matter of trading off one user's gain against another user's loss. The rest of this section provides more information to help you determine when a trade-off is worthwhile.

When a table is large, an index based on a column value that uniquely identifies each row can improve performance considerably when SELECT statements use the column to search for specific rows. Typically, columns that uniquely identify a row in a table are not changed after a row is stored. If the row has a relatively long life span and will be frequently retrieved or updated, the index is definitely worth defining.

The benefits of indexes outweigh their disadvantages for columns that contain duplicate data values as long as the table is being accessed more often for row retrieval than to insert or delete rows. For example, indexes can improve the performance of SELECT statements that specify the associated column in a GROUP BY clause. The MIN and MAX functions can be improved by an index on the column that contains the data value for which you want the minimum or maximum. In this case, Oracle Rdb can save time by retrieving the value from the index itself.

In addition, indexes are especially important for columns on which tables are joined. In this case, matching rows in tables by columns that have an associated index is often necessary to get results quickly.

In summary, you want to define indexes that speed performance of queries users are most likely to make. You can define as many indexes for a table as you think you need, but do not create indexes that are seldom or never used. Also, remember that creating multiple indexes for tables and basing them on columns that are frequently updated can seriously degrade the performance of transactions that insert, erase, or modify rows in the table.

Note

Unless you are creating storage maps for tables that depend on your indexes, you may want to defer creating sorted indexes for a database until after a database is loaded. This can substantially improve the performance of the load operation. (You should define hashed indexes before loading.) If you intend to define indexes with the UNIQUE clause, be sure that the data you are loading does not have duplicate values in the columns on which you will base those UNIQUE indexes.

For more information about index creation and load operations, see Section 6.1.

After your database is defined and loaded, you can monitor database activity to determine if the indexes are appropriate. Then, you can delete some indexes or add new ones to reflect data retrieval and update patterns. If database update is done at a different time than data retrieval (overnight or on weekends, for example), you have an additional alternative. Programs that update many rows can delete indexes not used during update transactions and then create the indexes again after update transactions are complete. Deleting indexes not used during update transactions on a table may speed execution of the program considerably if many rows are processed.

As an alternative to deleting and creating indexes again, you may want to look into some of the user-defined storage options that SQL offers you. For these options, see Chapter 4.

The logical data model in Figure 3–1 supplies examples of columns for which you would probably want to create an index. All columns that are primary or foreign keys are likely to be specified as SELECT statements as the basis for combining rows in different tables. Therefore, these columns are likely to need associated indexes. Some columns, such as the "Employee ID number" in the "Employee Personal Data" table, also uniquely identify rows in a table. If the table is large and users often retrieve its rows by "Employee ID number," the column should have an associated index even if that column is seldom used to join tables.

In the example in the preceding paragraph, some tables can be selected for multiple index definitions. If these are tables that are frequently updated with large amounts of data, you may want to define only those indexes that provide considerable performance improvements for the most frequent type of database queries.

The "Work status code" column in the "Employee Personal Data" tables is an example of a column on which you might base an index and find out that the costs of the index outweigh its benefits.

In the table "Employee Personal Data," the column "Work status code" contains so many duplicate values that using direct retrieval of rows may yield little performance advantage over sequentially searching the table. Indexes based on columns that contain many duplicate values also degrade table update performance more seriously than indexes based on columns that have few or no duplicate values.

The table "Employee Job History" in Figure 3–1 contains three foreign keys that probably need associated indexes for data retrieval ("Employee ID number," "Job code," and "Department code"). Suppose the table is updated by a program that runs overnight when no one is retrieving data from the database. Such a program is likely to use only the index for the "Employee ID number" column. If you find that the program takes too long to execute, you may try deleting the other two indexes for the duration of the update transactions. You can create the indexes again when the update transactions are complete. You may find that it takes less time to both update data and make indexes current when these operations are done separately rather than together.

The DROP INDEX and ALTER INDEX statements are discussed in Chapter 7.

For more information about creating indexes, see Chapter 4 and the section on the CREATE INDEX statement in the *Oracle Rdb7 SQL Reference Manual*. That section also explains how you can base an index on more than one column.

3.14.5 Creating Indexes Concurrently

You can create indexes at the same time other users are creating indexes, even if the indexes are on the same table. To allow concurrent index definition on the same table, use the SHARED DATA DEFINITION clause of the SET TRANSACTION statement.

The following example shows how to reserve the table for shared data definition and how to create an index:

```
SQL> SET TRANSACTION READ WRITE
cont>    RESERVING EMPLOYEES FOR SHARED DATA DEFINITION;
SQL>
SQL> CREATE INDEX EMP_LAST_NAME1 ON EMPLOYEES (EMPLOYEE_ID);
SQL> --
SQL> -- Commit the transaction immediately.
SQL> COMMIT;
```

When you use the SHARED DATA DEFINITION clause, be aware of the following points:

- You cannot query or update the reserved tables in the same transaction.
- Other users cannot query or update the reserved tables.
- Other users cannot perform any data definition operation on the reserved tables, other than creating indexes.
- To minimize lock conflicts with other users, commit the transaction immediately.
- All users who are defining indexes on the same table must reserve the table using the SHARED DATA DEFINITION clause.

Because of the method Oracle Rdb uses to detect unique index names during concurrent index definition, Oracle Rdb returns an error if you use index names that are longer than 27 characters and first 27 characters of the new index name does not represent a unique name. This occurs whether or not you use the SHARED DATA DEFINITION clause. For example, if the database contains the index DAILY_SALES_SUMMARY_01_04_95 and you attempt to a create a new index DAILY_SALES_SUMMARY_01_04_96, Oracle Rdb returns an error.

3.14.6 Creating Compressed Indexes

You can specify that indexes should be compressed. In a **compressed index**, information within the index is reduced in size so that data takes up less space. You can specify the following types of compression:

- **Run-length compression** compresses a sequence of space characters (octets) from text data types and binary zeros from nontext data types. (Different character sets have different representations of the space character. Oracle Rdb compresses the representation of the space character for the character sets of the columns comprising the index values.) Run-length compression is most useful when you have many sequences of space characters or binary zeros. You can specify run-length compression for hashed and sorted indexes.
- **SIZE IS segment truncation**, used with text or varying text columns, limits the number of characters used for retrieving data. You can specify segment truncation for sorted indexes only.
- **MAPPING VALUES compression**, used with numeric columns, translates the column values into a more compactly encoded form.
- **Duplicates compression** compresses duplicate dbkeys. You can specify duplicates compression for sorted ranked indexes only. For more information, see Section 3.14.1.

Run-length index compression results in the following benefits and costs:

- For hashed indexes, compression frees space on pages for more data. You can place more index values in a hash bucket.
- For hashed indexes, compression results in fewer I/Os because of reduced overflow.
- For sorted indexes, compression can significantly reduce the disk space used.
- For sorted indexes with a large number of index values, compression results in fewer I/Os.
- For sorted indexes, compression may reduce the level of concurrent activity.

  Because compression reduces the size of index keys, more index keys can fit in each node. Therefore, you lock more index values each time you update a row. To increase concurrency, reduce the node size so that each node contains fewer index keys.

SIZE IS segment truncation, MAPPING VALUES compression, and duplicates compression result in the following benefits (for sorted indexes only):

- Storage requirements for some applications are much lower.
- Fewer I/O operations are needed to retrieve data because more user index nodes may be included in buffers.
- Index-only retrieval is more efficient because more data may reasonably be included in an index.

The following sections describe how to specify run-length compression, SIZE IS segment truncation, and MAPPING VALUES compression. For information about specifying duplicates compression, see Section 3.14.1. For information about sizing indexes, see Section 4.8.3 and Section 4.7. For information about using RMU commands to help you size compressed indexes, see the *Oracle Rdb7 Guide to Database Performance and Tuning*.

3.14.6.1 Creating Run-Length Compressed Indexes

To create a run-length compressed index, use the ENABLE COMPRESSION MINIMUM RUN LENGTH clause of the CREATE INDEX statement. The value you specify in the MINIMUM RUN LENGTH clause indicates the minimum length of the sequence that Oracle Rdb should compress. For example, if you specify MINIMUM RUN LENGTH 2, Oracle Rdb compresses sequences of two or more spaces or two or more binary zeros.

As it compresses the sequences, Oracle Rdb replaces the sequence with the number of space characters or binary zeros specified by the minimum run-length value *plus* an extra byte that contains information about the number of characters compressed for that sequence. If many of the index values contain one space between characters in addition to trailing spaces, use a minimum run length of 2 so that you do not inadvertently expand the index.

The following example creates a compressed index on the DEPARTMENTS table:

```
SQL> CREATE INDEX DEPT_NAME_IND ON DEPARTMENTS (DEPARTMENT_NAME)
cont>       ENABLE COMPRESSION
cont>       (MINIMUM RUN LENGTH 2);
```

You cannot specify which characters are compressed, only the minimum length. Oracle Rdb determines which characters are compressed.

Because Oracle Rdb replaces the sequence with the value of the minimum run length *plus* 1 byte when it compresses indexes, you can inadvertently expand the index beyond the 255 character limit if you do not choose the minimum run length correctly.

For more information about when to use run-length compression, see the *Oracle Rdb7 Guide to Database Performance and Tuning*.

3.14.6.2 Creating SIZE IS Segment-Truncated Indexes

To create a SIZE IS compressed index for columns that use the CHAR or VARCHAR data types, use the SIZE IS clause of the CREATE INDEX statement for the column or columns being indexed, as shown in the following example:

```
SQL> CREATE INDEX EMP_LAST_NAME
cont>    ON EMPLOYEES (LAST_NAME SIZE IS 10)
cont>    TYPE IS SORTED;
```

Although you can create a SIZE IS compressed index and specify the UNIQUE clause, truncating the index key values may make the key values not unique. In that case, the index definition or insert or update statements will fail. For example, if you define the index EMP_LAST_NAME as UNIQUE and attempt to insert a row for an employee with the last name “Kilpatrick” and a row for an employee with the last name “Kilpatricks”, SQL returns the following error:

```
%RDB-E-NO_DUP, index field value already exists; duplicates not allowed for
EMP_LAST_NAME
```

3.14.6.3 Creating Mapping Values Compressed Indexes

To create a MAPPING VALUES compressed index for columns that use numeric data types, use the MAPPING VALUES clause of the CREATE INDEX statement for the column or columns being indexed. In the following example, YEAR_NUMBER is a column that has been defined with the UNIQUE clause.

```
SQL> CREATE INDEX PS_DATE_2 ON PRODUCT_SCHEDULE
cont>       (PRODUCT_ID,
cont>        YEAR_NUMBER  MAPPING VALUES 1970 to 2070,
cont>        PRODUCT_DESCR SIZE is 20);
```

3.15 Creating Temporary Tables

Often, you may need to store temporary results only for a short duration, perhaps to temporarily store the results of a query so that your application can act on the results of that query. You can create a table, store the results in a table, and drop the table when you are finished. As an alternative, you can use temporary tables to store temporary results without repeatedly creating and dropping tables. Temporary tables also provide the convenience of storing the data in a table and using SQL statements to manipulate the data, rather than storing the data in a flat file.

Temporary tables are tables whose data is automatically deleted when an SQL session or module ends. The tables only materialize when you refer to them in an SQL session and the data does not persist beyond an SQL session. The data in temporary tables is private to the user.

You can create the following types of temporary tables:

- Global temporary tables

  The metadata for a global temporary table is stored in the database and it persists beyond the SQL session. Different SQL sessions can share the same metadata.

  The data stored in the table cannot be shared between SQL sessions, although the data can be shared between modules in a single SQL session. The data does not persist beyond an SQL session.

- Local temporary tables

  The metadata for a local temporary table is stored in the database and it persists beyond the SQL session. Different SQL sessions can share the same metadata.

  The data stored in the table cannot be shared between different modules in a single SQL session or between SQL sessions. The data does not persist beyond an SQL session or module.

- Declared local temporary tables

  The metadata for a declared local temporary table is not stored in the database and cannot be shared by other modules. These tables are sometimes called **scratch tables**.

  The data stored in the table cannot be shared between SQL sessions or modules in a single session. The metadata and data do not persist beyond an SQL session.

Because the data in temporary tables is private to an SQL session, Oracle Rdb takes out locks only during data definition, resulting in increased performance.

You can use the ON COMMIT clause to specify whether the data is preserved or deleted when you commit the transaction.

Temporary tables are stored in virtual memory, not in a storage area. They use the same storage segment layout as persistent base tables, but they use additional space in memory for management overhead. On OpenVMS, temporary tables use 56 bytes per row for management overhead; on Digital UNIX, they use 88 bytes.

See Section 3.15.3 for information about estimating the virtual memory needs of temporary tables.

Because data in temporary tables is private to a session and because the metadata for declared temporary tables is not stored in the database, you cannot use temporary tables in as many places as you use persistent base tables. In particular, note the following points when you use temporary tables:

- You can drop global and local temporary tables using the DROP TABLE statement, but you cannot drop declared temporary tables.
- You cannot modify a temporary table. To modify a global or local temporary table, you must drop the table and create it again. You cannot modify or drop a declared local temporary table.
- You can truncate global temporary tables using the TRUNCATE TABLE statement, but you cannot truncate local temporary tables or declared local temporary tables.
- Temporary tables cannot contain data of the data type LIST OF BYTE VARYING.
- You can define constraints for global temporary tables, but not for local temporary tables or declared temporary tables. However, you can use domain constraints in global and local temporary tables or declared temporary tables.

  Constraints on a global temporary table can only refer to another global temporary table. If the referenced target table specifies ON COMMIT DELETE ROWS, the source table must specify ON COMMIT DELETE ROWS. This restriction does not apply if the referenced target table specifies ON COMMIT PRESERVE ROWS.
- You cannot grant or revoke privileges on declared temporary tables. On global and local temporary tables, you can grant and revoke privileges only using the ALL keyword.

See the *Oracle Rdb7 SQL Reference Manual* for more information about what actions you can take with temporary tables and when you can refer to temporary tables.

3.15.1 Creating Global and Local Temporary Tables

When you create a global or local temporary table, the metadata persists beyond the end of the SQL session. As a result, you can use the table definition again and again.

Oracle Rdb does not materialize the table until you refer to it in a session. Each SQL session that refers to a *global temporary table* causes a distinct instance of that table to be materialized. That instance is shared among all modules activated during the session. By contrast, each module or precompiled SQL program (in an SQL session) that refers to a *local temporary table* causes a distinct instance of that table to be materialized.

Assume that you have a base table called PAYROLL which is populated with data and that you want to extract the current week's information to generate pay checks for the company. The following example shows the definition for the PAYROLL table:

```
SQL> CREATE TABLE PAYROLL
cont>       (EMPLOYEE_ID ID_DOM,
cont>        WEEK_DATE DATE ANSI,
cont>        HOURS_WORKED INTEGER,
cont>        HOURLY_SAL INTEGER(2));
```

Example 3–26 shows how to create a global temporary table, PAYCHECKS_GLOB, and populate it with data from the PAYROLL and EMPLOYEES tables. Your application can now operate on the data in PAYCHECKS_GLOB to calculate deductions and net pay for each employee.

Example 3–26 Creating a Global Temporary Table

```
SQL> CREATE GLOBAL TEMPORARY TABLE PAYCHECKS_GLOB
cont>       (EMPLOYEE_ID ID_DOM,
cont>        LAST_NAME CHAR(14),
cont>        HOURS_WORKED INTEGER,
cont>        HOURLY_SAL   INTEGER(2),
cont>        WEEKLY_PAY   INTEGER(2))
cont>        ON COMMIT PRESERVE ROWS;
SQL> --
SQL> -- Insert data into the temporary tables from other existing tables.
SQL> INSERT INTO PAYCHECKS_GLOB
cont>         (EMPLOYEE_ID, LAST_NAME, HOURS_WORKED, HOURLY_SAL, WEEKLY_PAY)
cont>          SELECT P.EMPLOYEE_ID, E.LAST_NAME, P.HOURS_WORKED, P.HOURLY_SAL,
cont>                 P.HOURS_WORKED * P.HOURLY_SAL
cont>                 FROM EMPLOYEES E, PAYROLL P
cont>                 WHERE E.EMPLOYEE_ID = P.EMPLOYEE_ID
cont>                   AND P.WEEK_DATE = DATE '1995-08-01';
100 rows inserted
```

(continued on next page)

Example 3–26 (Cont.) Creating a Global Temporary Table

```
SQL> --
SQL> -- Display the data.
SQL> SELECT * FROM PAYCHECKS_GLOB LIMIT TO 2 ROWS;
 EMPLOYEE_ID   LAST_NAME        HOURS_WORKED       HOURLY_SAL       WEEKLY_PAY
 00165         Smith                      40            30.50          1220.00
 00166         Dietrich                   40            36.00          1440.00
2 rows selected
SQL> -- Commit the data.
SQL> COMMIT;
SQL> --
SQL> -- Because the global temporary table was created with PRESERVE ROWS,
SQL> -- the data is preserved after you commit the transaction.
SQL> SELECT * FROM PAYCHECKS_GLOB LIMIT TO 2 ROWS;
 EMPLOYEE_ID   LAST_NAME        HOURS_WORKED       HOURLY_SAL       WEEKLY_PAY
 00165         Smith                      40            30.50          1220.00
 00166         Dietrich                   40            36.00          1440.00
2 rows selected
```

You can use the ON COMMIT clause to specify that the data persist after a commit or that the data is deleted after a commit. In Example 3–26, the table definition specifies that the rows are preserved after the transaction is committed.

Because the PAYCHECKS_GLOB table is a global temporary table, different modules in the same session can share the data. When you end the SQL session (by using the DISCONNECT statement), Oracle Rdb deletes the data in the temporary table.

The following series of examples demonstrate the difference in scope between global and local temporary tables.

Assume that the modules PAYCHECK_INS_GLOB_MOD and LOW_HOURS_GLOB_MOD, shown in Example 3–27, are stored in the database. These modules insert data into and query the global temporary table, PAYCHECKS_GLOB.

Example 3–27 Creating Stored Modules That Use Global Temporary Tables

```
CREATE MODULE PAYCHECK_INS_GLOB_MOD
  LANGUAGE SQL
  PROCEDURE PAYCHECK_INS_GLOB;
```

(continued on next page)

Example 3–27 (Cont.) Creating Stored Modules That Use Global Temporary Tables

```
    BEGIN
      INSERT INTO PAYCHECKS_GLOB
            (EMPLOYEE_ID, LAST_NAME, HOURS_WORKED, HOURLY_SAL, WEEKLY_PAY)
            SELECT P.EMPLOYEE_ID, E.LAST_NAME, P.HOURS_WORKED, P.HOURLY_SAL,
                   P.HOURS_WORKED * P.HOURLY_SAL
            FROM EMPLOYEES E, PAYROLL P
            WHERE E.EMPLOYEE_ID = P.EMPLOYEE_ID
              AND P.WEEK_DATE = DATE '1995-08-01';
    END;
END MODULE;

CREATE MODULE LOW_HOURS_GLOB_MOD
  LANGUAGE SQL
  PROCEDURE LOW_HOURS_GLOB (:cnt INTEGER);
    BEGIN
      SELECT COUNT(*) INTO :cnt FROM PAYCHECKS_GLOB
             WHERE HOURS_WORKED < 40;
    END;
END MODULE;
```

Because PAYCHECKS_GLOB is a global temporary table, different modules in the same session can share the data. As Example 3–28 demonstrates, the procedure PAYCHECK_INS_GLOB from one module and the procedure LOW_HOURS_GLOB from another module share the data in the table.

Example 3–28 Sharing Data in Global Temporary Tables

```
SQL> -- Insert data into the table.
SQL> CALL PAYCHECK_INS_GLOB();
SQL> DECLARE :cnt integer;
SQL> -- Using a procedure from another module, query the table.  The data
SQL> -- inserted by the PAYCHECK_INS_GLOB procedure can be seen and used
SQL> -- by LOW_HOURS_GLOB.
SQL> CALL LOW_HOURS_GLOB(:cnt);
         CNT
           2
```

In contrast, different modules cannot share the data in local temporary tables. Assume that the temporary table PAYCHECKS_LOCAL and the modules PAYCHECK_INS_LOCAL_MOD and LOW_HOURS_LOCAL_MOD, all shown in Example 3–29, are stored in the database. The modules insert data into and query the local temporary table, PAYCHECKS_LOCAL.

Example 3–29 Creating Local Temporary Tables and Stored Modules That Use the Table

```
-- Create the local temporary table.
CREATE LOCAL TEMPORARY TABLE PAYCHECKS_LOCAL
       (EMPLOYEE_ID ID_DOM,
        LAST_NAME CHAR(14),
        HOURS_WORKED INTEGER,
        HOURLY_SAL   INTEGER(2),
        WEEKLY_PAY   INTEGER(2))
        ON COMMIT PRESERVE ROWS;

-- Create the stored modules.
CREATE MODULE PAYCHECK_INS_LOCAL_MOD
  LANGUAGE SQL
  PROCEDURE PAYCHECK_INS_LOCAL;
    BEGIN
      INSERT INTO PAYCHECKS_LOCAL
            (EMPLOYEE_ID, LAST_NAME, HOURS_WORKED,HOURLY_SAL, WEEKLY_PAY)
            SELECT P.EMPLOYEE_ID, E.LAST_NAME, P.HOURS_WORKED, P.HOURLY_SAL,
                   P.HOURS_WORKED * P.HOURLY_SAL
            FROM EMPLOYEES E, PAYROLL P
            WHERE E.EMPLOYEE_ID = P.EMPLOYEE_ID
              AND P.WEEK_DATE = DATE '1995-08-01';
    END;
END MODULE;

CREATE MODULE LOW_HOURS_LOCAL_MOD
  LANGUAGE SQL
  PROCEDURE LOW_HOURS_LOCAL (:cnt INTEGER);
    BEGIN
      SELECT COUNT(*) INTO :cnt FROM PAYCHECKS_LOCAL
             WHERE HOURS_WORKED < 40;
    END;
END MODULE;
```

Because PAYCHECKS_LOCAL is a local temporary table, different modules in the same session *cannot* share the data. As Example 3–30 demonstrates, the procedure PAYCHECK_INS_LOCAL from one module and the procedure LOW_HOURS_LOCAL from another module do not see the same instance of the table.

Example 3–30 Isolating Data in Local Temporary Tables

```
SQL> -- Insert data into the table.
SQL> CALL PAYCHECK_INS_LOCAL();
SQL> DECLARE :cnt integer;
SQL> -- Using a procedure from another module, query the table.   The data
SQL> -- inserted by the PAYCHECK_INS_LOCAL procedure cannot be seen by
SQL> -- the procedure LOW_HOURS_LOCAL.
SQL> CALL LOW_HOURS_LOCAL(:cnt);
         CNT
           0
```

3.15.2 Creating Declared Local Temporary Tables

When you create a declared local temporary table, the metadata and data do not persist beyond the end of the SQL session (in interactive SQL) or the end of the stored module which contains the table declaration.

You can use a declared local temporary table only in the following environments:

- Interactive SQL
- Dynamic SQL
- In a stored procedure

When you use a declared local temporary table, you must precede the table name with the keyword MODULE and a period (.).

Example 3–31 shows how to declare and use a declared local temporary table in interactive SQL.

Example 3–31 Declaring a Local Temporary Table in Interactive SQL

```
SQL> DECLARE LOCAL TEMPORARY TABLE MODULE.PAYCHECK_DECL_INT
cont>        (EMPLOYEE_ID ID_DOM,
cont>         LAST_NAME CHAR(14),
cont>         HOURS_WORKED INTEGER,
cont>         HOURLY_SAL   INTEGER(2),
cont>         WEEKLY_PAY   INTEGER(2))
cont>        ON COMMIT PRESERVE ROWS;
SQL> --
```

(continued on next page)

Example 3–31 (Cont.) Declaring a Local Temporary Table in Interactive SQL

```
SQL> INSERT INTO MODULE.PAYCHECK_DECL_INT
cont>         (EMPLOYEE_ID, LAST_NAME, HOURS_WORKED, HOURLY_SAL, WEEKLY_PAY)
cont>          SELECT P.EMPLOYEE_ID, E.LAST_NAME, P.HOURS_WORKED,
cont>                 P.HOURLY_SAL, P.HOURS_WORKED * P.HOURLY_SAL
cont>             FROM EMPLOYEES E, PAYROLL P
cont>                 WHERE E.EMPLOYEE_ID = P.EMPLOYEE_ID
cont>                   AND P.WEEK_DATE = DATE '1995-08-01';
100 rows inserted
SQL> SELECT * FROM MODULE.PAYCHECK_DECL_INT LIMIT TO 2 ROWS;
 EMPLOYEE_ID   LAST_NAME       HOURS_WORKED      HOURLY_SAL       WEEKLY_PAY
 00165         Smith                     40           30.50          1220.00
 00166         Dietrich                  40           36.00          1440.00
2 rows selected
```

You cannot use the SHOW TABLE statement to display declared local temporary tables.

Example 3–32 shows how to create a stored module which contains the following:

- A declared local temporary table, MODULE.PAYCHECK_DECL_TAB
- A procedure, PAYCHECK_INS_DECL, which inserts weekly salary records into the declared local temporary table
- A procedure, LOW_HOURS_DECL, which counts the number of employees with less than 40 hours worked

Example 3–32 also demonstrates that you can access the declared local temporary table only from within the module.

Example 3–32 Using Declared Local Temporary Tables in Stored Procedures

```
SQL> -- Create the module containing a declared temporary table.
SQL> CREATE MODULE PAYCHECK_DECL_MOD
cont>   LANGUAGE SQL
cont>   DECLARE LOCAL TEMPORARY TABLE MODULE.PAYCHECK_DECL_TAB
cont>         (EMPLOYEE_ID ID_DOM,
cont>          LAST_NAME CHAR(14) ,
cont>          HOURS_WORKED INTEGER, HOURLY_SAL INTEGER(2),
cont>          WEEKLY_PAY   INTEGER(2))
cont>          ON COMMIT PRESERVE ROWS
```

(continued on next page)

Example 3–32 (Cont.) Using Declared Local Temporary Tables in Stored Procedures

```
cont> -- Create the procedure to insert rows.
cont>   PROCEDURE PAYCHECK_INS_DECL;
cont>   BEGIN
cont>     INSERT INTO MODULE.PAYCHECK_DECL_TAB
cont>          (EMPLOYEE_ID, LAST_NAME, HOURS_WORKED, HOURLY_SAL, WEEKLY_PAY)
cont>          SELECT EMPLOYEE_ID, LAST_NAME, P.HOURS_WORKED, P.HOURLY_SAL,
cont>                 P.HOURS_WORKED * P.HOURLY_SAL
cont>                 FROM EMPLOYEES NATURAL JOIN PAYROLL P
cont>                 WHERE P.WEEK_DATE = DATE'1995-08-01';
cont>   END;
cont>
cont> -- Create the procedure to count the low hours.
cont>   PROCEDURE LOW_HOURS_DECL (OUT :cnt INTEGER);
cont>   BEGIN
cont>     SELECT COUNT(*) INTO :cnt FROM MODULE.PAYCHECK_DECL_TAB
cont>            WHERE HOURS_WORKED < 40;
cont>   END;
cont> END MODULE;
SQL>
SQL> -- Call the procedure to insert the rows.
SQL> CALL PAYCHECK_INS_DECL();
SQL>
SQL> -- Declare a variable and call the procedure to count records with
SQL> -- low hours.
SQL> DECLARE :low_hr_cnt integer;
SQL> CALL LOW_HOURS_DECL(:low_hr_cnt);
  LOW_HR_CNT
           2

SQL> -- Because the table is a declared local temporary table, you cannot
SQL> -- access it from outside the stored module which contains it.
SQL> SELECT * FROM MODULE.PAYCHECK_DECL_TAB;
%SQL-F-RELNOTDCL, Table PAYCHECK_DECL_TAB has not been declared in module or
environment
```

The procedure PAYCHECK_INS_DECL uses the syntax NATURAL JOIN. A natural join is equivalent to `TABLE_A.COL = TABLE_B.COL`. For more information about natural joins, see the *Oracle Rdb7 Introduction to SQL*.

You can qualify the name of the table with an alias name. For example, if the database alias is PERS, the qualified name of PAYCHECK_DECL_TAB is PERS.MODULE.PAYCHECK_DECL_TAB.

3.15.3 Estimating Virtual Memory for Temporary Tables

Because temporary tables are stored in virtual memory, if you use very large temporary tables or many temporary tables, you may exceed your memory limits. Temporary tables use the same storage segment layout as persistent base tables, but they use additional space in memory for management overhead.

Table 3–1 shows how to calculate the amount of memory needed to store temporary tables.

Table 3–1 Calculating Memory Usage for Temporary Tables

	Size in Bytes	
Category	**OpenVMS**	**Digital UNIX**
Management Overhead	56	88
Header	5	5
Version number	2	2
Null bytes	Based on Number of Columns[1]	Based on Number of Columns[1]
User data	Table row size	Table row size

[1] 1 byte per 8 columns calculated by the formula: null bytes = (no. of columns in the row +7) /8

For example, if you create a temporary table with the same columns as the EMPLOYEES table in the personnel database, there are 2 null bytes and 112 bytes of user data. (Table 4–4 calculates the column sizes for the EMPLOYEES table.) As a result, one row in the table uses 177 bytes of memory on OpenVMS:

$$56 + 5 + 2 + 2 + 112 = 177$$

OpenVMS VAX OpenVMS Alpha

If you use very large temporary tables or many temporary tables, you may need to increase the Page File Quota and the Virtual Page Count on OpenVMS. Because temporary tables are stored in memory, you may lose the data if you exceed the limits. ♦

3.16 Creating Views

The design of the sample databases separates the employee personnel data into logically related groups. Because the normalization process often results in defining additional tables, gathering data from these tables can be cumbersome. Accessing data that occurs in several different tables might mean entering the same complex queries repeatedly. However, SQL provides an efficient method to make these queries "permanent." You can create views to combine different portions of many tables in the database.

You can think of a view as a virtual table. A **view** does not store data but looks like a table to a database user. Accessing a view is a convenient way for a user to retrieve a subset of columns from one table or to retrieve combinations of columns stored in different tables. A view can also include and name virtual columns based on arithmetic calculations.

The new columns you name in the view definition have the same column attributes as the columns in the original tables. Section 8.5 shows you how to modify a view by dropping it and creating it again.

The following guidelines may help you decide which views to create when you first set up a database. You will probably find that you need more views once the database is in use:

- Create a view to prevent users from seeing data in a table that they are not authorized to see. You can define views to access subsets of table columns when you want to restrict user access to tables. You can define protection so that certain users can access only the view and not the table or tables on which the view is based.

  For example, in the EMPLOYEES table, most users may need to access only name and address information for current employees. You can create a view of the EMPLOYEES table that contains only this information and allow most users to access the view but not the table.

- Create views to meet the needs of reports that your site runs regularly.
- Create views to meet the needs of queries that interactive users frequently make.

Views provide performance enhancements. Performing a join that involves many tables can 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, it is usually more efficient to base all view definitions on the database tables themselves.

View definitions can specify formatting clauses that affect data display and query characteristics for interactive SQL users. Note that a view does not inherit column characteristics that may be specified by EDIT STRING, QUERY HEADER, QUERY NAME, and DEFAULT VALUE FOR DTR clauses in the definitions of tables on which the view is based.

You cannot create indexes for a view. Views use the indexes created for the tables they access. If you create a new view for your database, check its performance with the statement SELECT * FROM view-name. If data display seems unreasonably slow, defining new indexes for the tables that support the view may improve performance.

If a view is not read-only, users can update the database using a view but should do so with caution. When updating the database using a view, users may encounter error conditions they do not expect or cause unexpected changes in underlying tables. In general, Oracle Rdb recommends that you update tables directly.

However, by using the CHECK OPTION clause of the CREATE VIEW statement, you can ensure that any rows that a user updates or inserts into a view conform to the definition of the view. For example, the following view definition ensures that users cannot insert or update a row if the DEPARTMENT_CODE column does not equal 'ADMN'.

```
SQL> CREATE VIEW ADMIN_VIEW1
cont>     AS SELECT EMPLOYEE_ID,
cont>           JOB_CODE,
cont>           JOB_START,
cont>           DEPARTMENT_CODE
cont>     FROM JOB_HISTORY
cont>     WHERE JOB_END IS NULL
cont>           AND
cont>           DEPARTMENT_CODE = 'ADMN'
cont>  WITH CHECK OPTION;
```

Even if the view definition is based upon other views (nested views), the CHECK OPTION clause ensures that any changes to the data conform to the definition of the view.

You cannot update a view if it is read-only. SQL considers the following views read-only:

- Views that contain the following clauses and expressions:
  - Functions
  - GROUP BY
  - HAVING

- DISTINCT
- UNION

- Views that name more than one table or view in the FROM clause

The *Oracle Rdb7 SQL Reference Manual* contains a section on the CREATE VIEW statement. Refer to that section for special rules that apply to view definitions and the SELECT expression you include in view definitions. In addition, the section provides examples of problems that can occur when you use views to update the database.

As the examples in the following sections show, joining tables can be complex. If you frequently form SELECT expressions to retrieve rows from several tables, you might consider creating a view definition. A view can bring together columns from one or more tables based on a SELECT expression specified in the view definition. A user can refer to the view definition as if it were a single table and use SQL statements to display or manipulate column values. Thus, a user who might not understand the syntax for a complex join can still access data from such a join when it is defined in a view.

Before defining a view, you can join two, three, or more tables with a SELECT statement to be sure that you are accessing the correct data. Once you have determined that the data is correct, you can use the same columns from the join to create a view.

The sample personnel and mf_personnel databases each contains three view definitions. Each view models a different transaction in its view definition. Two of these views, CURRENT_JOB and CURRENT_SALARY, refer to two tables in a database. The third view, CURRENT_INFO, is more complex than the others because it refers to both tables and views in its definition.

The examples in the following sections illustrate how to create the three sample database views and show how to create views containing columns that compute dates.

3.16.1 Creating the CURRENT_JOB View

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

Table	Column
EMPLOYEES	EMPLOYEE_ID
EMPLOYEES	FIRST_NAME
EMPLOYEES	LAST_NAME
JOB_HISTORY	JOB_START
JOB_HISTORY	JOB_CODE
JOB_HISTORY	SUPERVISOR_ID
JOB_HISTORY	DEPARTMENT_CODE

Now you can form a query that joins these two tables. Because both the tables contain the EMPLOYEE_ID column, you can use this column as the join term in your SELECT expression.

```
SQL> SELECT E.LAST_NAME,
cont>        E.FIRST_NAME,
cont>        E.EMPLOYEE_ID,
cont>        JH.JOB_CODE,
cont>        JH.DEPARTMENT_CODE,
cont>        JH.SUPERVISOR_ID,
cont>        JH.JOB_START
cont>   FROM EMPLOYEES E, JOB_HISTORY JH
cont>   WHERE E.EMPLOYEE_ID = JH.EMPLOYEE_ID;
```

The JOB_HISTORY table may contain many job history rows for one employee. This query retrieves all job history rows for every employee. You can restrict the record stream further by requiring only the current job history row for each employee. No data value is stored in the JOB_END column for a current job history row. Therefore, you can find a current job history row by selecting rows in the JOB_HISTORY table where the JOB_END column is NULL. The following example adds a clause to the SELECT expression to include only current job history rows with rows from the EMPLOYEES table:

```
SQL> SELECT E.LAST_NAME,
cont>        E.FIRST_NAME,
cont>        E.EMPLOYEE_ID,
cont>        JH.JOB_CODE,
cont>        JH.DEPARTMENT_CODE,
cont>        JH.SUPERVISOR_ID,
cont>        JH.JOB_START
cont>   FROM EMPLOYEES E, JOB_HISTORY JH
cont>   WHERE E.EMPLOYEE_ID = JH.EMPLOYEE_ID
cont>         AND JOB_END IS NULL;
```

This query brings together the columns that are needed from both tables 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 definitions in the database. Example 3–33 shows how to create this view.

Example 3–33 Creating the CURRENT_JOB View

```
SQL> CREATE VIEW CURRENT_JOB
cont>   AS SELECT E.LAST_NAME,
cont>         E.FIRST_NAME,
cont>         E.EMPLOYEE_ID,
cont>         JH.JOB_CODE,
cont>         JH.DEPARTMENT_CODE,
cont>         JH.SUPERVISOR_ID,
cont>         JH.JOB_START
cont>   FROM EMPLOYEES E, JOB_HISTORY JH
cont>   WHERE E.EMPLOYEE_ID = JH.EMPLOYEE_ID
cont>         AND JOB_END IS NULL;
```

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

```
SQL> SELECT * FROM CURRENT_JOB
cont>   WHERE EMPLOYEE_ID = '00164';
 LAST_NAME        FIRST_NAME  EMPLOYEE_ID  JOB_CODE  DEPARTMENT_CODE
   SUPERVISOR_ID   JOB_START
 Toliver          Alvin       00164        DMGR      MBMN
   00228           21-SEP-1981 00:00:00.00

1 row selected
```

3.16.2 Creating the CURRENT_SALARY View

You can follow the same steps to create the CURRENT_SALARY view as shown in Section 3.16.1. The CURRENT_SALARY view joins the EMPLOYEES table with the SALARY_HISTORY table. First determine which columns you need from each table:

Table	Column
EMPLOYEES	LAST_NAME
EMPLOYEES	FIRST_NAME
EMPLOYEES	EMPLOYEE_ID
SALARY_HISTORY	SALARY_START
SALARY_HISTORY	SALARY_AMOUNT

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

```
SQL> SELECT E.LAST_NAME,
cont>       E.FIRST_NAME,
cont>       E.EMPLOYEE_ID,
cont>       SH.SALARY_START,
cont>       SH.SALARY_AMOUNT
cont>  FROM EMPLOYEES E, SALARY_HISTORY SH
cont>  WHERE E.EMPLOYEE_ID = SH.EMPLOYEE_ID
cont>        AND SH.SALARY_END IS NULL;
```

Now that you see the join works successfully, you can create the CURRENT_SALARY view as shown in Example 3–34.

Example 3–34 Creating the CURRENT_SALARY View

```
SQL> CREATE VIEW CURRENT_SALARY
cont>   AS SELECT E.LAST_NAME,
cont>         E.FIRST_NAME,
cont>         E.EMPLOYEE_ID,
cont>         SH.SALARY_START,
cont>         SH.SALARY_AMOUNT
cont>   FROM EMPLOYEES E, SALARY_HISTORY SH
cont>   WHERE E.EMPLOYEE_ID = SH.EMPLOYEE_ID
cont>         AND SALARY_END IS NULL;
```

3.16.3 Creating the CURRENT_INFO View

The third view in the sample databases uses the first two views and two other tables 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 columns distributed among several tables in the database. Again, you start by selecting the list of columns you need from each table or view:

Table or View	Column
CURRENT_JOB	LAST_NAME
CURRENT_JOB	FIRST_NAME
CURRENT_JOB	EMPLOYEE_ID
DEPARTMENTS	DEPARTMENT_NAME
JOBS	JOB_TITLE
CURRENT_JOB	JOB_START
CURRENT_SALARY	SALARY_START
CURRENT_SALARY	SALARY_AMOUNT

Views offer another feature that lets you create customized column names from the columns in the referenced tables and views. You assign new, local column names, listing them after the name of the view.

The view definition for CURRENT_INFO includes a SELECT expression to join the two views and the two tables and specifies the new column names to refer to the original column names. Example 3–35 shows how to create the CURRENT_INFO view.

Example 3–35 Creating the CURRENT_INFO View

```
SQL> CREATE VIEW CURRENT_INFO
cont>     (LAST_NAME,
cont>      FIRST_NAME,
cont>      ID,
cont>      DEPARTMENT,
cont>      JOB,
cont>      JSTART,
cont>      SSTART,
cont>      SALARY )
cont>   AS SELECT CJ.LAST_NAME,
cont>       CJ.FIRST_NAME,
cont>       CJ.EMPLOYEE_ID,
cont>        D.DEPARTMENT_NAME,
cont>        J.JOB_TITLE,
cont>       CJ.JOB_START,
cont>       CS.SALARY_START,
cont>       CS.SALARY_AMOUNT
cont>   FROM CURRENT_JOB CJ,
cont>      DEPARTMENTS D,
cont>      JOBS J,
cont>      CURRENT_SALARY CS
cont>   WHERE CJ.DEPARTMENT_CODE = D.DEPARTMENT_CODE
cont>         AND CJ.JOB_CODE = J.JOB_CODE
cont>         AND CJ.EMPLOYEE_ID = CS.EMPLOYEE_ID;
```

If you are not satisfied with the definitions of the views created in the preceding examples, you can enter a ROLLBACK statement to delete them from the physical definition. Once committed, these views can be readily modified as the requirements of the application change. You can use the SHOW statement specifying the VIEWS option to see which views have been defined.

To modify an existing view, delete the view and create it again. To delete a view already committed to the database, enter a DROP VIEW statement. Section 8.5 describes how to modify and delete views.

3.16.4 Creating Views to Calculate Dates

You can create views that contain columns that calculate dates, times, or the amount of time between dates or times. The EXTRACT function lets you select a single date or time field from a DATE ANSI, TIME, TIMESTAMP, or INTERVAL data type. The CAST function lets you explicitly convert a column from one data type to another. If your database contains a column of the DATE VMS data type, you can use the CAST function to convert the column to DATE ANSI, TIME, or TIMESTAMP data types, so that you can perform computations on the column. (Remember that you cannot perform calculations on columns with the DATE VMS data type.)

For example, you can create a view to find the employees in the personnel database who have 15 or more years of service with the company. Because an employee may have held more than one job with the company and thus may have more than one start date, you need to create two views.

The first view, YEARS_EMPLOYED, calculates how long each employee has been with the company. The view uses the MIN function to find the earliest start date for each employee. Then, it uses the CAST function to convert the JOB_START column, which has a DATE VMS data type, to the DATE ANSI data type. Finally, it subtracts the job start date from the current date and specifies the interval in years and months.

The second view, LONG_TERM_EMPLOYEES, uses the EMPLOYMENT_DURATION column from the YEARS_EMPLOYED view and the EMPLOYEE_ID and LAST_NAME column from the EMPLOYEES table to select those employees who have been employed by the company for 15 years or more.

Example 3–36 shows how to create a view to find the employees who have 15 or more years of service with the company.

Example 3–36 Creating a View That Contains Records for Employees with 15 or More Years of Service

```
SQL> -- Create a view to calculate how long each employee has been with
SQL> -- the company.
SQL> --
SQL> CREATE VIEW YEARS_EMPLOYED (EMPLOYEE_ID, EMPLOYMENT_DURATION)
cont>   AS SELECT EMPLOYEE_ID,
cont>        (CURRENT_DATE - CAST(MIN(JOB_START) AS DATE ANSI)) YEAR TO MONTH
cont>   FROM JOB_HISTORY
cont>   GROUP BY EMPLOYEE_ID;
SQL> --
```

(continued on next page)

Example 3–36 (Cont.) Creating a View That Contains Records for Employees with 15 or More Years of Service

```
SQL> -- Create a view to select those employees who have been employed by
SQL> -- the company for 15 years or more.
SQL> --
SQL>  CREATE VIEW LONG_TERM_EMPLOYEES
cont>             (EMPLOYEE_ID, LAST_NAME, EMPLOYMENT_DURATION)
cont>    AS SELECT  E.EMPLOYEE_ID, E.LAST_NAME, Y.EMPLOYMENT_DURATION
cont>    FROM EMPLOYEES E, YEARS_EMPLOYED Y
cont>    WHERE E.EMPLOYEE_ID = Y.EMPLOYEE_ID
cont>       AND Y.EMPLOYMENT_DURATION >= INTERVAL '15' YEAR
cont>    ORDER BY EMPLOYMENT_DURATION;
```

The CORPORATE_DATA database contains an example of creating a view using the EXTRACT function. In addition, the *Oracle Rdb7 SQL Reference Manual* and the *Oracle Rdb7 Introduction to SQL* contain more information about the EXTRACT and CAST functions and calculating dates and times.

4 Implementing a Multifile Database

This chapter explains how you can use multifile databases to improve statement response time and transaction throughput by assigning table data, snapshot files, and indexes to different files. Before reading this chapter, you should read Chapter 3, which explains how you use the Oracle Rdb data definition language to create a database and provides introductory information about multifile databases.

4.1 Deciding on a Storage Design for Your Multifile Database

To create a multifile database, you specify one or more CREATE STORAGE AREA clauses in a CREATE DATABASE statement. When you do so, the logical areas for definitions and for data are uncoupled from the root area. The database root file contains pointers to one or more storage area files that contain definitions and data.

When you create a multifile database, it always contains the storage area named RDB$SYSTEM. If your CREATE DATABASE statement contains CREATE STORAGE AREA clauses, but none of these refer to RDB$SYSTEM, Oracle Rdb automatically creates the file for the RDB$SYSTEM storage area.

For the following reasons, consider defining more than one storage area:

- You can assign different storage areas to files on different disks to increase the number of disk drives that handle database input/output (I/O) operations.

  To disperse tables and indexes to the different storage areas (and by doing so, to different files and disks), specify CREATE STORAGE MAP statements (for tables) and include STORE clauses in CREATE INDEX statements (for indexes).

- In each CREATE STORAGE AREA clause, you can assign the storage area's snapshot (.snp) file to a different disk than its storage area file to further spread I/O operations to prevent a disk I/O bottleneck. (When you enable snapshot capability for a multifile database, Oracle Rdb creates a separate snapshot file for each data storage area that you create.)

If you do not include SNAPSHOT FILENAME clauses in the CREATE STORAGE AREA clauses, Oracle Rdb creates .snp files with the same file names and on the same disks as the associated .rda files.

- You can specify a page format that is suited for general use (uniform) or a page format that can be customized to enhance particular queries (mixed).

Example 4–1, an excerpt from a command procedure, shows how to create a multifile database with four storage areas on OpenVMS. (Creating a multifile database on Digital UNIX uses the same syntax. The only differences are the file specifications.)

Example 4–1 Creating the Multifile personnel_db Database

```
$ SQL
SQL> @sample_db.com
CREATE DATABASE FILENAME persdisk0:[mfp]personnel_db
    RESERVE 15 STORAGE AREAS
    RESERVE 15 JOURNALS
CREATE STORAGE AREA PERSONNEL_MISC
    FILENAME persdisk0:[mfp]pers0
    PAGE FORMAT IS UNIFORM
   .
   .
   .
CREATE STORAGE AREA PERSONNEL_1
    FILENAME persdisk1:[mfp]pers1
    PAGE FORMAT IS MIXED
   .
   .
   .
CREATE STORAGE AREA PERSONNEL_2
    FILENAME persdisk2:[mfp]pers2
    PAGE FORMAT IS MIXED
   .
   .
   .
CREATE STORAGE AREA PERSONNEL_3
    FILENAME persdisk3:[mfp]pers3
    PAGE FORMAT IS MIXED
   .
   .
   .
```

The CREATE DATABASE statement in Example 4–1 creates the following files:

- In the persdisk0:[mfp] directory, the files personnel_db.rdb, personnel_db.rda, personnel_db.snp, pers0.rda, and pers0.snp

The RDB$SYSTEM storage area is named persdisk0:[mfp]personnel_db.rda.

- In the persdisk1:[mfp] directory, the files pers1.rda and pers1.snp
- In the persdisk2:[mfp] directory, the files pers2.rda and pers2.snp
- In the persdisk3:[mfp] directory, the files pers3.rda and pers3.snp

Because the CREATE DATABASE statement does not specify otherwise, Oracle Rdb enables snapshot files by default.

4.2 Understanding General Storage Options for a Multifile Database

Without extensive database design expertise, you can manipulate the storage characteristics of a multifile database to avoid an I/O bottleneck at any disk serving the database.

When you first allocate storage for a database, you may have no idea how many disk drives you need to avoid I/O bottlenecks. In this case, you might begin by creating a database with a separate storage area for each table that you expect to be heavily accessed or that is too large to share a disk with other tables. If you have two disks available for your database files, distribute the storage areas for heavily accessed tables between the two disks.

To allow for future growth, reserve additional slots for storage areas using the RESERVE STORAGE AREAS clause. If you reserve a sufficient number of storage area slots, you can add storage areas without interrupting database activity. See Section 3.5.5 and Section 7.6.1 for more information.

OpenVMS VAX OpenVMS Alpha

If database performance is slow after your database is loaded with data and in use, on OpenVMS, you can use the DCL command MONITOR DISK /ITEM=QUEUE to determine if I/O queues are developing for any disks serving the database. If you see queues forming, you can use the Oracle Rdb Performance Monitor to see which storage areas are accessed most heavily. See the *Oracle Rdb7 Guide to Database Performance and Tuning* for information about using the Performance Monitor. ♦

If several heavily accessed storage areas (data storage or snapshot areas) are assigned to a disk where I/O operations are queued, you should redistribute at least one of those storage areas to another disk using the RMU Move_Area command. See Section 7.6.4 for information about how to use the RMU Move_Area command to move storage areas.

If heavy access to a single storage area causes a disk I/O bottleneck, you may need to create additional storage areas and partition some tables across multiple storage areas.

You can partition a table either horizontally or vertically. When you partition a table horizontally, you divide the rows of the table among storage areas according to data values in one or more columns. Then, a given storage area contains only those rows whose column values fall within the range that you specify. When you partition a table vertically, you divide the columns of the table among storage areas. Then, a given storage area contains only some of the columns of a table. You can divide any table both horizontally and vertically.

Often, after the logical design has been determined and the tables have been normalized, database designers encounter conflicts between the logical and physical design. Real-time access and storage criteria mandate that the logical design be compromised because of the physical design criteria. Vertical record partitioning can resolve this conflict. For example, if you know that you have heavy access to some of the columns in a table, but that access to other columns is occasional, you can partition a table vertically as well as horizontally.

If you want to divide the EMPLOYEES table into horizontal partitions using the EMPLOYEE_ID as the partitioning key and if the EMPLOYEE_ID, LAST_NAME, FIRST_NAME, MIDDLE_INITIAL, and STATUS_CODE columns of the EMPLOYEES table are used most frequently and the other columns less frequently, you could divide the EMPLOYEES table as shown in Figure 4–1.

Figure 4–1 Partitioning a Table Vertically and Horizontally

Without vertical record partitioning, you would have to implement this design by placing the frequently accessed columns in one table and the other columns in another table and joining the tables together, resulting in the following additional costs:

- Application maintenance

  Programs containing SQL statements are based on the logical design of the database. If the physical placement of the columns changes, you must modify all data manipulation statements referring to those tables or views. Vertical record partitioning maintains a single representation of the table. Oracle Rdb automatically partitions the table.

- Execution

  When applications refer to columns within partitions, Oracle Rdb accesses only the partitions containing those columns. If you do not use vertical record partitioning, Oracle Rdb retrieves all column data and materializes the data in memory before it can be accessed by an application.

- Storage

  With vertical record partitioning, you can specify some attributes, such as compression, for some columns in a table but not for other columns. In addition, it may be more cost effective to store specific column data on certain types of storage media. Vertical record partitioning allows the physical database designer to customize the storage on a column basis.

For more information about vertical record partitioning, see Section 4.6.4.

If you have some database administrative experience, you may know in advance that either heavy access to one large table or insufficient disk space justifies partitioning that table in a CREATE STORAGE MAP statement during your first attempt at the storage design.

Oracle Rdb strongly recommends that you define storage maps for all tables in a multifile database and include STORE clauses in all index definitions.

Before you determine a storage design that eliminates disk I/O bottlenecks, expect several storage design alterations. Be sure to back up your database before making any changes to its storage characteristics.

4.3 Assigning Tables and Indexes to Storage Areas

You assign tables to storage areas by defining storage maps with the CREATE STORAGE MAP statement.

You assign indexes to storage areas by defining indexes with the CREATE INDEX statement and using a STORE clause.

You can assign a table and index to one storage area, partition a table across multiple storage area files, or cluster records that are likely to be accessed together so that one I/O operation retrieves all those records.

Assume that personnel_db database contains the table DEPARTMENTS in addition to the storage areas defined in Example 4–1. Example 4–2 shows how to assign both the DEPARTMENTS table and an associated sorted index to one storage area, PERSONNEL_MISC.

Example 4–2 Assigning a Table and an Index to a Storage Area

```
SQL> -- Assign the table DEPARTMENTS to the storage area PERSONNEL_MISC.
SQL> CREATE STORAGE MAP DEPARTMENTS_MAP
cont>    FOR DEPARTMENTS
cont>    STORE IN PERSONNEL_MISC;
SQL> --
SQL> -- Create the index DEPARTMENTS_INDEX and assign it to the storage area
SQL> -- PERSONNEL_MISC.
SQL> CREATE UNIQUE INDEX DEPARTMENTS_INDEX
cont>    ON DEPARTMENTS (DEPARTMENT_CODE)
cont>    TYPE IS SORTED
cont>    STORE IN PERSONNEL_MISC;
```

Note

You should create the storage maps before you load data in a table. However, because the sorted index definition in Example 4–2 is not used to place rows in storage, you should create the indexes after you load data in a table. (Section 4.6.1 discusses using indexes to place rows in storage.)

If you do not explicitly specify where to store a table or sorted index, Oracle Rdb stores that table or index in the default storage area, if one exists. If no default storage area exists, Oracle Rdb stores the table or index in the RDB$SYSTEM storage area.

You can partition a table across multiple storage areas. Example 4–3 shows how to create a storage map to partition the JOB_HISTORY table horizontally across three storage areas: PERSONNEL_1, PERSONNEL_2, and PERSONNEL_3.

Example 4–3 Partitioning the JOB_HISTORY Table

```
SQL> CREATE STORAGE MAP JH_MAP
cont>    FOR JOB_HISTORY
cont>    STORE USING (EMPLOYEE_ID)
cont>          IN PERSONNEL_1 WITH LIMIT OF ('00399')
cont>          IN PERSONNEL_2 WITH LIMIT OF ('00699')
cont>    OTHERWISE IN PERSONNEL_3;
```

Rows in which the value of EMPLOYEE_ID is less than or equal to 00399 are stored in PERSONNEL_1. Rows in which the value of EMPLOYEE_ID is greater than 00399 but less than or equal to 00699 are stored in PERSONNEL_2. Rows in which the value of EMPLOYEE_ID is greater than 00699 are stored in PERSONNEL_3, as indicated by the OTHERWISE clause.

The OTHERWISE clause specifies the storage area that will be used as the overflow partition. An **overflow partition** is a storage area that holds any values that are higher than those specified in the WITH LIMIT OF clauses of a storage map or index definition. That is, an overflow partition holds those values that "overflow" the partitions that have specified limits.

If you define a hashed index on the EMPLOYEE_ID column of the JOB_HISTORY table, you should partition the hashed index to parallel the way you partition the table. (In this case, the storage map for the JOB_HISTORY table in Example 4–3 would have included the clause PLACEMENT VIA INDEX JH_HASH_INDEX.)

Example 4–4 shows how you partition a hashed index based on the EMPLOYEE_ID column across storage areas in a way that parallels how you partition the table.

Example 4–4 Partitioning a Hashed Index

```
SQL> CREATE INDEX JH_HASH_INDEX ON JOB_HISTORY
cont>    (EMPLOYEE_ID)
cont>     TYPE IS HASHED
cont>     STORE USING (EMPLOYEE_ID)
cont>           IN PERSONNEL_1 WITH LIMIT OF ('00399')
cont>           IN PERSONNEL_2 WITH LIMIT OF ('00699')
cont>     OTHERWISE IN PERSONNEL_3;
```

Index values that are less than or equal to 00399 are stored in PERSONNEL_1. Index values that are greater than 00399 but less than or equal to 00699 are stored in PERSONNEL_2. Index values that are greater than 00699 are stored in PERSONNEL_3, as indicated by the OTHERWISE clause.

If you know that you will not store values greater than a specific range, you can omit the OTHERWISE clause in storage maps and index definitions. Omitting the OTHERWISE clause lets you quickly add new partitions to the storage map and index without reorganizing the storage areas.

For example, if you know that no employee will have an EMPLOYEE_ID greater than 10000 within the next year, consider omitting the OTHERWISE clause. Construct the storage map and index as shown in Example 4–5.

Example 4–5 Creating Indexes and Storage Maps Without Overflow Areas

```
SQL> -- Create the EMP_HASH_INDEX index. PERSONNEL_3 holds values between
SQL> -- 00699 and 10000.
SQL> --
SQL> CREATE INDEX EMP_HASH_INDEX ON EMPLOYEES
cont>    (EMPLOYEE_ID)
cont>     TYPE IS HASHED
cont>     STORE USING (EMPLOYEE_ID)
cont>           IN PERSONNEL_1 WITH LIMIT OF ('00399')
cont>           IN PERSONNEL_2 WITH LIMIT OF ('00699')
cont>           IN PERSONNEL_3 WITH LIMIT OF ('10000');
SQL> --
SQL> -- Create the EMP_MAP storage map. PERSONNEL_3 holds the rows with
SQL> -- EMPLOYEE_ID values between 00699 and 10000.
SQL> --
SQL>  CREATE STORAGE MAP EMP_MAP
cont>     FOR EMPLOYEES
cont>     PLACEMENT VIA INDEX EMP_HASH_INDEX
```

(continued on next page)

Example 4–5 (Cont.) Creating Indexes and Storage Maps Without Overflow Areas

```
cont>     STORE USING (EMPLOYEE_ID)
cont>           IN PERSONNEL_1 WITH LIMIT OF ('00399')
cont>           IN PERSONNEL_2 WITH LIMIT OF ('00699')
cont>           IN PERSONNEL_3 WITH LIMIT OF ('10000');
```

If you attempt to insert values that are out of range of the storage map or index, you receive an error similar to the following:

```
%RDMS-E-EXCMAPLIMIT, exceeded limit on last partition in storage map for
EMPLOYEES
```

Your applications should include code to handle this type of error.

If you need to add a partition, you can easily alter the storage map or index definition. For information about adding partitions to indexes and storage maps, see Section 7.7.2 and Section 7.9.

If you know that you have heavy access to some of the columns in a table, but that the access to other columns is occasional, you can partition a table vertically. See Section 4.6.4 for information about vertical partitioning.

4.3.1 Specifying Storage Map Options

In addition to specifying the partitioning when you create a storage map, you can specify the options, including the following:

- Whether to enable or disable compression
- Whether rows are placed according to the index
- Threshold values for uniform format areas, which represent a fullness percentage on a data page, and control how the database system finds available (free) space in a storage area
- Whether or not the partitioning is updatable

Example 4–6 shows how to specify the threshold values for a uniform format storage area, as well as enabling compression and placing rows by index.

Example 4–6 Specifying Threshold Values for Uniform Areas

```
SQL> CREATE STORAGE MAP TEST_MAP
cont>        FOR TEST_TAB
cont>        ENABLE COMPRESSION
cont>        PLACEMENT VIA INDEX EMP_IND
cont>        STORE USING (EMPLOYEE_ID)
cont>          IN TEST_AREA1 (THRESHOLDS ARE (70,80,90))
cont>             WITH LIMIT OF ('00200');
cont>          IN TEST_AREA2 (THRESHOLDS ARE (70,80,90))
cont>             WITH LIMIT OF ('00400');
```

Unless you carefully calculate the threshold values, you should use the default values provided. For information on calculating the threshold values, see the *Oracle Rdb7 Guide to Database Performance and Tuning*.

Note that you specify thresholds for mixed format storage areas in the CREATE STORAGE AREA clause.

4.3.2 Enforcing Storage Map Partitioning

By default, when you create a storage map, you can update the values in the partitioning keys (columns on which the partitioning is based). If you use the STORE USING clause, Oracle Rdb does not move the row to a different storage area even if the new value of the partitioning key is not within the limits of the storage area. For example, assume that the partitioning key is the employee's last name and an employee's last name changes from Jones to Smith. If the storage map is defined as shown in the following example, you can update the last name, but the row remains in the area NAME_MID.

```
SQL> CREATE STORAGE MAP EMP_MAP
cont>    FOR EMPLOYEES
cont>    STORE USING (LAST_NAME)
cont>          IN NAME_LOW WITH LIMIT OF ('I')
cont>          IN NAME_MID WITH LIMIT OF ('P')
cont>          OTHERWISE IN NAME_HIGH;
```

As a result, when you retrieve data, Oracle Rdb must consider all three storage areas in retrieving a row. Because the value of the partitioning key may have been updated, Oracle Rdb cannot be assured that the row is stored in the storage area based on the partitioning criteria. This default behavior is the same as specifying the PARTITIONING IS UPDATABLE clause of the CREATE or ALTER STORAGE MAP statement.

In many database designs, the values in the partitioning keys do not change. For example, the storage map JH_MAP, shown in Example 4–3, bases the partitioning on the EMPLOYEE_ID column. The identification number of an employee rarely, if ever, changes during the employee's tenure at a company.

If you know that the values of the partitioning key will not or should not change, you can improve the retrieval of data by using the PARTITIONING IS NOT UPDATABLE clause of the CREATE STORAGE MAP statement. The PARTITIONING IS NOT UPDATABLE clause guarantees that users cannot change the value of the partitioning key and that Oracle Rdb always stores the row in the storage area based on the partitioning criteria. As a result, Oracle Rdb can more quickly retrieve data because it can use the partitioning criteria when optimizing the query.

Example 4–7 shows how to create the EMPL_MAP storage map, which does not allow partitioning keys to be updated.

Example 4–7 Enforcing Storage Map Partitioning

```
SQL> CREATE STORAGE MAP EMPL_MAP
cont>   FOR EMPLOYEES
cont>   PARTITIONING IS NOT UPDATABLE
cont>   STORE USING (EMPLOYEE_ID)
cont>         IN EMP_LOW WITH LIMIT OF ('00199')
cont>         IN EMP_MID WITH LIMIT OF ('00399')
cont>         OTHERWISE IN EMP_HIGH;
SQL> --
SQL> -- You cannot modify the values in the EMPLOYEE_ID column.
SQL> UPDATE EMPLOYEES
cont>       SET EMPLOYEE_ID = '50198'
cont>       WHERE EMPLOYEE_ID = '00198';
%RDB-E-READ_ONLY_FIELD, attempt to update the read-only field EMPLOYEE_ID
```

Although you cannot modify the values in the EMPLOYEE_ID column, you can insert new rows. To update columns that are partitioning keys in a NOT UPDATABLE storage map, you must delete the rows, then reinsert the rows to ensure that they are placed in the correct location.

You can use the PARTITIONING IS clause only when you use the STORE USING clause.

You can change a storage map from PARTITIONING IS NOT UPDATABLE to UPDATABLE, but you cannot change it from UPDATABLE to NOT UPDATABLE because the data may no longer be strictly partitioned according to the criteria.

4.4 Choosing Uniform or Mixed Page Format

When you define a storage area, you specify either a uniform or mixed page format based on what data you plan to store on a page in the storage area and how you expect the data to be accessed.

Note

Page divisions in the RDB$SYSTEM storage area always adhere to the default on-disk structure (PAGE FORMAT IS UNIFORM).

4.4.1 Advantages of Uniform Page Format

A storage area defined as a uniform page format can only contain data from one specific table or one sorted index. However, if the tables or indexes are stored in the RDB$SYSTEM storage area, a page in a uniform storage area can contain data from one or more tables or index nodes from one or more sorted indexes defined on the same table.

Oracle Rdb maintains separate logical areas for each table's data and each table's indexes. Because indexes defined on the same table are assigned the same logical area ID number, each can reside on the same page. Oracle Rdb assigns an index defined on another table to another logical area identification (ID) number. That is, only one logical area ID is allowed per page.

A uniform page format is more appropriate than a mixed page format in the following situations:

- When storing of one or more tables that are to be accessed in a variety of ways. Store tables in uniform page format storage areas when you do not need to optimize for a particular kind of query (such as an exact match query)
- When you need the best possible performance for sequential scans of a table
- When you use sorted indexes

  A storage area defined as a uniform page format is most appropriate for storing sorted indexes. However, for optimal retrieval performance, two sorted indexes defined on the same table should be stored in different storage areas. This ensures that the index nodes for each index defined on the same table are not stored on the same page within the same storage area because both indexes have the same logical area ID number.

When you specify the uniform page format for a storage area, you can specify the space area management (SPAM) thresholds for the logical areas. You specify the thresholds for each logical area in the CREATE STORAGE MAP or CREATE INDEX statements. See Section 4.3.1 for an example of specifying threshold values in a CREATE STORAGE MAP statement.

When you specify the uniform page format, you implicitly specify that the database system divide or partition a storage area into sets of adjacent pages devoted to specific logical areas, which either store rows from a specific table or nodes from one or more indexes on that table by its logical area ID. These sets of adjacent pages are called **clumps**. Oracle Rdb calculates a clump size that provides the best performance given the buffer size value that you set when you created the database. Oracle Rdb use the following formula:

```
Default clump size ([3 pages]) = buffer size [6 blocks]/ page size [2 blocks]
```

Depending on what logical area a clump is dedicated to storing (a table's rows or nodes from one or more sorted indexes on a table), its pages contain rows from only that table or nodes from only those indexes defined on that table. Note that a table or one or more indexes defined on the same table may require multiple clumps and that these clumps are not necessarily adjacent to one another in the storage area. Nonetheless, clump storage (by logical area ID) helps conserve disk I/O operations. This is particularly true in the following cases:

- When the database system must scan an entire table or return a large amount of table data in response to a user's request
- When the database system navigates a sorted index

See the *Oracle Rdb7 Guide to Database Performance and Tuning* for more information on uniform page formats.

4.4.2 Advantages of Mixed Page Format

A storage area defined as a mixed page format can contain data from one or more logical areas (table rows, sorted index nodes, or hash index structures). Each mixed format page can contain data from more than one table, nodes from more than one sorted index, or hash structures from more than one hash index; that is, more than one logical area ID is allowed per page.

In a storage area with a mixed page format, the database system does not automatically dedicate a clump of adjacent pages to storage of the rows of only one table or only one index. Each page in an area with a mixed page format can store hash buckets (structures that contain a list of dbkeys to the logical area, database page, and line entry on the page that matches the value of the index key) if any hashed indexes are assigned to the storage area; sorted index

nodes, if any sorted indexes are assigned to the storage area; and rows from any table assigned to the area.

If you do not carefully control what is stored in a mixed page format storage area (hash buckets, sorted index nodes, rows), the preceding elements may be intermixed on pages and dispersed among pages in a way that can degrade the performance of all queries. On the other hand, if you reserve a storage area with a mixed page format for storage of only one table or a few tables whose rows are accessed together, you can manipulate data storage to achieve the highest possible performance for large tables or high-priority queries. In this case, you must correctly calculate values for initial file size, page size, and other memory and space management parameters to suit the data that you are storing in the storage area.

For example, you have the following options for a storage area only if you specify PAGE FORMAT IS MIXED when you define the storage area:

- You can store hashed indexes only in storage areas with a mixed page format.

  Hashed indexes can support direct retrieval of rows with fewer disk I/O operations than sorted indexes require. In addition, data retrieval using a hashed index is subject to fewer lock contention problems than data retrieval using a sorted index. If your database contains very large tables whose rows are frequently selected by column values that exactly match a specified key, you may want to create hashed indexes for those tables. In this case, you would store only the table and its hashed index in the storage area.

  For example, if the EMPLOYEES table of the mf_personnel database was very large, and if most users accessed rows by entering specific EMPLOYEE_ID values, you might want to create a hashed index on the EMPLOYEE_ID column of the EMPLOYEES table. You would store both the EMPLOYEES table and its hashed index in the same mixed page format storage area.

  Section 4.6.2 contains details about defining and using hashed indexes.

- You can store related rows from different tables on the same page only in storage areas with a mixed page format.

  Clustering rows from different tables on the same storage area page is appropriate when:

  - Your database must support a particular transaction-processing application with very high performance requirements
  - The application includes a join operation, updates rows in some tables depending on which rows are retrieved in other tables, or both

For example, if users entered EMPLOYEE_ID values to retrieve both EMPLOYEES and related JOB_HISTORY rows together or to retrieve EMPLOYEES rows and then update JOB_HISTORY rows, you could cluster related EMPLOYEES and JOB_HISTORY rows on the same or adjacent pages. In this case, you would define hashed indexes on the EMPLOYEE_ID columns of both tables and store both indexes and tables in the same mixed page format storage area.

Section 4.6.3 discusses clustering related rows in storage.

You can control how the database system finds available (free) space in a storage area by manipulating the space area management (SPAM) pages. The SPAM pages govern how the database system finds free space on pages when storing a row. Typically, you manipulate SPAM pages in conjunction with hashing and row clustering strategies. You can specify the space area management (SPAM) thresholds for each mixed format area by using the THRESHOLDS clause of the CREATE STORAGE AREA statements.

See the *Oracle Rdb7 Guide to Database Performance and Tuning* for information on calculating the threshold values.

Note

Customizing INTERVAL and THRESHOLDS values for the storage area in which you store lists (segmented strings) can result in significant performance improvement. Oracle Rdb recommends that you place lists in their own area, using mixed page format so that you can customize the values to use the space in the most efficient way. For more information, see the *Oracle Rdb7 Guide to Database Performance and Tuning* and the *Oracle Rdb7 SQL Reference Manual*.

For more information on indexing and storage options, see the following sections:

- Section 4.2 provides general guidelines about using indexing and storage options to eliminate I/O bottlenecks at disk drives while maintaining the flexibility of the database to support a wide range of applications.
- Section 4.6 explains optimizing for specific queries (exact match retrieval, range retrieval, and joins) and for the LIST OF VARBYTE data type.

4.5 Choosing Read/Write, Read-Only, or Write-Once Storage Areas

When you create storage areas, you usually create them as read/write storage areas. However, Oracle Rdb lets you both create write-once storage areas and change read/write and write-once storage areas to read-only.

If you have stable data you do not expect to change, you can move that data from read/write storage areas to read-only storage areas. Section 7.6.8 explains how to change a read/write storage area to a read-only storage area.

You can create write-once storage areas on write-once, read-many (WORM) optical devices. WORM optical devices are less expensive than magnetic disks for storing large amounts of data that will not be updated.

You cannot store data other than lists (segmented strings) in write-once storage areas. In addition, note the following points:

- You cannot create the default list storage area as a write-once area.
- You must specify mixed page format for write-once storage areas.
- You should specify that the snapshot file is created on a read/write device, not a WORM device.
- You should not allocate a large number of pages to the snapshot file.

Example 4–8 shows how to create a default list storage area on a read/write disk and a write-once storage area on a WORM optical device.

Example 4–8 Creating Write-Once Storage Areas on WORM Optical Devices

```
SQL>  CREATE DATABASE FILENAME db_disk:test
cont>     LIST STORAGE AREA IS LIST_DEF_STOR
cont> CREATE STORAGE AREA LIST_DEF_STOR  FILENAME db_disk:list_def_stor
cont>     PAGE FORMAT IS MIXED
cont>     SNAPSHOT FILENAME db_disk:list_def_stor
cont>     SNAPSHOT ALLOCATION IS 3
cont> CREATE STORAGE AREA WORM_STOR FILENAME oda0:[database]worm_stor
cont>     PAGE FORMAT IS MIXED
cont>     SNAPSHOT FILENAME db_disk:worm_stor
cont>     SNAPSHOT ALLOCATION IS 3
cont>     WRITE ONCE;
SQL> --
```

(continued on next page)

Example 4–8 (Cont.) Creating Write-Once Storage Areas on WORM Optical Devices

```
SQL> SHOW STORAGE AREA
Storage Areas in database with filename db_disk:test
     RDB$SYSTEM
     LIST_DEF_STOR                  List storage area.
     WORM_STOR
SQL>
```

Assume you have a table that stores resumes and contains two columns, EMPLOYEE_ID and RESUME. The RESUME column has a data type of LIST OF BYTE VARYING. Because write-once storage areas can contain only list data, create one storage map for the RESUME column. Create another storage map for the EMPLOYEE_ID column. Example 4–9 shows the table definition and how to create the storage maps.

Example 4–9 Creating Storage Maps for Write-Once Storage Areas

```
SQL> CREATE TABLE RESUMES
cont>    (EMPLOYEE_ID CHAR(5),
cont>     RESUME      LIST OF BYTE VARYING);
SQL> --
SQL> -- Create a storage map for the EMPLOYEE_ID column
SQL> --
SQL> CREATE STORAGE MAP RESUME_ID_MAP
cont>    FOR RESUMES
cont>    STORE IN RDB$SYSTEM;
SQL> --
SQL> -- Create a storage map for the RESUME column.
SQL> --
SQL> CREATE STORAGE MAP RESUME_LIST_MAP
cont>    STORE LISTS IN WORM_STOR FOR (RESUMES);
```

By definition, a WORM device lets you write once to the media. That is, you cannot write to the same blocks on the media more than once. To reduce wasted space on a disk, specify the page size as a multiple of the sector size.

For more information about write-once storage areas, see the following:

- Section 4.6.5 for an explanation of how to optimize your database for the LIST OF VARBYTE data type
- Section 7.6.5 for an explanation of how to modify storage areas that are read/write into storage areas that are write-once

- The *Oracle Rdb7 Guide to Database Maintenance* for information about how to maintain data on WORM optical devices

4.6 Achieving Optimal Performance for Queries and Update Operations

You can specify a storage design to support the following kinds of queries and updates:

- Retrieval of rows whose column values are in a specified range (See Section 4.6.1.)
- Retrieval of rows whose column values exactly match specified values (See Section 4.6.2.)
- Retrieval of rows using a join operation

  If your application updates rows in some tables based on retrieval of related rows in other tables, you design storage as if you were optimizing storage for a join operation. (See Section 4.6.3.)
- Retrieval of some columns in a table (See Section 4.6.4.)
- Retrieval of LIST OF VARBYTE data (See Section 4.6.5.)

You specify a storage design to optimize the most important transactions. Deciding on an index structure is part of your storage design.

4.6.1 Achieving Optimal Performance for Range Retrieval

Example 4–10 shows how you can optimize performance for a range retrieval query by first creating a sorted index before you store rows and then placing the rows in a storage area.

Example 4–10 Optimizing Performance for Range Retrieval Queries

```
CREATE DATABASE...
   .
   .
   .
CREATE STORAGE AREA PERSONNEL_1
    FILENAME persdisk1:[mfp]pers1
    PAGE FORMAT IS UNIFORM
    PAGE SIZE IS...
    ALLOCATION IS...
   SNAPSHOT FILENAME persdisk1:[mfp]pers1
```

(continued on next page)

Example 4–10 (Cont.) Optimizing Performance for Range Retrieval Queries

```
   .
   .
   .
CREATE STORAGE AREA PERSONNEL_MISC
    FILENAME persdisk0:[mfp]pers0
    PAGE FORMAT IS UNIFORM
    SNAPSHOT FILENAME persdisk0:[mfp]pers0
   .
   .
   .
CREATE UNIQUE INDEX EMP_SORT_INDEX ON EMPLOYEES
    (EMPLOYEE_ID)
     TYPE IS SORTED
     STORE IN PERSONNEL_MISC
   .
   .
   .
CREATE STORAGE MAP EMP_MAP FOR EMPLOYEES
    STORE
    IN PERSONNEL_1
   .
   .
   .
```

The definitions in Example 4–10 increase the chances that rows associated with the same or adjacent column values in the index are stored near one another. Storing rows on pages in index sort order reduces the I/O operations required to find rows that contain a specified range of indexed values.

Remember the following guidelines if you select this kind of storage strategy:

- Pre-sort your data file so that rows are in index sort order *before* you use the file to load a table.

  Loading unsorted data into the table decreases the chances that rows are placed on pages in index sort order.

- Store the table in a storage area with uniform page format and do not store indexes or other tables in that storage area.

  Make sure that your initial allocation for the storage area (ALLOCATION clause value) is large enough to accommodate all of the data initially loaded into the table.

- Store the sorted index in another storage area, one with uniform page format.

The one sorted index is likely to provide performance improvement for exact match queries as well as range retrieval queries.

4.6.2 Achieving Optimal Performance for Exact Match Retrieval

A hashed index on the appropriate column or columns of a table provides the most efficient retrieval for exact match queries. If you create a hashed index on the column or columns by which users most often retrieve rows, remember these guidelines:

- The hashed index must be stored in a storage area for which you specify a mixed page format.

  Only storage areas with mixed page format support hash buckets.

- Although rows do not have to be stored on the same page (or even in the same storage area) as the hash buckets that contain the row's dbkeys, you achieve the best performance from a hashed index if you cluster rows with an associated hash bucket on the same page of the same storage area.

  In other words, you usually assign both a table and its hashed index to the same storage area and specify the hashed index in a PLACEMENT VIA INDEX clause when you define a storage map for the table. Assuming that you set PAGE SIZE, ALLOCATION, and other storage area parameters correctly, you may need only one I/O operation to retrieve both the hash bucket and rows.

- Unless you are explicitly trying to cluster related rows from multiple tables on a page, do not use the storage area for storing tables that are not associated with the hashed index.

  Do not assign miscellaneous tables and sorted indexes to a storage area that contains rows retrieved by a hashed index. Unrelated indexes and tables can use up space on pages on which the hashing algorithm expects either to place or to find hash buckets and related rows. To be effective, hashed indexes require sufficient space for hash buckets and rows on specific pages of the storage area.

- The value set for a storage area by the PAGE SIZE clause is critical to the effectiveness of a hashed index.

  The page size should accommodate the hash bucket and associated rows. If the page size is too small, rows spill over onto pages that hashing calculations associate with other hash buckets. As a result, additional I/O operations are required for row retrieval because the accuracy of the hashing calculation for finding the correct page is reduced.

You can use hashed indexes to cluster, on the same page, related rows from one or multiple tables, hash buckets and associated rows, or both. In this case, page size calculation must take into account the combined space needed for all the elements being clustered on the page.

Page size calculations are easiest to manage when there are no duplicate data values for the column on which the hashed index is based. For example, calculating page size when a hashed index is based on the EMPLOYEE_ID column of the EMPLOYEES table is easier than calculating page size when a hashed index is based on the EMPLOYEE_ID column of the JOB_HISTORY table.

If duplicate column values are allowed in the column on which you base a hashed index, page size calculations are still relatively easy if the expected number of row duplicates does not vary much from one column value to another. When the number of row duplicates is nearly the same from one column value to the next, you can plan for the maximum number of duplicates without wasting too much disk space. However, if the number of duplicates is highly variable from one column value to the next, you may not be able to calculate a page size that works efficiently with your hashed index.

- The value for a storage area file set by the initial ALLOCATION clause is critical to the performance of a hashed index.

  The ALLOCATION clause for the storage area that stores the hashed index determines the number of pages included in the hashing algorithm. To store or retrieve a row, the hashing algorithm uses the data value in the column on which the index is based, along with the initial number of pages in the storage area, to calculate a page location.

  The hashing algorithm does not change when the storage area file is extended. New pages from the file extents are not included within the range of target pages the hashing algorithm uses for calculating where rows are stored. After a file extends, hashed index performance may degrade because the database system may need to search more than one hash bucket (a chain of hash buckets) to find data on pages in the extended portion of the file.

  Therefore, be sure to set the ALLOCATION clause value large enough to accommodate all the data you plan to load into the storage area. Leave extra space in the file to anticipate its growth until you plan to unload and load (export/import) data from tables in your database again. You can use the RMU Dump Header command to determine if a storage area file has been extended. Extensions to the file indicate that you need to specify a larger allocation value for the storage area when you reload the table.

Use the RMU Analyze command to analyze the effects on row placement of your storage area page size and allocation parameters.

- Define the hashed index *before* you load any rows into the table associated with the index.

  Existing rows are not moved around in storage to fit the needs of an algorithm established by a hashed index that you define after you load a table. If rows already exist in a table, their dbkeys are integrated into the appropriate bucket of a new hashed index. You may get some performance advantage for exact match queries by defining a hashed rather than a sorted index after a table is loaded, but the performance improvement is not optimal.

Example 4–11 illustrates the most common way a hashed index is created. Note that you create the hashed index before you load the table and that you store the index in the same storage area as the table.

Example 4–11 Creating a Hashed Index

```
SQL> CREATE DATABASE
     .
     .
     .
SQL> CREATE STORAGE AREA PERSONNEL_1
cont>    FILENAME persdisk1:[mfp]pers1
cont>    PAGE FORMAT IS MIXED     -- Required for storing hashed index
cont>    PAGE SIZE IS...          -- Calculate for acceptable performance
cont>    ALLOCATION IS...         -- Calculate for acceptable performance
cont>    SNAPSHOT FILENAME persdisk1:[mfp]pers1
     .
     .
     .
cont> CREATE UNIQUE INDEX EMP_HASH_INDEX ON EMPLOYEES
cont>     (EMPLOYEE_ID)
cont>     TYPE IS HASHED
cont>     STORE IN PERSONNEL_1
     .
     .
     .
```

(continued on next page)

Example 4–11 (Cont.) Creating a Hashed Index

```
cont> CREATE STORAGE MAP EMP_MAP FOR EMPLOYEES
cont>     STORE
cont>     IN PERSONNEL_1       -- Same area as hashed index (recommended)
cont>     PLACEMENT VIA INDEX EMP_HASH_INDEX
cont>                            -- Cluster row with hash bucket (recommended)
    .
    .
    .
```

Placing rows in storage with a hashed index should distribute the rows across the entire range of pages in a storage area. Furthermore, because of the way the hashing algorithm works, rows that store adjacent column values are not adjacent in storage.

When you place rows in a storage area using a hashed index, you improve the performance of exact match retrieval on the indexed column at the expense of sequential queries. After you load the table, you may need to define one or more sorted indexes to attain acceptable performance for queries based on other columns.

For detailed information about calculating and testing page size and file allocation values, see Section 4.8 through Section 4.8.7. Section 4.9 through Section 4.9.6 provide detailed information about placement and clustering strategies using hashed indexes.

4.6.3 Achieving Optimal Performance for Join Operations or Update of Related Rows

Before reading this section, you should be familiar with information in Section 4.6.2. This section continues discussion of issues that were introduced in Section 4.6.2.

You may need to achieve the best possible performance for a query that joins two or more tables or for any other operation that retrieves related rows from different tables. In this case, you can reduce the number of disk I/O operations needed for data retrieval, update, or both by clustering related rows from the different tables on the same or adjacent pages of a storage area. To cluster rows from different tables:

- For each table, define a hashed index on the column (or set of columns) that the query uses to match rows
- For each table, define a storage map that specifies the index in a PLACEMENT VIA INDEX clause

Storage maps for all tables specify the same mixed page format storage area or, if you partition the tables across multiple storage areas, use a STORE USING clause that specifies the same set of mixed page format storage areas.

You must define the same kind of index on the same column or columns for each table. For example, suppose you want data storage to best support a query that uses the EMPLOYEE_ID column to join rows from the EMPLOYEES table with related rows from the JOB_HISTORY table. In this case, you might define a hashed index with duplicates not allowed on the EMPLOYEE_ID column of the EMPLOYEES table and another hashed index on the EMPLOYEE_ID column of the JOB_HISTORY table.

Note

Attempting to both optimize for range retrieval *and* cluster rows from different tables is a difficult storage strategy to manage successfully. When tables are loaded completely and independently, placement of rows from different tables using sorted indexes does not result in clustering rows from different tables on the same or adjacent pages. Assuming that the data being loaded is already sorted, rows from a particular table may be stored in index sort order, but each table's data is loaded into a different range of pages in the storage area.

You may achieve satisfactory clustering results by careful manipulation of the load operations. For example, if you wanted to cluster related EMPLOYEES and JOB_HISTORY rows, and arrange rows in index sort order, your program would have to load related rows into different tables at the same time. In other words, your program would have to load the EMPLOYEES row with EMPLOYEE_ID '00162', JOB_HISTORY rows with EMPLOYEE_ID '00162', the EMPLOYEES row with EMPLOYEE_ID '00163', JOB_HISTORY rows with EMPLOYEE_ID '00163', and so forth.

Hashed indexes provide more control over row placement than sorted indexes do and therefore are suited to clustering rows from different tables. In general, think of row clustering as a storage strategy that is associated with hashed indexes and that is incompatible with range retrieval optimization.

The storage areas in which you cluster rows from different tables must be mixed page format storage areas. The storage areas should store only the tables whose rows are being clustered and the hash buckets for all the indexes.

Correct file allocation and page size calculations are critical for storage areas in which you cluster rows from different tables. Page size needs to accommodate, among other things:

- Hashed index rows for multiple tables
- The size of rows from different tables
- The expected number of row duplicates, if any, from each table

Example 4–12 is an excerpt from a CREATE DATABASE statement that clusters related EMPLOYEES and JOB_HISTORY rows. Because high-volume, transaction-processing applications either work with very large tables, require an I/O rate that cannot be attained using one disk drive, or both, the example partitions the two tables over three disk drives.

Example 4–12 Clustering Related Rows from Two Tables

```
   .
   .
   .
CREATE STORAGE AREA PERSONNEL_1
    FILENAME persdisk1:[mfp]pers1
    PAGE FORMAT IS MIXED
    ALLOCATION IS...
    PAGE SIZE IS...
    SNAPSHOT FILENAME persdisk1:[mfp]pers1

CREATE STORAGE AREA PERSONNEL_2
    FILENAME persdisk2:[mfp]pers2
    PAGE FORMAT IS MIXED
    ALLOCATION IS...
    PAGE SIZE IS...
    SNAPSHOT FILENAME persdisk2:[mfp]pers2

CREATE STORAGE AREA PERSONNEL_3
    FILENAME persdisk3:[mfp]pers3
    PAGE FORMAT IS MIXED
    ALLOCATION IS...
    PAGE SIZE IS...
    SNAPSHOT FILENAME persdisk3:[mfp]pers3

CREATE TABLE EMPLOYEES...

CREATE TABLE JOB_HISTORY...
```

(continued on next page)

Example 4–12 (Cont.) Clustering Related Rows from Two Tables

```
CREATE UNIQUE INDEX EMP_ID_HASH ON EMPLOYEES
    (EMPLOYEE_ID)
    TYPE IS HASHED
    STORE USING (EMPLOYEE_ID)
          IN PERSONNEL_1 WITH LIMIT OF ('00399')
          IN PERSONNEL_2 WITH LIMIT OF ('00699')
    OTHERWISE IN PERSONNEL_3;

CREATE STORAGE MAP EMP_MAP FOR EMPLOYEES
    STORE USING (EMPLOYEE_ID)
          IN PERSONNEL_1 WITH LIMIT OF ('00399')
          IN PERSONNEL_2 WITH LIMIT OF ('00699')
    OTHERWISE IN PERSONNEL_3
    PLACEMENT VIA INDEX EMP_ID_HASH;

CREATE UNIQUE INDEX JH_ID_HASH ON JOB_HISTORY
    (EMPLOYEE_ID)
    TYPE IS HASHED
    STORE USING (EMPLOYEE_ID)
          IN PERSONNEL_1 WITH LIMIT OF ('00399')
          IN PERSONNEL_2 WITH LIMIT OF ('00699')
    OTHERWISE IN PERSONNEL_3;

CREATE STORAGE MAP JH_MAP FOR JOB_HISTORY
    STORE USING (EMPLOYEE_ID)
          IN PERSONNEL_1 WITH LIMIT OF ('00399')
          IN PERSONNEL_2 WITH LIMIT OF ('00699')
    OTHERWISE IN PERSONNEL_3
    PLACEMENT VIA INDEX JH_ID_HASH;
```

When you cluster rows from different tables on the same page, you should also direct how the database system finds space in a storage area for row storage.

Section 4.8 provides more information about database and storage area parameters and storage map clauses that are important to consider for databases that either require much disk space, have very high performance requirements, or both. Section 4.3.1 shows how to specify threshold values for logical areas.

You may want to use global buffers, instead of local buffers. When you use global buffers, Oracle Rdb maintains one buffer pool on each node for each database. When you use local buffers, Oracle Rdb maintains a buffer pool for each process. To specify global buffers, use the GLOBAL BUFFERS ARE ENABLED clause.

The CREATE DATABASE statement includes clauses, such as ALLOCATION, that you can use to specify storage area characteristics. However, in a multifile database, you should explicitly control all storage area characteristics in each CREATE STORAGE AREA definition. Otherwise, it is easy to apply a default that is inappropriate for the intended use of the storage area.

4.6.4 Achieving Optimal Performance for Retrieving Some Columns in a Table

If access to some of the columns in a table is heavy, but the access to other columns is occasional or rare, you can partition a table vertically. For example, in the EMPLOYEES table in sample mf_personnel database, if the access to the EMPLOYEE_ID, LAST_NAME, FIRST_NAME, MIDDLE_INITIAL, and STATUS_CODE is frequent, but the access to the many of the other columns in the table is occasional and access to the SEX and BIRTHDAY columns are rare, consider partitioning the table vertically.

When you partition a table vertically, you divide the columns of the table among storage areas. As a result, a given storage area contains only some of the columns of a table. When you access data from tables that use vertical partitioning, Oracle Rdb always accesses the first partition. If the columns are in the second partition, Oracle Rdb accesses both the first and second partition. Therefore, you should place the most frequently accessed columns in the first partition.

Example 4–13 shows how to create a storage map to partition the EMPLOYEES table vertically. Assume that the ACTIVE_AREA, INACTIVE_AREA, and OTHER_AREA storage areas have been defined and placed on different disks by CREATE STORAGE AREA clauses in the database definition.

Example 4–13 Partitioning the EMPLOYEES Table Vertically

```
SQL> CREATE STORAGE MAP EMPLOYEES_1_MAP
cont>    FOR EMPLOYEES
cont>    ENABLE COMPRESSION
```

(continued on next page)

Example 4–13 (Cont.) Partitioning the EMPLOYEES Table Vertically

```
cont>    STORE COLUMNS (EMPLOYEE_ID, LAST_NAME, FIRST_NAME,
cont>                   MIDDLE_INITIAL, STATUS_CODE)
cont>                   DISABLE COMPRESSION
cont>                   IN ACTIVE_AREA
cont>    STORE COLUMNS (ADDRESS_DATA_1, ADDRESS_DATA_2, CITY,
cont>                   STATE, POSTAL_CODE)
cont>                   IN INACTIVE_AREA
cont>    OTHERWISE IN OTHER_AREA;
```

In Example 4–13, Oracle Rdb places the columns that are not named in the OTHER_AREA storage area.

To improve performance, consider disabling compression for those storage areas that are accessed frequently; to conserve disk space, enable compression for those areas that are not accessed frequently. Example 4–13 shows how to enable compression for the entire storage map and then disable it for one area, ACTIVE_AREA.

You can divide any table both horizontally and vertically. Example 4–14 shows how to divide the EMPLOYEES table so that the frequently used columns are divided into three storage areas, the less frequently used columns are divided into three more storage areas, and the rarely used columns are placed into one storage area.

Example 4–14 Partitioning the EMPLOYEES Table Vertically and Horizontally

```
SQL>  CREATE STORAGE MAP EMPLOYEES_1_MAP2
cont>        FOR EMP2
cont>        STORE COLUMNS (EMPLOYEE_ID, LAST_NAME, FIRST_NAME,
cont>                       MIDDLE_INITIAL, STATUS_CODE)
cont>              USING (EMPLOYEE_ID)
cont>              IN ACTIVE_AREA_A WITH LIMIT OF ('00399')
cont>              IN ACTIVE_AREA_B WITH LIMIT OF ('00699')
cont>              OTHERWISE IN ACTIVE_AREA_C
cont>        STORE COLUMNS (ADDRESS_DATA_1, ADDRESS_DATA_2, CITY,
cont>                      STATE, POSTAL_CODE)
cont>              USING (EMPLOYEE_ID)
```

(continued on next page)

Example 4–14 (Cont.) Partitioning the EMPLOYEES Table Vertically and Horizontally

```
cont>              IN INACTIVE_AREA_A WITH LIMIT OF ('00399')
cont>              IN INACTIVE_AREA_B WITH LIMIT OF ('00699')
cont>              OTHERWISE IN INACTIVE_AREA_C
cont>        OTHERWISE IN OTHER_AREA;
```

4.6.5 Achieving Optimal Performance for List Data

Data stored in lists can vary in size from several bytes to many megabytes. Generally, it is good practice to isolate list data in a separate storage area or areas. For storing large amounts of list data, store the lists in mixed page format areas so that you can adjust the space management (SPAM) thresholds and interval values for better performance and control over the placement of the lists.

For large lists, defining the RDMS$BIND_SEGMENTED_STRING_BUFFER logical name or RDB_BIND_SEGMENTED_STRING_BUFFER configuration parameter with a value large enough to hold the entire list may improve the performance of storing the list. However, SQL does not require you to define the value for the logical name or configuration parameter. Before the list is brought into the buffer, SQL knows the column with which list is associated and the table in which it is stored. See the *Oracle Rdb7 Guide to Database Performance and Tuning* for more information.

4.6.5.1 Storing List Data in Isolation

You can place storage areas containing list data on different disks if space requirements or performance considerations demand such separation. If list data is mixed with other data in the same storage area, Oracle Rdb distributes it randomly, causing the database pages to be cluttered, and possibly slowing retrieval.

The following excerpt from the SQL command procedure for the mf_personnel sample database demonstrates one way to separate list data from the other data in a table:

```
CREATE DATABASE FILENAME mf_personnel
-- To hold EMPLOYEE_ID data from RESUMES table.
   CREATE STORAGE AREA RESUMES

-- To hold resume (list) data from RESUMES table.
   CREATE STORAGE AREA RESUME_LISTS
```

```
   .
   .
   .
   CREATE DOMAIN ID_DOM  CHAR(5);
--
   CREATE DOMAIN RESUME_DOM  LIST OF BYTE VARYING;

  CREATE TABLE RESUMES
         (EMPLOYEE_ID ID_DOM
          REFERENCES EMPLOYEES (EMPLOYEE_ID),
          RESUME RESUME_DOM);

   CREATE STORAGE MAP LISTS_MAP
          STORE LISTS IN RESUME_LISTS FOR (RESUMES)
                      IN RDB$SYSTEM;

   CREATE STORAGE MAP RESUMES_MAP FOR RESUMES
          STORE IN RESUMES;
```

Oracle Rdb stores resumes from the RESUMES table in the RESUME_LISTS area and Employee IDs from the RESUMES table in the RESUMES area.

You can create only one storage map for lists within each database. If you try to add a list storage map, you get the following error message:

```
%RDB-E-NO_META_UPDATE, metadata update failed
-RDMS-E-SEGALRDYMP, there already exists a map for segmented strings
```

You can store lists across several storage areas, either randomly or sequentially. In addition, you can specify that some columns with the LIST data type are stored in only one storage area, but that all other columns with the LIST data type are stored randomly.

For example, assume that the RESUMES table has three columns with the LIST data type—RESUME, EMPL_APPLICATION, and INTERVIEW_NOTES. Using the storage map in the following example, SQL stores the RESUME column randomly across the LISTS1, LISTS2, and LISTS3 storage areas, but it stores the other LIST columns only in the LISTS1 storage area.

```
SQL> CREATE STORAGE MAP LISTS_MAP
cont>    STORE LISTS
cont> -- Store all LIST columns that are not explicitly specified in LISTS1.
cont> -- Also, store some of the RESUME columns in LISTS1.
cont>          IN LISTS1 FOR (RESUMES, RESUMES.RESUME)
cont>          IN LISTS2 FOR (RESUMES.RESUME)
cont>          IN LISTS3 FOR (RESUMES.RESUME)
cont>          FILL RANDOMLY
cont>          IN RDB$SYSTEM;
```

For more information about random and sequential storage, see Section 4.6.5.2.

You can store lists from different tables in the same storage area. This still effectively isolates list data. The following example stores the list data from different tables in the LISTS1 storage area:

```
SQL> CREATE STORAGE MAP LISTS_MAP  -- to direct the list data to area LISTS
cont>    STORE LISTS IN LIST1 FOR (TABLE1, TABLE2, TABLE3)
cont>                IN RDB$SYSTEM;
```

Alternatively, you can store lists from each table in unique areas. The following example stores lists from the TABLE1 table in the LISTS1 storage area and lists from the TABLE2 table in the LISTS2 storage area:

```
SQL> CREATE STORAGE MAP LISTS_MAP
cont>    STORE LISTS IN LIST1 FOR (TABLE1)
cont>                IN LIST2 FOR (TABLE2)
cont>                IN RDB$SYSTEM;
```

You can also specify that different columns from the same table go into different areas:

```
SQL>  CREATE STORAGE MAP LISTS_MAP
cont>     STORE LISTS IN AREA_ONE FOR (TABLE1.COL1)
cont>                 IN AREA_TWO FOR (TABLE1.COL2)
cont>                 IN RDB$SYSTEM;
```

You can use the LIST STORAGE AREA clause of the CREATE DATABASE statement to specify which storage area stores lists unless otherwise directed with a CREATE STORAGE MAP clause. If you do not use the LIST STORAGE AREA clause, Oracle Rdb uses RDB$SYSTEM as the default LIST storage area. The following example directs Oracle Rdb to place all lists in the LISTS storage area unless otherwise specified in a storage map:

```
SQL> CREATE DATABASE FILENAME mf_personnel
SQL>    LIST STORAGE AREA IS LISTS;
```

4.6.5.2 Storing List Data Randomly or Sequentially

You can store lists randomly or sequentially across several storage areas.

With random storage of lists, Oracle Rdb stores lists across multiple areas and fills the areas randomly. Random storage of lists is intended for read/write media that benefits from the I/O distribution across the storage areas.

The following example shows how to store lists randomly across two storage areas:

```
SQL>  CREATE STORAGE MAP LISTS_MAP
cont>     STORE LISTS
cont>           IN (LISTS1, LISTS2) FOR (RESUMES.RESUME)
cont>           FILL RANDOMLY
cont>           IN RDB$SYSTEM;
```

When a storage map specifies that a LIST column is stored in more than one storage area and specifies the FILL RANDOMLY clause (or does not specify the FILL SEQUENTIALLY clause), SQL stores that column randomly across those storage areas.

If you specify sequential storage of lists, Oracle Rdb stores lists in the first specified area until that area is filled. Sequential storage avoids excess swapping of platters when you store lists in write-once storage areas in a jukebox. The following example shows how to store lists sequentially:

```
SQL>  CREATE STORAGE MAP LISTS_MAP
cont>     STORE LISTS
cont>           IN (LISTS1, LISTS2) FOR (RESUMES.RESUME)
cont>           FILL SEQUENTIALLY
cont>           IN RDB$SYSTEM;
```

When a write-once storage areas is filled, Oracle Rdb marks it with a FULL flag and does not write to that storage area. To see if an area is filled, use the RMU Dump command to see if the FULL flag is set for that area.

4.6.5.3 Storing List Data on WORM Devices

Oracle Rdb lets you store lists in write-once storage areas on write-once, read-many (WORM) optical devices. WORM optical devices are less expensive than magnetic disks for storing large amounts of relatively stable list data. Consider storing list data on a WORM device in the following situations:

- The list data is stable. No new list data will be added for a long time.

  For example, an information catalog that is updated semiannually is available for read access. When you need to update the catalog, you can move the list data from the WORM device to a read/write disk, update the data, and move the list data back to the WORM device.

- New list data is added frequently, but that data is stable.

  When you write new data to a write-once storage area, the new list data is always written to the next unwritten block on the WORM device. If you have list data that probably will not change for a period of time, consider writing it to a WORM device. For example, image data, such as an employee's photograph, is relatively stable. Because each image may require 10 megabytes or more of storage space, it is costly to store such list data in read-write or read-only storage areas on read-write media. If the images are stable, you can permanently store these images at a much lower cost in write-once storage areas on a WORM device.

When you store lists in write-once storage areas, you can disable after-image journaling for those storage areas. Consider the following factors in deciding whether or not to disable after-image journaling:

- Because information in a write-once storage area is never overwritten, logging changes in the .aij file may be an unnecessary overhead.
- Disabling after-image journaling for write-once storage areas decreases the time spent rolling forward the database after a failure. When journaling for write-once storage areas is enabled, the area's records in the .aij file are applied even though the information in the area is never overwritten.
- If data written to the write-once storage area is not being logged to the .aij file, there is no guaranteed way of being able to recover from WORM media failures.

For more information about the advantages and disadvantages of disabling journaling for write-once storage areas, see the *Oracle Rdb7 Guide to Database Performance and Tuning*.

For information about creating write-once storage areas on WORM devices, see Section 4.5. For more information about moving list data to and from WORM devices, see Section 7.6.5 and Section 7.6.6.

4.7 Setting Sorted Index Characteristics for Performance

This section describes how you can optimize sorted (B-tree) indexes to maximize database performance.

You can specify the following characteristics for sorted indexes using the CREATE INDEX statement:

- Whether the index is a ranked or nonranked sorted index

  For information about specifying ranked and nonranked sorted indexes, see Section 3.14.1.
- The following types of compression:
  - Run-length compression
  - SIZE IS segment truncation
  - MAPPING VALUES compression
  - Duplicates compression

  For information about run-length compression, SIZE IS segment truncation, and MAPPING VALUES compression, see Section 3.14.6. For information about duplicate compression, see Section 3.14.1.

- Size of index nodes

  You can specify the size, in bytes, of each index node by using the NODE SIZE clause. The number and level of the resulting index nodes depend on this value, the number and size of the index keys, and the value of the PERCENT FILL clause. You can estimate the valid range for a user-specified index node size by determining the minimum and maximum node sizes.

- Initial fullness percentage

  You can specify the fullness percentage of the index nodes with the PERCENT FILL clause. This clause sets the *initial* fullness percentage for each node in the index structure in a range of 1 percent to 100 percent.

- Fullness percentage of each index node

  You can specify the fullness percentage of each index node by using the USAGE clause.

Section 4.7.1 describes how to calculate the size of sorted indexes. Section 4.7.2 describes how to specify the fullness percentage. Section 4.7.3 describes how to balance the node size and fullness percentages.

4.7.1 Calculating the Size of Sorted Indexes

You can specify the index size for ranked and nonranked sorted indexes by using the NODE SIZE IS clause of the CREATE INDEX statement. This section describes how to calculate a minimum and a maximum node size.

To calculate the minimum node size, use the following formula:

$$Minimum\ Node\ Size = (3 * (Key\ Size + No.\ of\ Segments + Key\ Overhead)) + 32$$

For ranked sorted indexes, the formula means the following:

- 3

  Assures that three entries always fit in an index node, which further assures that a perfect binary tree does not result. With key compression, frequently more than three entries fit into even this minimally sized node.

- Key Size

  Indicates the number of bytes it actually takes to represent the needed columns in the sorted index.

- No. of Segments

  Indicates the number of segments (columns) defined in the key.

- Key Overhead

Indicates the maximum number of overhead bytes per index key within a node. For ranked sorted indexes, the maximum is 18:

  - 2 bytes to hold the length of the variable length section (entry length)
  - 1 byte for the number of bytes in this entry that are not prefix compressed (separator length)
  - 1 byte for the number of bytes of last entry that are prefixed to this one
  - 1 byte to hold flags used to interpret the variable length section
  - 9 bytes for a dbkey that cannot be compressed
  - 4 bytes to hold the entry and leaf cardinality

- 32

  Represents index node overhead.

Assuming a key size of 1 and 1 segment, the formula yields a minimum node size of 92 bytes:

$$92 = (3 * (1 + 1 + 18)) + 32$$

For nonranked sorted indexes, the formula means the following:

- 3

  Assures that three entries always fit in an index node, which further assures that a perfect binary tree does not result. With key compression, frequently more than three entries fit into even this minimally sized node.

- Key Size

  Indicates the number of bytes it actually takes to represent the needed columns in the sorted index.

- No. of Segments

  Indicates the number of segments (columns) defined in the key.

- Key Overhead

  Indicates the maximum number of overhead bytes per index key within a node. For nonranked sorted indexes, the maximum is 11:

  - 1 byte for the number of bytes of last entry that are prefixed to this one
  - 1 byte for the number of bytes in this entry that are not prefix compressed (separator length)
  - 9 bytes for a dbkey that cannot be compressed

- 32

  Represents index node overhead.

Assuming a key size of 1 and 1 segment, the formula yields a minimum node size of 71 bytes:

$$71 = (3 * (1 + 1 + 11)) + 32$$

The maximum node size for ranked and nonranked sorted indexes is 32,767 bytes, but you can determine a practical upper limit with the following formula:

$$Maximum\ Node\ Size = Page\ Size - (Fixed + Snapshot + Variable\ Overhead)$$

Every index page has a fixed overhead of 42 bytes. In addition, Oracle Rdb takes into account the extra overhead (8 bytes) used on a snapshot page. The variable overhead for storing one row is 4 bytes for the line index, 4 bytes for the TSN index, and 2 bytes for the live line pointer on the snapshot page, for a total variable overhead of 10 bytes. Thus, the fixed and variable overhead needed to store one row on a page is 42+8+10 = 60 bytes. For a page size of 2 blocks, the maximum node size is 964 bytes:

$$964 = 1024 - (42 + 8 + 10)$$

An uncompressed node larger than 964 bytes is fragmented if stored on a 2-block page. (Refer to the *Oracle Rdb7 Guide to Database Maintenance* for detailed information on page structures.)

Specific recommendations for selecting node sizes, along with fullness percentages, are given in Section 4.7.3. Continuing with the previous calculations for a default page size of 2 blocks, if you specify a node size outside the calculated range, Oracle Rdb selects:

- The value 430 bytes, if it is large enough for the minimum node size.
- The value 860 bytes, if 430 is not large enough for the minimum node size.

For example, using ranked sorted indexes, if the key size is 114 bytes, and you specify a value lower than the minimum of (3 * (114+1+18)) + 32, or 431 bytes, Oracle Rdb supplies the value of 860 bytes.

If you omit the NODE SIZE clause, Oracle Rdb determines the default value from this same formula. For ranked indexes, Oracle Rdb uses 430 bytes if the total key size is 113 bytes or less; 860 bytes if the total key size is more than 113 bytes. For nonranked indexes, Oracle Rdb uses 430 bytes if the total key size is 120 bytes or less; 860 bytes if the total key size is more than 120 bytes.

If you specify a node size value that is lower than the minimum value for the index, the metadata update fails and Oracle Rdb displays the following error message:

```
%RDB-E-NO_META_UPDATE, metadata update failed
-RDB-E-IMP_EXC, facility-specific limit exceeded
-RDMS-F-INDEX_S_MIN, user requested node size of 70 bytes for index needing 83
```

4.7.2 Specifying Fullness Percentages for Sorted Indexes

You can specify the fullness percentage of the index nodes with the PERCENT FILL clause. This option sets the *initial* fullness percentage for each node in the index structure in a range of 1 percent to 100 percent. The default fullness percentage is 70 percent. For example, suppose you specify PERCENT FILL 50 in the CREATE INDEX statement, and the NODE SIZE value is 400 bytes. Oracle Rdb builds as many index levels as necessary to ensure that the entries in each node do not exceed 50 percent of the node size, minus 32 bytes. Note that Oracle Rdb always attempts to include *at least* two keys per node.

The USAGE clause sets the fullness percentage of each index node. You can choose one of the following options:

- USAGE UPDATE

  If you select USAGE UPDATE, the default, Oracle Rdb sets the fullness percentage of each index node at 70 percent.

- USAGE QUERY

  If you select USAGE QUERY, Oracle Rdb sets the fullness percentage of each index node at 100 percent.

You can supply both the PERCENT FILL and USAGE clauses; the USAGE option takes precedence over an explicit PERCENT FILL value.

4.7.3 Balancing Node Size and Fullness Percentages

Generally, query-intensive applications can benefit from index structures with large, full nodes, while update-intensive applications can benefit from index structures with small, partially full nodes.

If your application performs 100 percent retrieval on a particular table, set the initial fullness percentage of an index node, or the fill factor, to 100 percent on indexes for that table. By specifying that you want each index node filled, the resulting index tree contains fewer levels. For each read operation, you increase the likelihood that the keys your query needs are available in your buffer.

If your application updates a table frequently, try setting the node to a small size and try setting the initial fullness percentage to a small value. Because the settings are dynamic, experiment with the new values and change as necessary.

Note that when an index node fills, the node splits, thereby creating two parallel nodes. In the B-tree structure, pointers in the node above the two nodes that split must be updated. A user of an index leaves *read locks* while traversing the hierarchical index structure, and takes out a *write lock* on the index node currently being updated. But when a node splits, write locks must be placed on higher nodes as pointers of the parent nodes are updated. The locking can have a ripple effect up the B-tree structure and lock out concurrent users who attempt to use the same index. A fill factor set high enough can prevent this type of split, and the potential lock, from occurring.

Thus, these fill factor and node size settings involve a two-way trade-off. If you set the fill factor to a low value, which results in an index structure with many levels, little computation time may be spent to find records because individual nodes contain few keys and it is easy to get to the next index level. When you set the fill factor and node size to high values, causing each index node to contain many keys, it takes more CPU cycles to look through each node.

Generally, a deep index structure (one having many levels) requires more I/O operations because memory is usually not sufficient to hold the entire index tree structure. When you attempt to tune the index, consider the available memory of your system and determine what is most practical for the size of your index. That is, because index size is a function of the number of records to be indexed and the size of the key, and overall index size is dominated by the size of the leaf nodes, you may be able to fit only a portion of your index in memory. Therefore, using the index requires additional I/O operations. Within practical limits of available memory, additional tuning may not be possible considering that the index is properly tuned to this point. For example, be sure that the database page size is set large enough to store the largest index node of any index in your database, especially if you store all these sorted index structures in the RDB$SYSTEM storage area.

If you set a large index node size and the index keys are small in length, your transaction may lock out many users because each node may contain the index pointers to many (30 or more) records. To avoid this situation, make each node smaller when the index key is small. The fewer keys in each node, the less locking occurs during times of many simultaneous transactions. Remember to increase the NUMBER OF BUFFERS value for the database so that the number of I/O operations does not increase significantly.

You can use the ALTER INDEX statement to periodically reorganize the index to obtain more free space. For more information about how to use the PERCENT FILL, NODE SIZE, and USAGE clauses, see the Section 7.7.1.

In addition, see the *Oracle Rdb7 Guide to Database Performance and Tuning* for information about using the RMU Analyze Indexes command to display information about indexes and the Oracle Rdb Performance Monitor to display useful information for tuning sorted indexes, including evaluating trade-offs and sizing compressed indexes. The *Oracle Rdb7 Guide to Database Performance and Tuning* contains a decision tree that summarizes the steps you can follow to analyze index performance.

4.8 Setting Database and Storage Area Parameters When Using Hashed Indexes

When you use hashed indexes, correct file allocation and page size calculations for storage areas are critical. To determine the size of a storage area, you must understand the following factors:

- The data to be stored in the storage area, including record size, the number of unique key values, the key size, the number of duplicate rows, and the total number of rows in the tables associated with the hashed indexes
- The database and storage area parameters that you can specify, including storage area page size, allocation, and SPAM thresholds and intervals
- Where the hashed index is stored relative to the data—in separate storage areas or in the same storage area using the PLACEMENT VIA INDEX option

Remember that you can store hashed indexes only in mixed page format storage areas.

To determine the size of the storage area, first calculate the page size for the storage area and then how many data rows are to be stored in the storage area. The sections that follow describe:

- The factors that affect the page size: Section 4.8.1
- How to calculate the page size: Section 4.8.2 through Section 4.8.6
- How to calculate the file allocation size: Section 4.8.7

4.8.1 Understanding the Page Overhead and Record Types

Before you calculate the page size, you must determine the page overhead, what record types the data page can hold, the frequency of occurrence of these record types, and the space used by each record type (data rows and index records), including overhead. The fixed and variable page overhead and record types that can occur on a data page include:

- Fixed and variable page overhead

  The fixed page overhead includes the page header and page tail for the data page. The variable page overhead includes the line index and transaction sequence number (TSN) index for each record that occurs on the data page.

- System record

  The system record contains the pointers (dbkeys) to the hash bucket or buckets that occur on the data page. There is always one system record per page and rarely do these system records overflow to a nearby page; if they do, a system record overflow record results.

- Hashed index structures

  - Hash bucket

    The hash bucket contains the hash bucket entries (one entry per data row or duplicate node record) that hash to the page. There is one hash bucket per hashed index for each table of data rows that hash to the page. If two tables are stored within the same storage area, two hash buckets (one for each table's index) contain the dbkeys that point to each respective table's data rows or duplicate node records that can fit on the page.

  - Duplicate node record

    If there are duplicate data rows or if multiple child rows are associated with a parent row, Oracle Rdb creates duplicate node records to hold the dbkeys for these duplicate data rows. Each duplicate node record can hold 10 dbkeys. If there are more than 10 duplicate data rows, Oracle Rdb creates a second duplicate node record, chained to the first duplicate node record, and so forth. If there are no duplicate records, Oracle Rdb creates no duplicate node records.

- Data row

  The data row contains the column values or data. If data rows from two or more tables are clustered on the page using the PLACEMENT VIA INDEX clause, each table's data rows hash to the same page based on their common key value.

The following sections describe the procedures for calculating fixed and variable page overhead and the record sizes for each record type on the data page.

4.8.2 Calculating the Size of Fixed and Variable Page Overhead

The fixed page overhead includes the page header and page tail; the variable page overhead includes the line index and TSN index.

Neither the line nor TSN index is considered part of the record overhead, but they are both used in calculations to determine if a record type (system, hash bucket, duplicate node, or data row) can fit on the page. These two indexes both locate and identify each record stored on the page. The amount of storage used is proportional to the number of records stored on the page (this includes the system record and any hash buckets). A small, fixed, 2-byte portion is also required to hold the record count.

Table 4–1 provides information for calculating the fixed and variable overhead for a data page.

Table 4–1 Calculating the Fixed and Variable Overhead for a Page

Category	Variable	Bytes per Entry	Total
FIXED PAGE OVERHEAD			
Page header	a	22	
Page tail	b	18	
Total		a+b	40
VARIABLE PAGE OVERHEAD—LINE and TSN INDEX[1]			
System record[2]	c+d	1(4+4)	8
Hash bucket[3]	c+d	2(4+4)	16
Duplicate node record[4]	c+d	1(4+4)	8

[1]The line index (4 bytes) and TSN index (4 bytes) plus a small fixed portion of 2 bytes are not considered part of the record, but are used in calculations that determine if a record type (system record, hash bucket, duplicate node record, or data row) can fit on the page.

[2]One line index (c) and TSN index (d) per system record per page (rarely more than one per page).

[3]One line index (c) and TSN index (d) per hash bucket on the page (one or more per page).

[4]One line index (c) and TSN index (d) per duplicate node record on the page (one or more per page).

(continued on next page)

Table 4–1 (Cont.) Calculating the Fixed and Variable Overhead for a Page

Category	Variable	Bytes per Entry	Total
VARIABLE PAGE OVERHEAD—LINE and TSN INDEX[1]			
Data row[5]	c+d	4(4+4)	32
Total[6]	2+8(c+d)		66

[1]The line index (4 bytes) and TSN index (4 bytes) plus a small fixed portion of 2 bytes are not considered part of the record, but are used in calculations that determine if a record type (system record, hash bucket, duplicate node record, or data row) can fit on the page.

[5]One line index (c) and TSN index (d) per data row on the page (one or more per page).

[6]The total depends on the occurrence and total frequency (n) for all record types on the page plus 2 bytes of fixed overhead required to hold the record count.

If you group more than one parent/child record on a page, note the following about the number of bytes required for the variable page overhead:

- The number of bytes required for the system record remains the same.
- The number of bytes required for the hash bucket remains the same.
- The number of bytes required for the duplicate node record is multiplied by the number of parent/child records on a page.

See Table 4–8 for an example of the different sizes required when you group more than one parent/child record on a page.

4.8.3 Calculating the Size of Hashed Index Structures

The hashed index can contain hash buckets and duplicate node records. Table 4–2 shows how to calculate the size of hashed index structures. The variable "a" (the number of hashed indexes defined for a storage area) is applied when "a" is greater than one (a>1) or when a storage area contains more than one hashed index. This table is especially useful for calculating the sizes of hashed indexes when both indexes and data are placed on the same page using the PLACEMENT VIA INDEX option in the CREATE STORAGE MAP or ALTER STORAGE MAP statement.

Table 4–4 in Section 4.8.5 shows an example of calculating data row sizes. Note that because of synonyms, collisions are possible, and you cannot predict the exact number of hash bucket entries during hashing. A **synonym** is a column value that has the same or similar meaning as another column value and hashes to the same hash bucket, thus becoming another primary entry in the hash bucket.

Table 4–2 Calculating the Size of Hashed Indexes

Category	Bytes per Entry	Total
SYSTEM RECORD		
No. of hashed indexes in storage area	a	a
Total system record size		
Overhead	4	4
Minimum	6	(6*a)+4
Maximum[1]	10	(10*a)+4
HASH BUCKET		
Total hash bucket entry size		
Key size[2]	1	1
Key length[3]	k+1	k+1
Overhead/entry[4]	12	12
Total/entry[5]	12+1+k+1=b	b
No. of entries[6]	c	c
Overhead/bucket[7]	13	13
Total bucket size	(b*c)+13	(b*c)+13
DUPLICATE NODE RECORD		
No. of duplicates[8]	d	d
No. of entries/node[9]	10	

[1]Assume the maximum size for the system record.

[2]A single unsigned byte is used to store the key length.

[3]The VARCHAR (or VARYING STRING) data type includes a 2-byte length field that is ignored for the index field length. The key value is space filled in the index. One byte (the 1 of K+1) is used to indicate null values.

[4]Duplicate count field (4) plus dbkey pointer (8).

[5]Size of each hash entry in the hash bucket is the sum of key size plus key length plus overhead/entry.

[6]An entry is any key value that maps to the same page, but is not a duplicate of an existing entry.

[7]Record type (4) plus overflow bucket dbkey pointer (8) and flags field (1).

[8]Number of duplicate records for the same value.

[9]The maximum number of duplicate entries that can fit in a duplicate node.

(continued on next page)

Table 4–2 (Cont.) Calculating the Size of Hashed Indexes

Category	Bytes per Entry	Total
DUPLICATE NODE RECORD		
No. of duplicate nodes[10]	(d+5)/10=e	e
Overhead/node[11]	92	92
Total	e*92	e*92
GRAND TOTAL FOR THE HASHED INDEX		
Minimum		(6*a)+4
Maximum		((10*a)+4) + ((b*c)+13) + (e*92)

[10]The number of duplicate nodes rounded up to nearest whole number; if there are no duplicate records, no duplicate nodes are created.

[11]Total overhead of one duplicate node.

4.8.4 Calculating the Size of Hashed Indexes for the Sample Database

Table 4–3 shows how to calculate the size of hashed indexes for the very small database mf_personnel. Remember, Table 4–3 is for illustration only and the real benefit of a hashed index is accessing rows from fairly large tables. Table 4–3 uses two hashed indexes, EMPLOYEES_HASH and JOB_HISTORY_HASH, and the data rows for the EMPLOYEES and JOB_HISTORY tables, all stored over three storage areas, empids_low.rda, empids_mid.rda, and empids_over.rda.

Table 4–3 Calculating the Size of Hashed Indexes for the Mf_personnel Database

Category	EMPLOYEES_ HASH Index	JOB_HISTORY_ HASH Index	Total
SYSTEM RECORD			
No. of hashed indexes	1	1	2
Total system record size			
Overhead			4
Minimum	6*1=6	6*1=6	12+4=16

(continued on next page)

Table 4–3 (Cont.) Calculating the Size of Hashed Indexes for the Mf_personnel Database

Category	EMPLOYEES_ HASH Index	JOB_HISTORY_ HASH Index	Total
SYSTEM RECORD			
Maximum	10*1=10	10*1=10	20+4=24
HASH BUCKET			
Key size	1	1	
Key length	5+1=6	5+1=6	
Overhead/entry	12	12	
Total	12+1+6=19	19	
No. of entries	1	1	
Overhead/bucket	13	13	
Total bucket size	(19*1)+13=32	(19*1)+13=32	64
DUPLICATE NODE RECORD			
No. of duplicates	0	3	
No. of entries/node	10	10	
No. of duplicate nodes	0	(3+5)/10=1	
Overhead/node	92	92	
Total	0*92	1*92	92

You calculate the hashed index sizes as follows:

- System record

  A hashed index's system record entry requires 6 to 10 bytes (compressed) per pointer to each hashed bucket (each defined hashed index) multiplied by the number of hashed indexes defined for the storage area plus 4 overhead bytes for a total of between 16 to 24 bytes.

- Hash bucket

  Each entry in the hash bucket requires the key size plus the length of the key entry plus 1 plus 12 bytes. If the key size for the EMPLOYEE_ID column takes 1 byte and the key entry is 5 bytes long plus 1 byte overhead or 6 bytes, and the overhead per entry is 12 bytes, each hash bucket entry

totals 19 bytes. This total plus 13 bytes of overhead for each bucket equals a grand total of 32 bytes per hashed index or a total of 64 bytes for both hashed indexes.

- Duplicate node record

  If duplicates are allowed, each duplicate node record is 92 bytes. Each duplicate node record contains sufficient space to hold pointers to a maximum of 10 duplicate records; for every 10 duplicate records, Oracle Rdb creates another duplicate node record and points to it from the previous duplicate node record. No duplicate records are allowed in the EMPLOYEES_HASH hashed index. There is one duplicate node record in the JOB_HISTORY_HASH hashed index that contains an estimated three pointers to the three JOB_HISTORY records for employee Janet Kilpatrick. This record requires 92 bytes. Note that the number of duplicate records is an average estimated value for the purpose of sizing pages.

  Because of synonyms in which different key values hash to the same page, you cannot predict the exact number of entries in the hash bucket. If there are duplicate values, a second duplicate node record is created. You may need to consider this when sizing pages and allocating storage areas.

- Total size of two hashed indexes

  To calculate the total size of both hashed indexes in the storage area, add the byte totals for each record type. The system record is between 16 to 24 bytes long; both hash buckets total 64 bytes; and the duplicate node record is 92 bytes, for a total of between 172 to 180 bytes on the page.

For information about using RMU commands to help you size compressed indexes, see the *Oracle Rdb7 Guide to Database Performance and Tuning*.

4.8.5 Calculating the Size of Data Rows

Now consider the space needed for data rows in the same storage area. To prevent fragmentation, you can calculate this space by basing the space on a full, uncompressed row or by using the average compressed row size. To determine the row size, add its column sizes, calculate its overhead, and add these values together.

For the mf_personnel database, each EMPLOYEES row consists of the columns and column sizes shown in Table 4–4.

Table 4–4 Column Sizes in the EMPLOYEES Table

Column	Domain	Data Type	Size in Bytes
EMPLOYEE_ID	ID_DOM	CHAR	5
LAST_NAME	LAST_NAME_DOM	CHAR	14
FIRST_NAME	FIRST_NAME_DOM	CHAR	10
MIDDLE_INITIAL	MIDDLE_INITIAL_DOM	CHAR	1
ADDRESS_DATA_1	ADDRESS_DATA_1_DOM	CHAR	25
ADDRESS_DATA_2	ADDRESS_DATA_2_DOM	CHAR	20
CITY	CITY_DOM	CHAR	20
STATE	STATE_DOM	CHAR	2
POSTAL_CODE	POSTAL_CODE_DOM	CHAR	5
SEX	SEX_DOM	CHAR	1
BIRTHDAY	DATE_DOM	DATE	8
STATUS_CODE	STATUS_CODE_DOM	CHAR	1
Total bytes			112

Each JOB_HISTORY row consists of the columns and column sizes shown in Table 4–5.

Table 4–5 Column Sizes in the JOB_HISTORY Table

Column	Domain	Data Type	Size in Bytes
EMPLOYEE_ID	ID_DOM	CHAR	5
JOB_CODE	JOB_CODE_DOM	CHAR	4
JOB_START	DATE_DOM	DATE	8
JOB_END	DATE_DOM	DATE	8
DEPARTMENT_CODE	DEPARTMENT_CODE_DOM	CHAR	4
SUPERVISOR_ID	ID_DOM	CHAR	5
Total bytes			34

The EMPLOYEES row is 112 bytes long and the JOB_HISTORY row is 34 bytes long.

You can quickly calculate the column sizes for all tables in a database by querying the system tables. For example, to find the sizes of the columns in all tables in the mf_personnel database, use the statement in Example 4–15.

Example 4–15 Finding the Sizes of Columns in a Table

```
SET DIALECT 'SQL92';
SELECT r.rdb$relation_name,
       SUM(CASE WHEN f.rdb$field_type = 37 -- VARCHAR data type
                THEN f.rdb$field_length + 2
                ELSE f.rdb$field_length
           END) AS "Byte Count",
       COUNT(*) AS "Column Count"
  FROM rdb$relations r, rdb$relation_fields rf, rdb$fields f
  WHERE r.rdb$relation_name = rf.rdb$relation_name
        AND rf.rdb$field_source = f.rdb$field_name
        AND r.rdb$system_flag = 0
  GROUP BY r.rdb$relation_name;
 R.RDB$RELATION_NAME                              Byte Count   Column Count
 CANDIDATES                                              282              4
 COLLEGES                                                 56              5
 CURRENT_INFO                                             99              8
 CURRENT_JOB                                              50              7
 CURRENT_SALARY                                           41              5
 DEGREES                                                  29              5
 DEPARTMENTS                                              47              5
 EMPLOYEES                                               112             12
 JOBS                                                     33              5
 JOB_HISTORY                                              34              6
 RESUMES                                                  13              2
 SALARY_HISTORY                                           25              4
 WORK_STATUS                                              23              3
13 rows selected
```

The data row overhead bytes are calculated for a noncompressed data row in the manner shown in Table 4–6.

Table 4–6 Calculating the Fixed Overhead for a Page

Category	Variable	Bytes per Entry	Total
EMPLOYEE Row Overhead			
No-compression bytes[1]	a	1	
Version number	b	2	
Null bytes[2]	c	2	
Total		a+b+c	5
JOB_HISTORY Row Overhead			
No-compression bytes[1]	a	1	
Version number	b	2	
Null bytes[2]	c	1	
Total		a+b+c	4

[1]Minimum number of no-compression bytes = (uncompressed row size + 64)/128

[2]1 byte per 8 columns calculated by the formula: number of null bytes = (number of columns in the row +7)/8

There are 5 overhead bytes for each EMPLOYEES row and 4 overhead bytes for each JOB_HISTORY row.

Table 4–7 shows the row overhead for the data row on the page. This overhead includes a 2-byte record type identifier for the data row and an additional 3 bytes of control information.

Table 4–7 Calculating the Data Row Overhead for a Page

Category	Variable	Bytes per Entry	Total
Data Row Overhead			
Data row overhead[1]	a+b	5	
Total		a+b	5

[1]Data row overhead consists of a 2-byte record identifier (a) and 3 bytes of control information (b).

4.8.6 Calculating the Page Size

After you calculate the page overhead, the record overhead, and the record sizes for all record types that will be stored on the page, you calculate the page size required to hold these records. Table 4–8 shows the results of the calculations that apply to the EMPIDS_LOW, EMPIDS_MID, and EMPIDS_HIGH storage areas for the mf_personnel database.

Table 4–8 Calculating the Page Size

Category	Bytes per Entry	1-Block Page Size	2-Block Page Size	3-Block Page Size
Fixed page overhead	40	40	40	40
Variable page overhead	2+8*8	66	106	146
System record size	24	24	24	24
EMPLOYEE_HASH hash bucket size	32	32	51	70
JOB_HISTORY_HASH hash bucket size	32	32	51	70
JOB_HISTORY_HASH duplicate node size	92	92	184	276
EMPLOYEE row size	112	112	224	336
EMPLOYEE row overhead	10	10	20	30
JOB_HISTORY row size	34*3	102	204	306
JOB_HISTORY row overhead	((4)*3)+((5)*3)	27	54	81
Total		537	958	1379

Table 4–8 shows the total amount of space needed to store one group of parent/child data rows, associated index records, record overhead, system record, and fixed page overhead. To size the data page, determine in increments of 1 block (512 bytes) the minimum page size that fits this one group of records, beginning with a 1-block page size. If the space needed to store one group of records is larger than the page size, the page size is too small to efficiently store one group of records. Usually the next larger page size by 1-block increment is large enough.

Table 4–9 shows these calculations for three different sized pages. Note that the 1-block page size is too small and that the next larger page size (2 blocks or 1024 bytes) is large enough to hold two groups of records, leaving 66 bytes of available space on the page. A 3-block page size can hold three complete groups of records and leave 157 bytes of available space on the page.

4.8.7 Calculating the File Allocation Size

To calculate the file allocation size, determine the number of data rows for each table. The EMPLOYEES table has 100 data rows; the JOB_HISTORY table has 274 data rows. After you determine an adequate page size, divide the number of parent rows (100) by the number of groups of records that the page can hold.

Table 4–9 shows the file allocation sizes for three different page sizes that would hold the EMPLOYEES and JOB_HISTORY rows. The calculated file allocation size has a 90 percent fill factor, or, about 10 percent extra space in the file for storing records. This represents a fairly stable storage area. If you expect the storage area to grow, use a fill factor that would give you 50 to 70 percent fill of the storage area. The value you select depends on how much disk space you have available for your database application, how quickly the storage area is expected to fill, and how often you want or need to tune your database.

Table 4–9 Calculating the File Allocation Size

Category	1-Block Page Size	2-Block Page Size	3-Block Page Size
Page size (bytes)	512	1024	1536
Fixed page overhead/page (bytes)	40	40	40
No. of EMPLOYEE data rows	100	100	100
No. of JOB_HISTORY data rows	274	274	274
No. of record groups/page	0	2	3
Space used by records (bytes)	537	958	1379
Available page space (bytes)	–15	66	157
File allocation/area (pages)[1]	None	50	33

[1]File allocation size is calculated by the formula: Total number of data rows/Number of record groups per page times 1.1 for a 90% fill factor + No. of SPAM pages

Note that the 1-block page size is too small (by 15 bytes) to hold one record group. If you use this page size and load the data rows into the storage area, there will be a relatively large number of hash bucket overflows and data row displacements from their hash buckets. As a result, you might lose the performance objective of one I/O operation for some exact match retrievals. Avoid using page sizes where at least one entire record group cannot fit on the

same page. The next page size is 2 blocks or 1024 bytes. This is the minimum recommended page size because two complete record groups fit on the page.

In collisions or synonyms where more than two parent records hash to the same page though their key value is different, the record group is placed in whole or in part on an adjacent or nearby page. If your buffer size is 6 pages, make certain that you can retrieve the entire record in one I/O operation or that the minimum I/O value is still 1 as determined from placement analysis using the RMU Analyze Placement command. If it is not, reload your data and make the file allocation larger (perhaps 20 percent larger so there are more pages to which to hash records). Alternatively, you might select the next larger page size and again make the file allocation larger.

These calculations estimate the number of records that fit on a page. Often, users assume that the number of hash entries equals the number of rows on the page. In fact, a row may have been stored on this page as overflow from another page, or a row may fit on the current page, but the hash bucket may have insufficient room to be extended and may overflow to an adjacent or nearby page. Oracle Rdb may choose to store a data row and fragment the hash index on another page.

A real application is unlikely to have evenly distributed key values. After calculating the storage area allocation, using the INSERT . . . PLACEMENT ONLY RETURNING DBKEY statement to get the dbkey values, and sorting the dbkeys, you should examine the dbkeys that have been saved and determine the maximum number of entries that hash to the same page. Use this value to size the data page again, or adjust the allocation, use the INSERT . . . PLACEMENT ONLY RETURNING DBKEY statement again, and then examine the dbkeys again before loading the data. This cycle guarantees the best fit of the data to the storage area. In practice, if you carefully calculate the page and file allocation size needed for a storage area, your first estimate shown in Table 4–10 will probably be adequate. See Chapter 6 for more information on using the INSERT . . . PLACEMENT ONLY RETURNING DBKEY statement and other considerations when loading data into storage areas.

To calculate the allocation required to store 100 EMPLOYEES and JOB_HISTORY data row groups so that two complete record groups fit per page, use the formula shown in Table 4–10.

Table 4–10 Calculating the File Allocation Size to Store 100 Data Pages

Data allocation	$= No.\ of\ rows/(No.\ of\ record\ groups\ that\ fit\ on\ a\ page) \times File\ fill\ factor$
	$= 100/(2) \times 1.1$
	$\cong 55$

For reasonably stable storage areas, increase the file allocation by 10 percent. For storage areas you expect to grow, increase the file allocation by 30 percent. If you use these numbers, a reasonably stable storage area with a 2- and 3-block page size has an estimated file allocation size of 55 and 36 pages respectively, while a storage area that is expected to grow has an estimated file allocation size of 65 and 43 pages, respectively.

Oracle Rdb creates space area management (SPAM) pages at regular intervals throughout the storage area. If you do not explicitly define this interval for the storage area, it defaults to 216 pages. For large to very large storage areas for which you want to calculate the amount of space that a storage area file uses on the disk, it is useful to know the number of database pages required for space area management. In this case, the amount is actually quite small, as shown in Table 4–11.

Table 4–11 Calculating the SPAM Pages and Adding These Pages to the Estimated File Allocation Size

File allocation	$= Data\ allocation + (Data\ allocation/SPAM\ interval)$
	$= 55 + (55/216)$
	$\cong 55 + 1$
	$\cong 56$

In Table 4–11, about 1 percent of the allocated space is used for SPAM pages. You should always consider the amount of space allocated to the SPAM pages so you can adjust the allocation size accordingly.

In the mf_personnel database, each storage area defined to store the EMPLOYEES and JOB_HISTORY data rows (EMPIDS_LOW, EMPIDS_MID, and EMPIDS_OVER) has an initial 50-page allocation or a total of 150 pages for all three horizontally partitioned storage areas.

If you use the RMU Analyze Areas command to analyze the EMPIDS_LOW storage area, you can see that too much free space (67 percent free bytes) exists in the storage area, partly because the data rows are compressed, and partly because about two-thirds more file allocation space is used than was originally estimated. If the EMPLOYEES and JOB_HISTORY tables grow at a moderate rate in these three storage areas, this total allocation is acceptable. But if these two tables are relatively stable in size, this is wasted space. If the number of employees is fairly stable, but employees change jobs frequently, many duplicate JOB_HISTORY rows may result. As the average number of duplicate JOB_HISTORY rows increases from its current value of 2.74 to around 12, the available file allocation space will be used up.

Extra space may or may not be considered wasted space depending on the following factors:

- How you anticipate that space to be used over a period of time
- How soon the extra space is to be used
- How soon you want to perform some tuning to provide extra space before pages fill up, files extend, and performance drops

Knowing how many duplicate rows there are, what they are, and how they are distributed gives you an idea of how much space on the page will be dedicated to both the duplicate node records and the data rows. If you have many duplicate records, your performance may drop when you use hashed indexes, especially if page sizes are not sufficiently large and overflow occurs. As more and more pages overflow, you soon lose the advantage of having everything you need in the buffer in one I/O operation, and a second or third I/O operation is needed to gather the required information. To avoid this problem, you can check your statistics (using the Oracle Rdb Performance Monitor) for hashed indexes to inspect the values for hashed index fetches, and particularly bucket fragments and duplicates.

To determine how performance might be improved when you use a hashed index for your tables, do some size calculations and understand your data, especially how your duplicate rows are distributed. If the distribution of duplicate rows is skewed on certain rows, you may be able to define a multisegmented key to make this distribution more uniform. For example, instead of just the LAST_NAME column as the primary key (where skewing may occur around last names like SMITH and JONES), define a multisegmented key of the LAST_NAME and the FIRST_NAME columns. If possible, select a unique key as the hash key, for example, social security number, where no duplicates are allowed.

When duplicate rows are necessary, some adjustments and trade-offs may be required as to where you place duplicate rows and where you place the hashed index defined for the table with duplicate rows. Because a duplicate node can hold 10 entries that point to each duplicate row, additional duplicate nodes are required for each set of 10 duplicate rows. Each duplicate node takes up 92 bytes of space on the page. So, if duplicate rows are small and there are no more than 9 or 10 per value, there may be enough space on the page (or you can adjust the page size) to store the duplicate rows with the parent rows and the hashed indexes and still maintain your performance objective of one I/O operation.

If not all data can fit on the same page, data may spill over to adjacent pages, nearby pages, or pages outside the range of the buffer. If this happens, you may no longer consistently achieve one I/O operation due to hash bucket overflows and row displacement from hash buckets. You may be fortunate to initially achieve two I/O operations, but as you insert new rows, performance may drop dramatically. You are faced with the prospect of constantly changing the storage maps to tune the placement of the rows by reorganizing the storage area.

To achieve at least two I/O operations and ensure that this performance objective can be maintained for some period of time without more tuning, consider the following alternatives:

- Enlarge the page size for the storage area and the buffer size for the application.

  You can make the page size only so large and increase the buffer size only so much before you begin to affect other performance aspects of your application or begin to exceed the available system resources.

- Shadow the duplicate child data rows with the parent data rows.

  For the storage area that contains the child duplicate rows, select a page size that holds only this record type and then choose a size allocation that maintains the clustering effect for some period of time. The storage area that holds the parent rows and both hashed indexes can be appropriately sized for page and allocation size to contain only these record types and index structures. This approach may be practical when you have many duplicate rows with two or more duplicate nodes per page and moderate-to-large duplicate row sizes.

- Cluster the child duplicate rows with the parent rows in a separate storage area from the hashed indexes.

Assuming page and file allocation sizes are optimal, you can achieve a minimum of two I/O operations with this arrangement. One I/O operation is used to access the hashed index for the parent table; one I/O is used to read the data row in the other storage area. In the best case, subsequent access of the duplicate child row is read directly from the buffer, requiring zero I/O operations because the parent and child rows are clustered together.

- Put the hashed index for the child duplicate rows in a separate storage area and shadow the data rows with the index.

  Potentially, you can achieve three I/O operations with this arrangement if you store the parent rows and its hashed index in one storage area, the child rows in a second storage area, and the hashed index for the child rows in a third storage area. One I/O operation is required to read the parent row. A second I/O operation is required to access the index for the child duplicate indexes. By shadowing the child record index with the child data rows, a third I/O operation is required to read the child data row. This arrangement guarantees three I/O operations, but may be useful in situations where duplicate rows are very large and numerous.

Duplicate rows exact a cost on your hashed index but, depending on your application, may be necessary. Knowing why this is so and what adjustments you can make are important considerations. When duplicate rows are not allowed, the performance gains from use of hashed indexes are significant.

Section 4.9 provides more information on strategies for clustering, shadowing, and placing rows.

4.9 Implementing Placement and Clustering Strategies Using Hashed Indexes

This section examines placement and clustering of rows using hashed indexes. It describes the concepts of placement, clustering, and shadowing and presents the relative merits of each configuration. These configurations are described relative to where rows and hashed indexes are stored, whether or not the PLACEMENT VIA INDEX clause is used to store (place) the rows, and whether (parent and child) rows are clustered or shadowed:

- Separate storage areas, no placement clause

  Placing rows and the hashed index in separate storage areas and not using the placement clause option to place rows

- Separate storage areas, with placement clause

Placing rows and the hashed index in separate storage areas and using the placement clause to place rows

- Same storage area, with placement clause (one I/O operation)

  Placing rows and the hashed index in the same storage area and using the placement clause to place rows

- Clustering: add child rows, separate storage area for indexes, with placement clause

  Placing parent and child rows in the same storage area, and hashed indexes in separate storage areas and using the placement clause to place rows

- Shadowing: child and parent rows in separate storage areas, with placement clause

  Placing parent data rows and hashed indexes in one storage area, and child rows in a second storage area, and using the placement clause

- Clustering: child and parent rows and indexes all in the same storage area, with placement clause

  Placing parent and child rows and hashed indexes all in the same storage area and using the placement clause to place rows

4.9.1 Separate Areas, No Placement Clause

You can store table rows and hashed indexes in separate storage areas. In Example 4–16, the EMPLOYEES table rows and its EMPLOYEES_HASH hashed index are stored in separate storage areas. The AREA_A storage area is used for the data and could be a uniform page format, which is suited to sequential retrieval. The AREA_B storage area must be a mixed page format because a hashed index requires a system record and a hash bucket on the same page.

The rows of the EMPLOYEES table are scattered arbitrarily in the storage area. Oracle Rdb chooses the first available page with space to insert the next row.

Example 4–16 Placing Rows and the Hashed Index in Separate Storage Areas and Not Using the Placement Clause

```
SQL> CREATE STORAGE MAP EMP_MAP
cont>    FOR EMPLOYEES
cont>    STORE IN AREA_A;
SQL> --
SQL> CREATE UNIQUE INDEX EMPLOYEE_HASH
cont>     ON EMPLOYEES (EMPLOYEE_ID)
cont>     STORE IN AREA_B
cont>     TYPE IS HASHED;
```

4.9.2 Separate Areas, with Placement Clause

Adding a PLACEMENT VIA INDEX clause as shown in Example 4–17 allows the data to shadow the hash bucket page. This tends to disperse the rows randomly over the AREA_A storage area. When other tables are grouped with the rows of the EMPLOYEES table, this random dispersal can be useful, leaving space for related rows of different types.

Example 4–17 Placing Rows and the Hashed Index in Separate Storage Areas and Using the Placement Clause

```
SQL> CREATE STORAGE MAP EMP_MAP
cont>    FOR EMPLOYEES
cont>    STORE IN AREA_A;
cont>    PLACEMENT VIA INDEX EMPLOYEE_HASH;
SQL> --
SQL> CREATE UNIQUE INDEX EMPLOYEE_HASH
cont>    ON EMPLOYEES (EMPLOYEE_ID)
cont>    STORE IN AREA_B
cont>    TYPE IS HASHED;
```

4.9.3 Same Area, with Placement Clause (One I/O Operation)

The best placement puts the EMPLOYEES rows in the same storage area as the hashed index, as shown in Example 4–18. This means that inserting a new row or retrieving a row by hashed index requires only one I/O operation. This placement is optimal, but can be achieved only by appropriately sizing the data page and correctly allocating the storage area to accommodate the data and hash buckets.

Section 4.8 discusses how to size data pages by showing you how to calculate the size of rows and hashed index structures, including overhead, that are to fit on a page, and then how to calculate the file allocation size based on the page size and number of rows to be stored in the storage area.

Example 4–18 Placing Rows and the Hashed Index in the Same Storage Area, Using the Placement Clause

```
SQL> CREATE STORAGE MAP EMP_MAP
cont>     FOR EMPLOYEES
cont>     STORE IN AREA_B
cont>     PLACEMENT VIA INDEX EMPLOYEE_HASH;
SQL> --
SQL> CREATE UNIQUE INDEX EMPLOYEE_HASH
cont>    ON EMPLOYEES (EMPLOYEE_ID)
cont>    STORE IN AREA_B
cont>    TYPE IS HASHED;
```

4.9.4 Clustering: Add Child Rows, Separate Storage Area, with Placement Clause

Adding child rows adds a second level of data. The JOB_HISTORY child rows are related to the EMPLOYEES parent rows by the EMPLOYEE_ID column. This means that the hash algorithm locates the hashed index buckets on the same page.

The parent and child rows are stored in a mixed page format storage area, AREA_A, which is separate from the hashed index storage area, AREA_B, as shown in Example 4–19. The page size for the two storage areas can be defined to accommodate the relevant rows and hashed index structures. The rows of the EMPLOYEES and JOB_HISTORY tables may take up a lot of space on the page. The storage area AREA_A page size may be quite large, and therefore may not be appropriate for storing the hashed index, so the hashed indexes are stored in the AREA_B storage area.

Access by hashed indexes for both the EMPLOYEES and JOB_HISTORY tables requires two I/O operations: one to access the index in storage area AREA_B and one to access the row in storage area AREA_A. In the best case, subsequent access by EMPLOYEE_ID for a JOB_HISTORY row is read directly from memory, requiring zero I/O operations.

Example 4–19 Placing Parent and Child Rows in One Storage Area, Hashed Indexes in a Separate Area, Using the Placement Clause

```
SQL> CREATE STORAGE MAP EMP_MAP
cont>    FOR EMPLOYEES
cont>    STORE IN AREA_A
cont>    PLACEMENT VIA INDEX EMPLOYEE_HASH;
SQL>
```

(continued on next page)

Example 4–19 (Cont.) Placing Parent and Child Rows in One Storage Area, Hashed Indexes in a Separate Area, Using the Placement Clause

```
SQL> CREATE UNIQUE INDEX EMPLOYEE_HASH
cont>    ON EMPLOYEES (EMPLOYEE_ID)
cont>    STORE IN AREA_B
cont>    TYPE IS HASHED;
SQL>
SQL> CREATE STORAGE MAP JOB_HISTORY_MAP
cont>    FOR JOB_HISTORY
cont>    STORE IN AREA_A
cont>    PLACEMENT VIA INDEX JOB_HISTORY_HASH;
SQL>
SQL> CREATE INDEX JOB_HISTORY_HASH
cont>    ON JOB_HISTORY (EMPLOYEE_ID)
cont>    STORE IN AREA_B
cont>    TYPE IS HASHED;
```

4.9.5 Shadowing: Child and Parent in Separate Areas, with Placement Clause

When you shadow child and parent rows, the parent rows of the EMPLOYEES table are placed in storage area AREA_B clustered around the hash bucket as shown in Example 4–20. The child rows of the JOB_HISTORY table are stored in a separate storage area, AREA_A. This storage scheme has the same performance characteristics as clustering in that access by hashed indexes for both the EMPLOYEES and JOB_HISTORY tables requires two I/O operations: one to access the index in the AREA_B storage area and one to access the JOB_HISTORY row in the AREA_A storage area. However, if the AREA_A storage area is a uniform area, sequential access to JOB_HISTORY rows performs much better with the rows still distributed similarly to the parent rows.

This storage scheme, called shadowing, is a good strategy when you have a small amount of data (EMPLOYEES) with a large number of related child rows (JOB_HISTORY) that are inserted randomly over a period of time. Shadowing means that the transactions will be grouped together, and if enough space is allocated in storage area AREA_A, space will be available to maintain the clustering effect.

Example 4–20 Placing Parent Rows and Hashed Indexes in the Same Storage Area, Child Rows in a Separate Storage Area, Using the Placement Clause

```
SQL> CREATE STORAGE MAP EMP_MAP
cont>    FOR EMPLOYEES
cont>    STORE IN AREA_B
cont>    PLACEMENT VIA INDEX EMPLOYEE_HASH;
SQL>
SQL> CREATE UNIQUE INDEX EMPLOYEE_HASH
cont>    ON EMPLOYEES (EMPLOYEE_ID)
cont>    STORE IN AREA_B
cont>    TYPE IS HASHED;
SQL>
SQL> CREATE STORAGE MAP JOB_HISTORY_MAP
cont>    FOR JOB_HISTORY
cont>    STORE IN AREA_A
cont>    PLACEMENT VIA INDEX JOB_HISTORY_HASH;
SQL>
SQL> CREATE INDEX JOB_HISTORY_HASH
cont>    ON JOB_HISTORY (EMPLOYEE_ID)
cont>    STORE IN AREA_B
cont>    TYPE IS HASHED;
```

4.9.6 Clustering: Child and Parent Rows and Hashed Index All in the Same Area, with Placement Clause

Clustering parent and child rows and their hashed indexes all in the same storage area as shown in Example 4–21 attains the best performance, but is the hardest to achieve in practice. All data and hash bucket information is stored on the same page, or at least on adjacent pages if there are many rows in the JOB_HISTORY table.

Only a single I/O operation is required to fetch an EMPLOYEES row and all its related JOB_HISTORY rows.

You must determine the page size carefully so that the page can contain the one system record, both hash buckets, the EMPLOYEES row, and all JOB_HISTORY rows (with perhaps some overflow to adjacent pages). Some extra space is necessary to account for some collisions (data values hashing to the same page), to allow for duplicate node index records, and to store EMPLOYEES and JOB_HISTORY rows.

Example 4–21 Placing Parent and Child Rows and Hashed Indexes in the Same Storage Areas, Using the Placement Clause

```
SQL> CREATE STORAGE MAP EMP_MAP
cont>    FOR EMPLOYEES
cont>    STORE IN AREA_B
cont>    PLACEMENT VIA INDEX EMPLOYEE_HASH;
SQL>
SQL> CREATE UNIQUE INDEX EMPLOYEE_HASH
cont>    ON EMPLOYEES (EMPLOYEE_ID)
cont>    STORE IN AREA_B
cont>    TYPE IS HASHED;
SQL>
SQL> CREATE STORAGE MAP JOB_HISTORY_MAP
cont>    FOR JOB_HISTORY
cont>    STORE IN AREA_B
cont>    PLACEMENT VIA INDEX JOB_HISTORY_HASH;
SQL>
SQL> CREATE INDEX JOB_HISTORY_HASH
cont>     ON JOB_HISTORY (EMPLOYEE_ID)
cont>     STORE IN AREA_B
cont>    TYPE IS HASHED;
```

5 Implementing a Multischema Database

Using SQL, you can create a multischema database. A **multischema database** is a database that contains one or more schemas organized within one or more catalogs. When you have more than one schema in a database, you can join data from tables in the different schemas.

This chapter describes how to create a multischema database, including how to create catalogs, schemas, and schema elements.

You can create the sample corporate_data multischema database by invoking the personnel script in the sample directory.

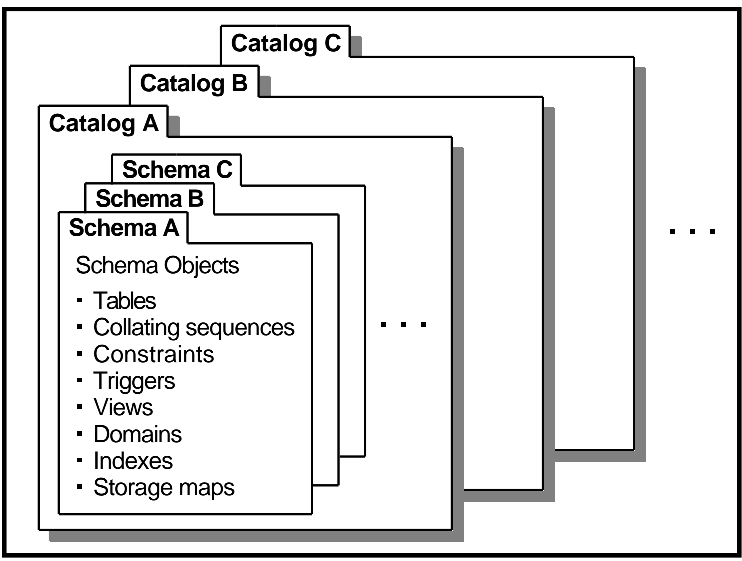
Before you read this chapter, read Chapter 3, which explains how to use the data definition language to create a database.

5.1 Understanding Multischema Databases

A database can contain one or more schemas organized within one or more catalogs. A **schema** consists of metadata definitions such as tables, views, constraints, collating sequences, indexes, storage maps, triggers, and the privileges for each of these. A **catalog** is a group of schemas within one database.

Figure 5–1 illustrates that a multischema database can contain more than one catalog and a catalog can contain more than one schema.

Figure 5–1 Multischema Database with Multiple Catalogs and Schemas



5.2 Creating Multischema Databases

To create a multischema database, you use the MULTISCHEMA IS ON clause of the CREATE DATABASE statement. The CREATE DATABASE statement shown in Example 5–1 creates a single-file, multischema database.

Example 5–1 Creating a Multischema Database

```
SQL> CREATE DATABASE FILENAME 'corporate_data_test'
cont>        MULTISCHEMA IS ON;
```

As with any CREATE DATABASE statement, the statement can contain definitions for all the database elements, or it can specify only the database. Because the statement shown in Example 5–1 does not specify any physical characteristics, the database uses the defaults for the physical characteristics.

5.3 Creating Catalogs

After you define the physical characteristics for the database, you use the CREATE CATALOG statement to define one or more catalogs. (When you create a multischema database, SQL automatically creates the system catalog, RDB$CATALOG, which holds the system schema, RDB$SCHEMA.)

Example 5–2 shows how to create the ADMINISTRATION catalog.

Example 5–2 Creating a Catalog

```
SQL> -- The system catalogs and schemas are already defined.
SQL> SHOW CATALOGS
Catalogs in database with filename corporate_data_test
    RDB$CATALOG
SQL> SHOW SCHEMAS
Schemas in database with filename corporate_data_test
    RDB$SCHEMA
SQL> -- Create a catalog.
SQL> CREATE CATALOG ADMINISTRATION;
```

The CREATE CATALOG statement can contain definitions for subordinate elements such as schemas and tables. When it does, the statement contains only one semicolon (;), which is at the end of the statement. Definitions for the subordinate elements do not end with a semicolon when contained in a CREATE CATALOG statement.

5.4 Creating Schemas

You can create one or more schemas in addition to the system schema that SQL creates. When you use the CREATE SCHEMA statement as a separate statement, you must specify which catalog will contain the schema. You specify the catalog by using the SET CATALOG statement.

Example 5–3 shows how to create a schema in the ADMINISTRATION catalog.

Example 5–3 Creating a Schema

```
SQL> -- Specify the catalog.
SQL> SET CATALOG 'ADMINISTRATION'
SQL> --
SQL> -- Create the schema.
SQL> CREATE SCHEMA PERSONNEL;
```

(continued on next page)

Example 5–3 (Cont.) Creating a Schema

```
SQL> SHOW SCHEMAS
Schemas in database with filename corporate_data_test
    PERSONNEL
    RDB$CATALOG.RDB$SCHEMA
SQL>
```

When the CREATE SCHEMA statement is part of a CREATE CATALOG or CREATE DATABASE statement, SQL implicitly qualifies the name of the schema with the name of the catalog preceding it. Section 5.5 explains when and how you qualify names in multischema databases.

The CREATE SCHEMA statement can contain definitions for subordinate elements such as domains and tables. When it does, the statement contains only one semicolon (;), which is at the end of the statement. Definitions for the subordinate elements do not end with a semicolon when contained in a CREATE SCHEMA statement.

Example 5–4 shows how to use one statement to create a schema with subordinate elements.

Example 5–4 Creating a Schema with Subordinate Elements

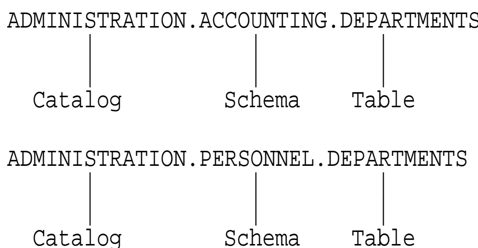
```
SQL> -- Create the schema.
SQL> CREATE SCHEMA RECRUITING
cont> --
cont> -- Create the domains.
cont> CREATE DOMAIN ID CHAR(5)
cont> CREATE DOMAIN NAME CHAR(20)
cont> CREATE DOMAIN STATUS_CODE CHAR(1)
cont> --
cont> -- Create a table.
cont> CREATE TABLE CANDIDATES
cont>        (CANDIDATE_ID       ID
cont>            CONSTRAINT      CANDIDATE_ID_NOT_NULL
cont>            PRIMARY KEY   NOT DEFERRABLE,
cont>         LAST_NAME          NAME
cont>            CONSTRAINT      CANDIDATES_LAST_NAME_NOT_NULL
cont>            NOT NULL      NOT DEFERRABLE,
```

```
cont>        FIRST_NAME         NAME,
cont>        MIDDLE_INITIAL     CHAR(1),
cont>        CANDIDATE_STATUS   STATUS_CODE
cont>         );
```

You can also use separate statements to create a schema and its subordinate elements. The following sections explain how you create schema elements and specify the schema and catalog in which the elements are contained.

5.5 Naming Elements

In a multischema database, elements in different schemas or catalogs can have the same element name. SQL differentiates between the elements by qualifying the subordinate element names with the names of the catalogs and schemas that contain them. For example, in the sample corporate_data database, there are two DEPARTMENTS tables in different schemas. SQL qualifies the table names with the schema name and the catalog name, as the following illustration shows:



When you create a database element such as a schema, table, or domain, you must specify, either implicitly or explicitly, the names of the elements that contain the new element. You specify the names in the following ways:

- If you create an element such as a catalog or schema, and in the *same* SQL statement, you create a subordinate element, SQL implicitly qualifies the name of the subordinate element with the name of the element that contains it. The following example demonstrates this:

```
SQL> CREATE CATALOG ADMIN
cont> CREATE SCHEMA TEST1;
SQL> --
SQL> SHOW SCHEMAS
Schemas in database with filename corporate_data_test
    ADMIN.TEST1
    PERSONNEL
    RECRUITING
    RDB$CATALOG.RDB$SCHEMA
SQL>
```

SQL qualifies the schema name TEST1 with the catalog name ADMIN.

- You use the SET CATALOG or SET SCHEMA statements to specify the current catalog or schema. In the following example, the schema PERS is created in the catalog ADMIN:

```
SQL> SET CATALOG 'ADMIN'
SQL> CREATE SCHEMA PERS;
```

- You qualify the name of the element. The following example creates a schema in the catalog RDB$CATALOG:

```
SQL>  CREATE SCHEMA RDB$CATALOG.ACCOUNT;
```

5.5.1 Using Qualified Names

As Section 5.5 explains, when you create an element, you must qualify the name with the names of the elements that contain the new element. Similarly, when you refer to elements in a multischema database, you must qualify the name of the subordinate element with the names of the elements that contain it.

If you do not qualify the name of a schema, SQL implicitly qualifies it with the name of the current catalog. (If you have not used a SET CATALOG statement, the current catalog is the default catalog, RDB$CATALOG.) If the schema is not contained in the current catalog, you get an error, as the following example shows:

```
SQL> SET CATALOG 'RDB$CATALOG'
SQL> DROP SCHEMA RECRUITING;
%SQL-F-SCHNOTDEF, Schema RECRUITING is not defined
```

If you do not qualify the name of a schema element (that is, an element contained in a schema), SQL implicitly qualifies it with the current authorization identifier. An **authorization identifier** is a name that identifies the definer of a schema. The following example shows what happens when you do not qualify the name of a schema element:

```
SQL> SELECT * FROM DEPARTMENTS;
%SQL-F-SCHNOTDEF, Schema GREMBOWSKI is not defined
```

In the previous example, GREMBOWSKI is the operating system user name of the definer of the schema. See the *Oracle Rdb7 Guide to SQL Programming* for more information about authorization identifiers.

5.5.2 Using Stored Names and SQL Names

When you use SQL with multischema enabled, use names of the database elements qualified by the elements that contain them.

When you use SQL with multischema disabled or when you use interfaces other than SQL, use the stored names of the database elements. **Stored names** are nonqualified names assigned by Oracle Rdb. When more than one element has the same nonqualified name, Oracle Rdb appends a numeric suffix to the name.

Example 5–5 shows the differences between stored names and the names used by SQL.

Example 5–5 Displaying Stored Names

```
SQL> --  With multischema enabled, you see the qualified SQL names.
SQL> --
SQL> ATTACH 'FILENAME corporate_data MULTISCHEMA IS ON';
SQL> SHOW TABLES
User tables in database with filename corporate_data
     ADMINISTRATION.ACCOUNTING.DAILY_HOURS
     ADMINISTRATION.ACCOUNTING.DEPARTMENTS
     ADMINISTRATION.ACCOUNTING.PAYROLL
     ADMINISTRATION.ACCOUNTING.WORK_STATUS
     ADMINISTRATION.PERSONNEL.CURRENT_INFO
                                A view.
     ADMINISTRATION.PERSONNEL.CURRENT_JOB
                                A view.
     ADMINISTRATION.PERSONNEL.CURRENT_SALARY
                                A view.
     ADMINISTRATION.PERSONNEL.DEPARTMENTS
   .
   .
   .
SQL> DISCONNECT DEFAULT;
SQL> --
SQL> -- Attach to the database, disabling multischema.
SQL> -- Notice that Oracle Rdb appends a "1" to the stored name of
SQL> -- one of the DEPARTMENTS table to ensure that all table
SQL> -- names are unique.
SQL> --
SQL> ATTACH 'FILENAME corporate_data MULTISCHEMA IS OFF';
```

(continued on next page)

Example 5–5 (Cont.) Displaying Stored Names

```
SQL> SHOW  TABLES
User tables in database with filename corporate_data
     CANDIDATES
     COLLEGES
     CURRENT_INFO                    A view.
     CURRENT_JOB                     A view.
     CURRENT_SALARY                  A view.
     DAILY_HOURS
     DEGREES
     DEPARTMENTS ➊
     DEPARTMENTS1 ➋
   .
   .
   .
SQL> DISCONNECT DEFAULT;
```

In Example 5–5, the callouts have the following meaning:

➊ Labels the stored name for the table ADMINISTRATION.PERSONNEL.DEPARTMENTS.
➋ Labels the stored name for the table ADMINISTRATION.ACCOUNTING.DEPARTMENTS.

When you create an element, you can specify a stored name rather than letting Oracle Rdb assign it a stored name. Example 5–6 shows how you specify stored names.

Example 5–6 Specifying Stored Names

```
SQL> ATTACH 'FILENAME corporate_data_test MULTISCHEMA IS ON';
SQL> --
SQL> SET SCHEMA 'ADMIN.PERS';
SQL> CREATE DOMAIN TEST_DOM
cont>    STORED NAME IS TEST_EXTRNL_DOM
cont>    CHAR(5);
SQL> --
SQL> SHOW DOMAIN TEST_DOM
TEST_DOM                        CHAR(5)
    Stored name is TEST_EXTRNL_DOM
SQL> --
SQL> COMMIT;
```

5.6 Using Aliases

When you create a database or when you otherwise attach to a database, you can use an alias for the database name. SQL recognizes the database by its alias during that attach to the database.

When you use an alias to attach to a database, you must qualify catalog and schema names with the alias in subsequent statements. To do this, you enclose the alias and catalog name in a set of double (") quotation marks.

Note that, by default, the Oracle Rdb implementation of SQL considers strings enclosed in double quotation marks to be string literals, but the ANSI/ISO standard for SQL considers strings enclosed in (or delimited by) double quotation marks to be **delimited identifiers**. To take advantage of the ANSI/ISO standard, you must use the SET QUOTING RULES 'SQL92' or SET DIALECT 'SQL92' statement before you issue any statements. These statements let you use double quotation marks around the alias, catalog, and schema names so that you can delimit the element names. (Consider including the SET QUOTING RULES 'SQL92' or SET DIALECT 'SQL92' statement in your SQL initialization file, sqlini.sql.)

Example 5–7 illustrates the use of an alias when you are creating a database.

Example 5–7 Using an Alias

```
SQL> -- Create the database and use an alias.
SQL> --
SQL>  CREATE DATABASE ALIAS MSCH_DB FILENAME TEST1
cont>     MULTISCHEMA IS ON;
SQL> --
SQL> -- You cannot refer to a catalog without using the alias.
SQL> --
SQL> CREATE CATALOG ADMINISTRATION;
%SQL-F-NODEFDB, There is no default database
SQL> --
SQL> -- By default, SQL considers double quotation marks to be a deprecated
SQL> -- feature used to enclose string literals.
SQL> --
SQL> CREATE CATALOG "MSCH_DB.ADMINISTRATION"
%SQL-I-DEPR_FEATURE, Deprecated Feature: " used instead of ' for string literal
CREATE CATALOG "MSCH_DB.ADMINISTRATION"
                 ^
%SQL-W-LOOK_FOR_STT, Syntax error, looking for:
%SQL-W-LOOK_FOR_CON,            name,
%SQL-F-LOOK_FOR_FIN,    found MSCH_DB.ADMINISTRATION instead
SQL> --
```

(continued on next page)

Example 5–7 (Cont.) Using an Alias

```
SQL> -- To use double quotation marks to delimit an identifier, use the
SQL> -- SET QUOTING RULES 'SQL92' or SET DIALECT 'SQL92' statement.
SQL> -- Otherwise, SQL treats the identifier as a string literal and
SQL> -- gives you an error message as shown in the previous statement.
SQL> --
SQL> SET QUOTING RULES 'SQL92';
SQL> CREATE CATALOG "MSCH_DB.ADMINISTRATION";
SQL> --
SQL> -- When you refer to a schema, specify the alias and the catalog
SQL> -- name.
SQL> --
SQL> CREATE SCHEMA "MSCH_DB.ADMINISTRATION".PERS;
SQL> --
SQL> -- When you refer to a domain or a table, specify the alias name, the
SQL> -- catalog name, and the schema name.
SQL> --
SQL>  CREATE DOMAIN "MSCH_DB.ADMINISTRATION".PERS.ID_DOM CHAR (5);
SQL> --
SQL> -- You can specify the catalog and schema names, and implicitly the
SQL> -- alias, by using the SET CATALOG and SET SCHEMA statements.
SQL> --
SQL> SET CATALOG 'ADMINISTRATION'
SQL> SET SCHEMA 'PERS'
```

As Example 5–7 shows, when you refer to other subordinate elements such as domain and table names, you must qualify them with the alias, catalog, and schema names.

5.7 Creating Schema Elements

Creating schema elements in multischema databases is similar to creating those elements in a single schema database. When you create the elements, you must specify which catalog and schema will contain the element, as explained in Section 5.5. If you refer to elements in other catalogs or schemas, you must qualify the names of those elements, as explained in Section 5.5.1.

Note that you can create more than one element with the same name if you create them in different catalogs and schemas.

The following examples show how to create some of the schema elements in a multischema database.

Example 5–8 shows how to create domains in the PERSONNEL schema.

Example 5–8 Creating Domains in Multischema Databases

```
SQL> ATTACH 'FILENAME corporate_data_test MULTISCHEMA IS ON';
SQL> --
SQL> -- Set the catalog and schema.
SQL> --
SQL> SET CATALOG 'ADMINISTRATION'
SQL> SET SCHEMA 'PERSONNEL'
SQL> --
SQL> -- Create the domains for the PERSONNEL schema.
SQL> --
SQL> CREATE DOMAIN ID CHAR(5);
SQL> CREATE DOMAIN NAME CHAR(20);
SQL> CREATE DOMAIN MIDDLE_INITIAL CHAR(1)
cont>       DEFAULT ' '
cont>       EDIT STRING IS 'X';
SQL> CREATE DOMAIN ADDRESS_LINE CHAR(25);
SQL> CREATE DOMAIN STATE_CODE CHAR (4);
SQL> CREATE DOMAIN POSTAL_CODE CHAR(5);
SQL> CREATE DOMAIN SALARY INTEGER(2)
cont>       EDIT STRING IS '$$$$,$$9.99';
SQL> CREATE DOMAIN STATUS_CODE CHAR(1);
SQL> --
SQL> -- Specify a stored name for a domain.
SQL> --
SQL> CREATE DOMAIN CODE STORED NAME IS PERS_CODE CHAR(4);
```

When you create a table in one schema, you can refer to domains from other schemas.

Example 5–9 shows how to create tables in a multischema database and how to refer to elements in the database.

Example 5–9 Creating Tables That Refer to Objects in Other Schemas

```
SQL> SET CATALOG 'ADMINISTRATION'
SQL> SET SCHEMA 'PERSONNEL'
SQL> -- Create the EMPLOYEES table.
SQL> --
SQL> CREATE TABLE EMPLOYEES
cont>      (EMPLOYEE_ID            ID
cont>          PRIMARY KEY  DEFERRABLE,
cont>       LAST_NAME              NAME,
cont>       FIRST_NAME             NAME,
cont>       MIDDLE_INITIAL         MIDDLE_INITIAL,
cont>       ADDRESS_DATA_1         ADDRESS_LINE,
```

(continued on next page)

Example 5–9 (Cont.) Creating Tables That Refer to Objects in Other Schemas

```
cont>       ADDRESS_DATA_2          ADDRESS_LINE,
cont>       CITY                    NAME,
cont>       STATE                   STATE_CODE,
cont>       ZIP_CODE                POSTAL_CODE,
cont>       SEX                     CHAR(1),
cont>          CONSTRAINT            EMP_SEX_VALUES
cont>          CHECK                   (SEX in ('M', 'F'))
cont>          DEFERRABLE,
cont>       BIRTHDAY                DATE,
cont>       STATUS                  STATUS_CODE,
cont>          CONSTRAINT            STATUS_CODE_VALUES
cont>          CHECK                 (STATUS in ('0','1','2','N') )
cont>          DEFERRABLE
cont>      );
SQL> --
SQL> -- Specify the RECRUITING schema.
SQL> SET SCHEMA 'RECRUITING'
SQL> --
SQL> -- Create a table that refers to domains in the PERSONNEL schema.
SQL> --
SQL> CREATE TABLE COLLEGES
cont>      (COLLEGE_CODE            PERSONNEL.CODE
cont>          PRIMARY KEY DEFERRABLE,
cont>       CITY                    PERSONNEL.NAME,
cont>       STATE                   PERSONNEL.STATE_CODE,
cont>       ZIP_CODE                PERSONNEL.POSTAL_CODE
cont>      );
SQL> --
SQL> -- Create a table with constraints that refer to columns
SQL> -- in the PERSONNEL schema.
SQL> --
SQL> CREATE TABLE DEGREES
cont>   (CANDIDATE_ID            ID
cont>       REFERENCES  CANDIDATES (CANDIDATE_ID) DEFERRABLE,
cont>    EMPLOYEE_ID          PERSONNEL.ID
cont>       -- The constraint refers to a column in the PERSONNEL schema.
cont>       REFERENCES  PERSONNEL.EMPLOYEES (EMPLOYEE_ID) DEFERRABLE,
cont>    COLLEGE_CODE         PERSONNEL.CODE
cont>         -- The constraint refers to a column in the current schema.
cont>       REFERENCES COLLEGES (COLLEGE_CODE) DEFERRABLE,
```

(continued on next page)

Example 5–9 (Cont.) Creating Tables That Refer to Objects in Other Schemas

```
cont>    YEAR_GRADUATED       SMALLINT,
cont>    DEGREE               CHAR(3),
cont>       CONSTRAINT  DEGREE_VAL
cont>       CHECK    (DEGREE in ('BA','BS','MA','MS','AA','PhD') )
cont>       DEFERRABLE,
cont>    DEGREE_FIELD         CHAR(15)
cont>   );
```

Example 5–10 shows the definition for the CURRENT_INFO view in the corporate_data database.

Example 5–10 Creating Views That Refer to Tables in Other Schemas

```
SQL>  CREATE VIEW CURRENT_INFO
cont>           (LAST_NAME,     FIRST_NAME,
cont>            ID,             DEPARTMENT,
cont>            JOB,            JSTART,
cont>            SSTART,         SALARY)
cont>        AS SELECT
cont>            E.LAST_NAME,          E.FIRST_NAME,
cont>            E.EMPLOYEE_ID,        D.DEPARTMENT_NAME,
cont>            P.JOB_TITLE,          JH.JOB_START,
cont>            SH.SALARY_START,      SH.SALARY_AMOUNT
cont> --
cont> -- Refer to tables from the PERSONNEL schema.
cont>        FROM  PERSONNEL.EMPLOYEES E,
cont>              PERSONNEL.JOB_HISTORY JH,
cont> --
cont> -- Refer to tables from the ACCOUNTING schema.
cont>              ACCOUNTING.DEPARTMENTS D,
cont>              ACCOUNTING.PAYROLL P,
cont> --
cont> -- Refer to tables from the PERSONNEL schema.
cont>              PERSONNEL.SALARY_HISTORY SH
```

(continued on next page)

Example 5–10 (Cont.) Creating Views That Refer to Tables in Other Schemas

```
cont>        WHERE JH.DEPARTMENT_CODE = D.DEPARTMENT_CODE
cont>              and JH.JOB_CODE = P.JOB_CODE
cont>              and E.EMPLOYEE_ID = SH.EMPLOYEE_ID;
```

Example 5–11 shows how to create triggers in a multischema database.

Example 5–11 Creating Triggers That Refer to Objects in Other Schemas

```
SQL>  CREATE TRIGGER STATUS_CODE_CASCADE_UPDATE
cont> --
cont> -- Refer to a column in the ACCOUNTING schema.
cont>    BEFORE UPDATE OF STATUS_CODE ON ACCOUNTING.WORK_STATUS
cont>      REFERENCING OLD AS OLD_WORK_STATUS
cont>                  NEW AS NEW_WORK_STATUS
cont> --
cont> -- Refer to a column in the PERSONNEL schema.
cont>    (UPDATE PERSONNEL.EMPLOYEES
cont>            SET STATUS = NEW_WORK_STATUS.STATUS_CODE
cont>            WHERE STATUS = OLD_WORK_STATUS.STATUS_CODE)
cont>    FOR EACH ROW;
```

In addition to the elements shown in this chapter, you can create any other schema elements, such as indexes or stored procedures, in a multischema database.

6 Loading Data

This chapter shows you how to efficiently load data into an Oracle Rdb database and describes the interaction between load operations and query performance. The chapter assumes that you have read the previous chapters and that you have designed a database and created it using Oracle Rdb.

Oracle Rdb provides two ways to load data from existing data files into a database:

- Using a high-level language program that reads rows or records and uses the SQL INSERT statement to store them in the database.
- Using the RMU Load command. The RMU Load command reads sequential flat files and specially structured unload (.unl) files created by the RMU Unload command. You can use a single process or a multiprocess method.

  The multiprocess method, also called **parallel load**, enables Oracle RMU to use your process to read the input file and use one or more executors (detached processes or subprocesses) to load the data into the target table. This results in concurrent read and write operations and, in many cases, substantially improves the performance of the load operation.

This chapter shows you how to load data using these methods and describes how to improve performance while loading data.

6.1 Improving Performance When Loading Data

How you load data into the database can affect the query performance of the database. The following general guidelines can help ensure an efficient load of large amounts of data into an Oracle Rdb database and help you understand any trade-offs between efficient load operations and subsequent query performance:

- To improve the performance during the load operation, use parallel load. When you use the RMU Load command, you can use a single-process or

multiprocess (parallel) load operation. For more information on parallel load, see Section 6.7.

- To improve query performance for retrieving ranges of values, use a clustering index. That is, use a sorted index to place presorted table rows in a storage area so that the rows are stored in approximately the same order in which they appear in the index. However, the performance of the load operation will be affected. See the Section 4.6.1 for more information on using clustering indexes for range retrieval.

  If you are using parallel load and hashed indexes, do not sort the data prior to loading it. Instead, use the Place qualifier to the RMU Load command to sort the data as it is loaded. (The Place qualifier is useful for hashed indexes, not sorted indexes.)

- If performance during range retrieval is not important (for example, if performance during exact match retrieval is more important), first load the data and then build the indexes to avoid poor load performance from index updates and B-tree rebalancing.

  As an alternative, you can use the Defer_Index_Updates qualifier of the RMU Load command to defer the building of secondary indexes until commit time. This can result in improved load performance. For more information, see Section 6.7.

- The presence of indexes adversely affects the performance of the load operation.

  Only the placement indexes (an index specified in the PLACEMENT VIA INDEX option of a storage map) or a sorted index used to cluster presorted rows for range retrieval performance should be defined before you perform the load operation.

- The presence of constraints, triggers, or COMPUTED BY columns adversely affects the performance of the load operation.

  Delete any triggers and create them again after the load operation completes. Alternatively, if you use the RMU Load command, you can use the Notrigger_Relations qualifier to temporarily disable triggers.

  Delete any constraints and create them again after the load operation completes. Alternatively, if you use the RMU Load command, you can use the Constraints=Deferred or Noconstraints qualifier. For more information about these qualifiers, see Section 6.6.1.

- To reduce database locking (locking requires virtual or physical memory) and to avoid snapshot file input/output (I/O) operations, store the rows using a transaction with exclusive share mode.

- To further reduce database locking, enable adjustable lock granularity.
- To reduce root file I/O operations, commit loading transactions after a large number of rows have been loaded.
- Define the database-wide parameters. Set the buffer size to match your page size. Disable snapshot files (or make them deferred) to eliminate or reduce snapshot file I/O operations. If you use the RMU Load command, you can use the Exclusive mode to eliminate snapshot I/O. For more information, see Section 6.3.1 and Section 6.6.1.
- Improperly calculating sizes of storage areas in an Oracle Rdb database adversely affects the performance. See Section 4.6.2 for more information on estimating the size of storage areas for holding table and hashed index structures.
- Increase the number of buffers available to the load process by defining the logical name RDM$BIND_BUFFERS or the configuration parameter RDB_BIND_BUFFERS. As an alternative, you can use the Buffer qualifier to the RMU Load command to define the number of buffers used by the load process. See Section 6.3.1 for more information.
- To reduce the number of stalls while waiting for writes to the disk to complete, use asynchronous batch-write operations. For more information about asynchronous batch-write operations, see the *Oracle Rdb7 SQL Reference Manual*.
- To improve I/O, consider placing the data files on different disks than the disks that hold the storage areas into which you are loading data. For example, if you are loading data into the storage areas pers_stor1.rda and pers_stor2.rda and they are located on the disks DISK1 and DISK2, do not place the data files on either of these disks. Place them on a separate disk.
- To improve I/O, consider placing the sortwork files and the .ruj files on different disks than the disks that hold the storage areas.
- Enable fast commit processing. When you enable fast commit processing, Oracle Rdb keeps updated pages in the buffer pool and does not write the pages to disk when a transaction commits. Instead, the updated pages remain in the buffer pool until a user-specified checkpoint is reached. You reduce page I/Os as well as I/Os to the .ruj file.
- If you are using only sorted indexes, use uniform format areas rather than mixed format areas because uniform format areas load more quickly.

- Enable detected asynchronous prefetch. With detected asynchronous prefetch enabled, Oracle Rdb prefetches all pages before they are needed for the load operation, reducing the time it takes to complete the operation.

When loading data using a PLACEMENT VIA INDEX clause (for a hashed index) in a storage map definition, there are many additional important guidelines to follow to ensure an efficient load operation. Check your application against the following guidelines because a minor error might lead to poor load performance and poor retrieval performance:

- If you are not using a parallel load operation, sort the data in database key (dbkey) order before you load it or as you load it. To do this, use one of the following methods:
  - Use the PLACEMENT ONLY RETURNING DBKEY clause of the INSERT statement to return the dbkey value where each row will be stored. This clause does not store any data or allocate any space for that data in the database. See Section 6.2.2 for more information about sorting the dbkeys.
  - Use the RMU Load command with the Place qualifier to build an ordered set of dbkeys and automatically store the rows in dbkey order in the database. For more information, see the Section 6.6.1.
- Define the storage area. Estimate the page size and file allocation size based on calculated data row and index structure sizes for all tables and indexes to be clustered together. If you cluster table data and a hashed index, define a mixed page format for the storage area. If you partition tables across multiple storage areas, determine the number of storage areas and their size carefully. For more information, see Section 6.3.2.
- Define the hashed index. There should be no null values in the columns selected for the index. Specify whether or not duplicate rows are allowed. Use the STORE USING clause to refer to the key on which to store the index. If you partition the index across multiple storage areas, specify the storage area names and their storage limit values in the WITH LIMIT OF clause.

  For optimal performance, consider using only one column as the partitioning criteria and, if possible, making that column an unscaled integer column.
- Define the storage maps. Specify the index name in the PLACEMENT VIA INDEX clause by which rows are to be placed. If you enable row compression, be sure to estimate row sizes on the uncompressed row size. If you cluster table data and indexes together, be sure to specify STORE USING clauses that are identical to those of your index definitions. If you

partition the tables across multiple storage areas, specify storage area names and their WITH LIMIT OF values that are identical to those of your index definitions. See Chapter 4 for information on page and row sizes and clustering data.

For more information about improving the performance of the RMU Load command, see Section 6.6.1.

6.2 Using the PLACEMENT ONLY RETURNING DBKEY Clause

By using the PLACEMENT ONLY RETURNING DBKEY clause of the SQL INSERT statement and sorting the dbkeys prior to loading the database, you can greatly improve performance of bulk load programs. The PLACEMENT ONLY clause causes the INSERT statement to return the dbkey where each row will be stored, but does not store any data or allocate any space for that data in the database. SQL returns the targeted dbkey in a host language variable.

Note

Do not sort the data when you are using a parallel load operation.

Use the PLACEMENT ONLY clause *only* in SQL programs that load data into a database in which rows are placed via a hashed index (using the PLACEMENT VIA INDEX clause). Additionally, before loading the database, you should determine whether or not the area provides sufficient space. This process reduces the I/O that normally occurs with the PLACEMENT VIA INDEX option when you load the database.

When a storage map includes the PLACEMENT VIA INDEX clause for a hashed index, Oracle Rdb uses a hashing algorithm to determine placement. The hashing algorithm randomly assigns data to target pages in the database, taking the key value and assigning a dbkey target. The dbkey contains three components: the logical area number (important if the data and index are partitioned), the page number, and the line number (this is set to –1 for all rows that are placed with the INSERT statement). The hashing algorithm tends to generate page numbers randomly, even when the records are sorted by their key values. Simply reading the data file and loading from it would result in excessive random I/O; that is, multiple I/Os to the same database pages would be required. It is beneficial to store all rows going to the same database page at one time, so that the page requires updating only once during the load operation. By using the PLACEMENT ONLY clause, sorting the resulting data file, and then loading the database from the sorted data, significant performance improvement results.

The INSERT statement should assign actual values to the columns used in the index referred to by the PLACEMENT VIA INDEX clause. Otherwise, the columns default to null values and the dbkey that SQL returns is identical for all rows.

To effectively use the PLACEMENT ONLY RETURNING DBKEY clause of the INSERT statement, apply the following strategy:

1. Pass the data to the database using the INSERT statement with the PLACEMENT ONLY RETURNING DBKEY clause to get the target dbkey values.
2. Sort the dbkeys in ascending order.
3. Pass the same data to the database, this time in ascending dbkey order using the INSERT statement without the PLACEMENT ONLY clause. In this step, SQL actually stores the data.

Sections 6.2.1, 6.2.2, and 6.2.3 demonstrate one implementation of this strategy.

6.2.1 Using the INSERT Statement to Get the Dbkey for Each Row

Get the dbkey of each row to be stored using the INSERT statement as shown in the Pascal program in Example 6–1. This example reads the data from an Oracle Rdb database. However, the data could be read from a flat file.

Example 6–1 Using the PLACEMENT ONLY Clause to Extract Dbkeys from a Table

```
program PLACEMENT (data);

{
| This program unloads the row (including the NULL indicators) and the target
| row dbkey used for presorting.
}

exec sql include sqlca;
exec sql declare sfdb alias filename personnel;
exec sql declare mfdb alias filename mf_personnel;
{ Declare variables.}

type
    emp_record = record

{ Saved dbkey from PLACEMENT for sorting. }

        db_key: packed array [1..8] of char;
```

(continued on next page)

Example 6–1 (Cont.) Using the PLACEMENT ONLY Clause to Extract Dbkeys from a Table

```
{ data portion }

        employee_id: packed array [1..5] of char;
        last_name: packed array [1..14] of char;
        first_name: packed array [1..10] of char;
        middle_initial: packed array [1..1] of char;
        address_data_1: packed array [1..25] of char;
        address_data_2: packed array [1..20] of char;
        city: packed array [1..20] of char;
        state: packed array [1..2] of char;
        postal_code: packed array [1..5] of char;
        sex: packed array [1..1] of char;
        birthday: SQL$DATE;
        status_code: packed array [1..1] of char;

{ Null indicators }

        employee_id_ind: SQL$INDICATOR;
        last_name_ind: SQL$INDICATOR;
        first_name_ind: SQL$INDICATOR;
        middle_initial_ind: SQL$INDICATOR;
        address_data_1_ind: SQL$INDICATOR;
        address_data_2_ind: SQL$INDICATOR;
        city_ind: SQL$INDICATOR;
        state_ind: SQL$INDICATOR;
        postal_code_ind: SQL$INDICATOR;
        sex_ind: SQL$INDICATOR;
        birthday_ind: SQL$INDICATOR;
        status_code_ind: SQL$INDICATOR;
    end;

var
    emp_row: emp_record;
    data: file of emp_record;

begin

{ Start the transactions, both READ ONLY to avoid locking. }

exec sql set transaction
              on sfdb using (read only)
              and
              on mfdb using (read only);

{
| Open the output flat file.
}

open(data, file_name := 'EMPLOYEES.DATA', history := NEW);
rewrite(data);
```

(continued on next page)

Example 6–1 (Cont.) Using the PLACEMENT ONLY Clause to Extract Dbkeys from a Table

```
{
| Process all rows in this table.
| After reading the row from the old database, pass the column values (and
| NULL indicators) to be inserted into the new database with PLACEMENT ONLY.
| This returns a target dbkey that can be used for sorting the data
| into target page order.
}

exec sql declare read_cursor
                 read only table cursor
                 for
                 select * from sfdb.employees;

exec sql open read_cursor;

while (sqlca.sqlcode = 0) do begin

{
| Save columns from this database row in the output buffer.
}

    exec sql
        fetch read_cursor into
            :emp_row.employee_id indicator :emp_row.employee_id_ind,
            :emp_row.last_name indicator :emp_row.last_name_ind,
            :emp_row.first_name indicator :emp_row.first_name_ind,
            :emp_row.middle_initial indicator :emp_row.middle_initial_ind,
            :emp_row.address_data_1 indicator :emp_row.address_data_1_ind,
            :emp_row.address_data_2 indicator :emp_row.address_data_2_ind,
            :emp_row.city indicator :emp_row.city_ind,
            :emp_row.state indicator :emp_row.state_ind,
            :emp_row.postal_code indicator :emp_row.postal_code_ind,
            :emp_row.sex indicator :emp_row.sex_ind,
            :emp_row.birthday indicator :emp_row.birthday_ind,
            :emp_row.status_code indicator :emp_row.status_code_ind;

    if (sqlca.sqlcode = 0) then begin

{
| Use the PLACEMENT ONLY clause of the INSERT statement to fetch
| the dbkey.
}

      exec sql insert into mfdb.employees
                 (employee_id,
                  last_name,
                  first_name,
                  middle_initial,
                  address_data_1,
                  address_data_2,
                  city,
```

(continued on next page)

Example 6–1 (Cont.) Using the PLACEMENT ONLY Clause to Extract Dbkeys from a Table

```
                  state,
                  postal_code,
                  sex,
                  birthday,
                  status_code)
               values (
                  :emp_row.employee_id indicator :emp_row.employee_id_ind,
                  :emp_row.last_name indicator :emp_row.last_name_ind,
                  :emp_row.first_name indicator :emp_row.first_name_ind,
                  :emp_row.middle_initial indicator :emp_row.middle_initial_ind,
                  :emp_row.address_data_1 indicator :emp_row.address_data_1_ind,
                  :emp_row.address_data_2 indicator :emp_row.address_data_2_ind,
                  :emp_row.city indicator :emp_row.city_ind,
                  :emp_row.state indicator :emp_row.state_ind,
                  :emp_row.postal_code indicator :emp_row.postal_code_ind,
                  :emp_row.sex indicator :emp_row.sex_ind,
                  :emp_row.birthday indicator :emp_row.birthday_ind,
                  :emp_row.status_code indicator :emp_row.status_code_ind)
               placement only
               returning dbkey into :emp_row.db_key; { return database key }

{
| Write the data buffer.
}

        data^ := emp_row;
        put(data);
    end;{if}

end;{while}

exec sql
    close read_cursor;

close(data);
exec sql commit;
exec sql disconnect default;

end.
```

6.2.2 Sorting the Dbkeys in Ascending Order

Sort the dbkeys in ascending order and store them in either a flat file or a database table. Dbkeys should be sorted by logical area, by page, and by line number.

Most host languages do not provide a QUADWORD data type; therefore, many applications save the dbkey in a TEXT field. Ensure that the sorting is done properly by using binary (not text) characteristics. The following example shows how to sort the key using an OpenVMS DCL command:

```
$ SORT/KEY=(POS:1,SIZ:8,BINARY,UNSIGNED) -
_$ EMPLOYEES.DATA EMPLOYEES_SORT.DATA
```

The storage area allocation or page size may be too small for the amount of data you want to store. For instance, too many rows may share the same target page. When data is hashed to a page that is full, a search begins for a nearby page on which to store the data. In some cases, this might be acceptable. However, usually this overflow causes poor performance during the load and during run-time retrieval. You can examine the sorted data to determine the maximum number of rows that map to the same page. From this value, you can calculate whether or not that number of rows will cause an overflow. If an overflow will be created, use the ALTER DATABASE statement to either increase the page size for the area involved or increase the area allocation.

If the rows are stored in an Oracle Rdb database, you can use the GROUP BY clause to analyze the distribution of rows placed across the storage area and determine if the storage area allocation or page size is sufficient. In particular, you can use a statement similar to the following to determine the number of rows that hash to a given page:

```
SQL> SELECT DB_KEY_VALUE, COUNT(*) FROM DBKEY_TABLE
cont>       GROUP BY DB_KEY_VALUE;
```

Alternatively, you can write a program that runs through the dbkeys (after the sort) and counts the number of rows on a page to see if they fit.

6.2.3 Reading the Rows in Sorted Order and Storing Them in the Database

You can now use the sorted data file to read the rows in ascending dbkey order and store them in the database. In your data loading program, this process requires a simple read and insert operation. Typically, you commit the rows periodically to avoid a very large .ruj file. The database should either have snapshot files deferred or disabled, or the load transaction should be in exclusive write mode to avoid snapshot file I/O operations.

Example 6–2 shows a simple Pascal load program that uses the sorted flat file created in Example 6–1 and Section 6.2.2.

Example 6–2 Loading the Sorted Data

```
program DATA_STORAGE (data);

{
| This program loads the data row (including the NULL information) into the
| target database.  The data is sorted by the target dbkey.
}

exec sql include sqlca;
exec sql declare mfdb alias filename mf_personnel;

{ Declare variables }

type
    emp_record = record

{ Saved dbkey from PLACEMENT for sorting }

        db_key: packed array [1..8] of char;

{ data portion }

        employee_id: packed array [1..5] of char;
        last_name: packed array [1..14] of char;
        first_name: packed array [1..10] of char;
        middle_initial: packed array [1..1] of char;
        address_data_1: packed array [1..25] of char;
        address_data_2: packed array [1..20] of char;
        city: packed array [1..20] of char;
        state: packed array [1..2] of char;
        postal_code: packed array [1..5] of char;
        sex: packed array [1..1] of char;
        birthday: SQL$DATE;
        status_code: packed array [1..1] of char;

{ Null indicators }

        employee_id_ind: SQL$INDICATOR;
        last_name_ind: SQL$INDICATOR;
        first_name_ind: SQL$INDICATOR;
        middle_initial_ind: SQL$INDICATOR;
        address_data_1_ind: SQL$INDICATOR;
        address_data_2_ind: SQL$INDICATOR;
        city_ind: SQL$INDICATOR;
        state_ind: SQL$INDICATOR;
        postal_code_ind: SQL$INDICATOR;
        sex_ind: SQL$INDICATOR;
        birthday_ind: SQL$INDICATOR;
        status_code_ind: SQL$INDICATOR;
    end;

var
    emp_row: emp_record;
    data: file of emp_record;
```

(continued on next page)

Example 6–2 (Cont.) Loading the Sorted Data

```
begin

{ Start the transactions in EXCLUSIVE mode to avoid
  snapshot I/O operations. }

exec sql
    set transaction on mfdb
        using (read write reserving
               mfdb.employees for exclusive write);

{ Open the input data file. }

open(data, file_name := 'EMPLOYEES_SORT.DATA', history := OLD);
reset(data);

{ Process all records in the data file. }

while not EOF(data) do begin

    { Assign the row to the buffer for use in the INSERT statement. }
    emp_row := data^;

    exec sql
        insert into mfdb.employees
                   (employee_id,
                    last_name,
                    first_name,
                    middle_initial,
                    address_data_1,
                    address_data_2,
                    city,
                    state,
                    postal_code,
                    sex,
                    birthday,
                    status_code)
          values (
                  :emp_row.employee_id indicator :emp_row.employee_id_ind,
                  :emp_row.last_name indicator :emp_row.last_name_ind,
                  :emp_row.first_name indicator :emp_row.first_name_ind,
                  :emp_row.middle_initial indicator :emp_row.middle_initial_ind,
                  :emp_row.address_data_1 indicator :emp_row.address_data_1_ind,
                  :emp_row.address_data_2 indicator :emp_row.address_data_2_ind,
                  :emp_row.city indicator :emp_row.city_ind,
                  :emp_row.state indicator :emp_row.state_ind,
                  :emp_row.postal_code indicator :emp_row.postal_code_ind,
                  :emp_row.sex indicator :emp_row.sex_ind,
                  :emp_row.birthday indicator :emp_row.birthday_ind,
                  :emp_row.status_code indicator :emp_row.status_code_ind);
```

(continued on next page)

Example 6–2 (Cont.) Loading the Sorted Data

```
    { Get the next data buffer. }
    get(data);

end;{while}

close(data);

exec sql commit;
exec sql disconnect default;

end.
```

You can perform all the previous steps in one large program. This program would extract the dbkey values, sort the data using the sort utility for your operating system, and then insert the data.

6.3 Modifying the Database to Load Data

This section describes aspects of the database definition that need to be adjusted before performing a load operation:

- Database-wide parameters
- Storage area parameters
- Table definition
- Index definition
- Storage map definition

6.3.1 Adjusting Database-Wide Parameters

Before the load operation, you should make the following database-wide changes:

- Disable or defer snapshot files or use an exclusive share mode.
- Increase the number of buffers.

You disable or defer snapshot files using an SQL ALTER DATABASE statement. For parallel load operations, you should disable snapshot files.

The following example shows how to disable the snapshot file:

```
SQL> ALTER DATABASE FILENAME mf_personnel
cont>      SNAPSHOT IS DISABLED;
```

The preferred method is to run the load operation in an exclusive transaction share mode, which never updates the snapshot file. The following example shows how to set the transaction share mode so that the EMPLOYEES table is reserved for exclusive use when you use an SQL program to load your data.

```
SET TRANSACTION READ WRITE
    RESERVING EMPLOYEES FOR EXCLUSIVE WRITE;
```

For RMU Load operations, use the Transaction qualifier to specify exclusive mode.

To increase the number of buffers, use one of the following methods:

- Specify a larger value for the number of buffers using the RDM$BIND_BUFFERS logical name or the RDB_BIND_BUFFERS configuration parameter to temporarily override the database default value. This is the preferred method.
- Specify a larger value in the NUMBER OF BUFFERS IS option of an SQL ALTER DATABASE statement.

  Remember to set this value back to the desired number of buffers following the load operation.
- Specify the Buffers qualifier to the RMU Load command.

See the *Oracle Rdb7 Guide to Database Performance and Tuning* for more information on using the RDM$BIND_BUFFERS logical name or RDB_BIND_BUFFERS configuration parameter.

6.3.2 Adjusting Storage Area Parameters

Correctly estimating page and file allocation size for the storage area is one of the most important parts of the loading process. To estimate the page size correctly you must know what entities (tables and indexes) are on the page, calculate their row and record sizes, including overhead, and then estimate how many rows and index structures reside on the page (assuming a uniform load of the data).

If you cluster tables and hashed indexes together, including parent and child tables to achieve one I/O operation for an exact match retrieval, you must consider the total amount of space all four entities (2 tables and 2 hashed indexes) require on that page. To estimate the file allocation size correctly, you must know how many rows are to be loaded, the page size, and approximately how many rows might fit on the page. These calculations are described in Section 4.8.

If you underestimate the page size you may create a situation that produces many hash bucket overflows and hash bucket displacement from the row. Both conditions can lead to poor load and retrieval performance.

If you underestimate the file allocation, your file may extend, and cause hash bucket overflows and further hash bucket displacement from the row, resulting in even poorer load and retrieval performance. Any miscalculations can dramatically increase the load time. To determine how your load operation is proceeding during a load operation, refer to Section 6.4. If you overestimate these parameters by too large a margin, you will have too much empty space in your storage area and again performance may be poor.

6.3.3 Modifying Tables

If you have tables with constraints, you should be aware that they can affect the load operation. If you define constraints in one table and reference them in another, you must be careful about the order in which you load the tables. For example, the sample personnel database defines several constraints in the EMPLOYEES table and defines several references to the EMPLOYEES table in the JOB_HISTORY table. In this instance, you must be sure to load the EMPLOYEES table before the JOB_HISTORY table.

To ensure the best load performance when you are using SQL, you should delete any constraints and triggers and create them again after the load operation completes. If you use the RMU Load command, you can use the Constraints=Deferred or the Noconstraints qualifier to postpone constraint evaluation and the Notriggers qualifier to temporarily disable triggers.

6.3.4 Modifying Indexes

Only the following indexes should be defined before the load operation:

- A placement index that is used to place the rows in the storage area (and that is referred to by a storage map statement.)
- A sorted index that is used to cluster presorted table rows in a storage area resulting in a significant improvement in range retrieval performance.

  Define this index for any table in which range retrieval, not exact match queries, is the predominant access method.

Any other indexes should be defined following the load operation.

To cluster rows to achieve one I/O operation for an exact match retrieval, store the hashed placement index with the rows. If either the hashed or sorted placement index is to be partitioned across multiple storage areas, the storage areas named and the WITH LIMIT OF values specified must be identical for each placement index used to store rows in the same group of storage areas.

See Section 4.9 for information about clustering strategies.

6.3.5 Modifying Storage Maps

The storage map definition determines where the rows are to be stored. The PLACEMENT VIA INDEX clause specifies the index used to place the rows. For example, if the hashed index and rows are to be clustered together in the same storage area to achieve one I/O operation for an exact match retrieval, the PLACEMENT VIA INDEX clause specifies the name of the hashed index and the STORE USING clause in the storage map definition must be identical to the index definition STORE USING clause.

If a sorted index (clustering index) is to be used to cluster sorted table rows in a storage area to improve range retrieval performance, specify the name of the sorted index in the PLACEMENT VIA INDEX clause of the storage map statement. The STORE USING clause must specify the name of the storage area.

In general, it is advantageous to store the sorted index in one storage area and the data in a separate storage area. This reduces the maintenance for the storage area containing the data, and is especially important if you expect the table to grow and to be updated frequently.

If you want the table to be partitioned across multiple storage areas and if the sorted index definition specifies more than one column, the column names specified in the STORE USING clause of the storage map definition must be identical and in the same order as they appear in the sorted index definition.

If you want parent and child tables stored in the same storage area (clustering), you must specify identical storage map definitions, except for their names, for each table.

For information about clustering strategies and how to implement them, see Section 4.9.

If you want the rows to be stored in uncompressed form, you must specify DISABLE COMPRESSION in the storage map definition. The default is compressed format.

6.4 Troubleshooting Data Load Operations

After you start a large load operation that may be loading millions of rows into a table for the first time, check the progress of the operation. If you have made any miscalculations, problems will arise while you are loading data. You can use the following guidelines to check progress:

- Check the following statistics using the Oracle Rdb Performance Monitor:
  - Hash inserts per second (Hash Index Statistics display)
  - Physical input/output (PIO) statistics (PIO Statistics displays)
  - Record statistics (Record Statistics display)

OpenVMS VAX OpenVMS Alpha

- Check OpenVMS quotas using the Oracle Rdb Performance Monitor's VMS Parameters screen to help in the analysis. ♦
- Use the Statistics qualifier of the RMU Load command to check the following statistics during the load operation:
  - Page faults
  - CPU usage

  You can also use the Oracle Rdb Performance Monitor's Process Account screen.
- For parallel load, use the Statistics qualifier to the RMU Load command to check the state of each executor, in addition to CPU usage and page faults.

  The following example shows output from the Statistics qualifier during a parallel load operation:

```
ELAPSED:   0 00:00:33.22  CPU: 0:00:01.64  BUFIO: 55  DIRIO: 222  FAULTS: 2935

 1908 data records read from input file.
  657 records loaded before last commit.
  807 records loaded in current transaction.
  0 early commits by executors.
  3 executors:  0 Initializing; 0 Idle; 0 Terminated
                0 Sorting; 0 Storing; 2 Committing; 1 Executing
```

After the load operation completes, you should check the following items to determine the status of the operation:

- For parallel load, look at the "Idle time" listed in the statistics at the end of the job to detect data skew. Look at the number of "Early commits", which may indicate locking contention.

  If some executors have a large amount of idle time, you likely have data that is skewed. That is, the data in the input file may be sorted or records to be inserted into specific storage areas are grouped together in the input file.
- Check for file extents (RMU Dump command).
- Check for disk fragmentation (On OpenVMS, the DCL DUMP/HEADER command).

- Check for hash bucket overflows (RMU Analyze command with the Indexes qualifier).
- Check for data row displacement from its hash bucket (RMU Analyze command with the Placement qualifier).
- Check storage area usage and fullness percentage (RMU Analyze command with the Areas qualifier).

During a load operation, note trends in the statistics for the items in the previous lists. For example, decreasing hash insert values and higher CPU usage values may signal problems with the load operation due to miscalculations in the page size and file allocation size. Low estimates of page and file allocation sizes result in the load operation using extra I/O to look for available space on other data pages. When page and file allocations are too small, hash buckets can overflow, rows can be displaced many pages from their hash bucket, and extents are created to store rows that could not fit in the original allocation.

Assume, in a non-parallel load, you forget to sort the dbkeys before or during the data load operation. What might you expect to result from this oversight, and how could you quickly determine that this was the problem? Run the Oracle Rdb Performance Monitor for the database you are loading and select the PIO Statistics—Writes display to show the pool overflow rates. If these rates are high, many different data pages are being brought into the buffer to store data. Ideally, if the dbkeys are sorted, the pool overflow rates would be quite low because the load operation would populate the data pages of the storage area in a sequential fashion, with efficient use of data buffers.

Also, check the Record Statistics display. High record statistic rates for rows stored indicate that only an individual row is being written to each data page in the buffer, and then written to disk. This process is writing many pages to disk that contain just a single data row and its hash bucket or hash bucket entry. Lower rates for direct I/O operations indicate that newly inserted rows will be stored in the current page in-memory. This results in a load operation that reads each storage area page once, fills it with data rows and a hash bucket, and then writes the full page to disk.

In addition, you should check your operating system quotas before beginning the load operation and use the Oracle Rdb Performance Monitor to check your system performance during the load operation. Check page faults and CPU usage values to determine if these values are in their normal range.

If you suspect problems, after the load operation completes, use the RMU Analyze command to analyze row placement. Specifically, use the RMU Analyze command with the Indexes qualifier to check for hash bucket overflows and the depth or level of the overflows. Check the minimum I/O operation values for each index to ensure that, if hash buckets overflowed or data rows were displaced from their hash buckets, the row can be retrieved in one I/O operation. To check the minimum I/O operation values, use the RMU Analyze command with the Placement qualifier. See the *Oracle Rdb7 Guide to Database Performance and Tuning* for more information on using this command and its qualifiers.

Display the data on the page using the RMU Dump command to further diagnose a problem. See the *Oracle Rdb7 Guide to Database Maintenance* for more information on using this command. You use the RMU Dump command to display the database root file to check if any of the storage area files extended as a result of the load operation.

6.5 Loading Data from a Flat File Using SQL Programs

You can use an SQL precompiled program or an SQL module language program to load data from a flat file into a database. An SQL load program contains an INSERT statement that assigns columns from the source file to columns in an Oracle Rdb table. In the simplest case, the columns in the input file exactly match the columns in the database table, both in size and data type. In this case, you load the columns from each row in a single data file into a single table. You can also copy the data declarations for the program from the repository.

Your load program can also change the structure of the data. The program can split a file across more than one table, combine several data files in a single table, convert data from one data type to another, and transform the data in other ways. If your program loads a large amount of data, you should include the exclusive share mode on the database resources you intend to update. If the program must run concurrently with other users, it may run out of row locks. To solve this problem, you can use one of the following methods:

- Include a counter to count the rows as they are stored, and commit the transaction after the counter reaches a certain number. This number should be between 100 and 1000.
- For a more precise method, use the following formula to determine the number of rows to commit:

$$\begin{pmatrix} No.\ of\ rows\ between \\ commit\ operations \end{pmatrix} = \begin{pmatrix} No.\ of\ rows \\ per\ page \end{pmatrix} * \frac{Buffer\ size}{Page\ size} * No.\ of\ buffers$$

In an ideal load operation, this number causes all buffers to fill completely and then commits them to disk using I/O operations that result in maximum I/O speed.

The following subroutine from an SQL precompiled C program shows how such a counter might work:

```
trans_size = 100
trans_counter = trans_counter + 1

if (trans_counter >= trans_size)
   {
    EXEC SQL COMMIT;
    EXEC SQL SET TRANSACTION READ WRITE;
    trans_counter = 0
   }
```

Section 6.5.1 describes a BASIC program that calls an SQL module to load data into an Oracle Rdb database; Section 6.5.2 describes an SQL precompiled COBOL program; and Section 6.5.3 describes an SQL precompiled C program. These examples and other examples are available online in the sample directory.

6.5.1 Using the SQL Module Language and BASIC to Load Data

This section discusses a sample BASIC program and an SQL module called by the BASIC program to load data from a data file into the JOBS table in the personnel database.

This program illustrates a special problem. All the columns in the input file are ASCII text. In the target database, however, the salary columns are defined as INTEGER(2) data type. BASIC does not automatically convert ASCII text data to integer data. Furthermore, you cannot use the %INCLUDE %FROM %CDD directive because the data type that BASIC copies from the repository for the salary columns does not match the data type for the corresponding columns in the input file. Your program must convert the data, using the INTEGER function.

Example 6–3 shows sql_load_jobs.bas, a host language BASIC program, which calls SQL module procedures to load data into a database.

Example 6–3 BASIC Program That Calls an SQL Module to Load Data

```
! This program calls procedures from the SQL module sql_load_jobs_bas.sqlmod
! to load data into the JOBS table of the personnel database.

OPTION TYPE = EXPLICIT

ON ERROR GOTO ERR_ROUTINE

! Declare a variable to hold the value of SQLCODE.
DECLARE LONG SQL_RETURN_STATUS

! Declare variables to hold the integer values of the input salary amounts.
DECLARE INTEGER MAX_SAL, MIN_SAL

! Use the RECORD statement to specify a record structure.

RECORD JOBS
       STRING JOB_CODE = 4
       STRING FILL1 = 3
       STRING WAGE_CLASS = 1
       STRING FILL2 = 3
       STRING JOB_TITLE = 20
       STRING FILL3 = 3
       STRING MINIMUM_SALARY = 6
       STRING FILL4 = 3
       STRING MAXIMUM_SALARY = 6
       STRING FILL5 = 1
END RECORD JOBS

! Use MAP to associate the record structure with the input file.
MAP (LINEIN) JOBS JOB_REC

! Declare the calls to the SQL module language procedures.
EXTERNAL SUB SET_TRANSACTION (LONG)
EXTERNAL SUB INSERT_JOBS (LONG,STRING,STRING,STRING,INTEGER,INTEGER)
EXTERNAL SUB COMMIT_TRANS (LONG)
EXTERNAL SUB ROLLBACK_TRANS (LONG)

! Open the file, using MAP to associate the input file with
! the record structure.

OPEN "sql_jobs.dat" &
FOR INPUT AS FILE #1, ORGANIZATION SEQUENTIAL &
VARIABLE, RECORDTYPE ANY, MAP LINEIN

! Call the SQL module to start the transaction.
CALL SET_TRANSACTION(sql_return_status)

get_loop:
   WHILE -1%
   GET #1
```

(continued on next page)

Example 6–3 (Cont.) BASIC Program That Calls an SQL Module to Load Data

```
     ! Use the INTEGER function to convert the TEX data type to LONGWORD.
     MIN_SAL = INTEGER(JOB_REC::MINIMUM_SALARY)
     MAX_SAL = INTEGER(JOB_REC::MAXIMUM_SALARY)

     ! Call the SQL module to insert a row in the jobs table.   Notice
     ! that the program stores MIN_SAL and MAX_SAL, the converted integer
     ! values, instead of the TEXT fields from the input file.

     CALL INSERT_JOBS (sql_return_status,JOB_REC::JOB_CODE, &
                      JOB_REC::WAGE_CLASS, JOB_REC::JOB_TITLE, &
                      MIN_SAL,MAX_SAL)

   NEXT

err_routine:

    SELECT ERR
    CASE 11
         PRINT "Encountered end-of-file"
         ! Commit the transaction.
         CALL COMMIT_TRANS(sql_return_status)
         RESUME JOB1
    CASE ELSE
         PRINT "Unexpected error number "; ERR
         PRINT "Error is "; ERT$(ERR)
         ! Roll back the transaction if an unexpected error occurs.
         CALL ROLLBACK_TRANS(sql_return_status)
         RESUME JOB1

    END SELECT

JOB1:
        END
```

Example 6–4 shows the source code for the SQL module sql_load_jobs_bas.sqlmod.

Example 6–4 Using an SQL Module to Load Data

```
-- This SQL module provides the SQL procedures needed by the
-- sql_load_jobs.bas program.  The module illustrates how to use
-- SQL module language to load a database.
```

(continued on next page)

Example 6–4 (Cont.) Using an SQL Module to Load Data

```
---------------------------------------------------------------------------
-- Header Information Section
---------------------------------------------------------------------------
MODULE          SQL_LOAD_JOBS_BAS  -- Module name
LANGUAGE        BASIC              -- Language of calling program
AUTHORIZATION   SQL_SAMPLE         -- Provides default db handle
PARAMETER COLONS

---------------------------------------------------------------------------
-- DECLARE Statements Section
---------------------------------------------------------------------------
DECLARE ALIAS FILENAME personnel    -- Declaration of the database.

---------------------------------------------------------------------------
-- Procedure Section
---------------------------------------------------------------------------

-- This procedure uses the executable statement SET TRANSACTION to
-- start a transaction.

PROCEDURE SET_TRANSACTION
    SQLCODE;

    SET TRANSACTION READ WRITE RESERVING
        JOBS FOR EXCLUSIVE WRITE;

-- This procedure inserts a row in the JOBS table.  The list of names in
-- the VALUES clause corresponds to the parameter list for the procedure.

PROCEDURE INSERT_JOBS
    SQLCODE
    P_JOB_CODE        CHAR(4),
    P_WAGE_CLASS      CHAR(1),
    P_JOB_TITLE       CHAR(20),
    P_MINIMUM_SALARY  INTEGER,
    P_MAXIMUM_SALARY  INTEGER;

     INSERT INTO JOBS
     VALUES (P_JOB_CODE,P_WAGE_CLASS,P_JOB_TITLE,
             P_MINIMUM_SALARY,P_MAXIMUM_SALARY );

-- This procedure commits the transaction.

PROCEDURE COMMIT_TRANS
    SQLCODE;

    COMMIT;
```

(continued on next page)

Example 6–4 (Cont.) Using an SQL Module to Load Data

```
-- This procedure rolls back the transaction.

PROCEDURE ROLLBACK_TRANS
    SQLCODE;

    ROLLBACK;
```

Online versions of the source files sql_load_jobs.bas and sql_load_jobs_bas.sqlmod are available in the sample directory.

6.5.2 Using the SQL Module Language, COBOL, and Repository Definitions to Load Data

This section discusses a COBOL program and the SQL module that it calls to load data into an Oracle Rdb database.

In this example, the structure of the input file matches exactly the structure of the Oracle Rdb database. If you used the repository when you created the database, you can use the definition of the table from the repository with this load operation. Although the domain (field) and table (record) definition are stored in the repository, they do not have directory names. That is, you cannot see the names when you enter the directory name in the Common Dictionary Operator (CDO), and you cannot use CDO commands to refer to them. To do this, you must use the CDO ENTER command shown here to give the definition a directory name.

```
CDO> ENTER RECORD table-name FROM DATABASE your-cdd-path.database-name
```

You must use the CDO ENTER command for every table and field you wish to make visible. The example in this section loads data into the DEPARTMENTS table of the personnel database. To make the DEPARTMENTS table visible, use the following command:

```
CDO> ENTER RECORD DEPARTMENTS FROM DATABASE CDD$DEFAULT.PERSONNEL
```

The load operation involves the following steps:

1. Declare the input file.
2. Use the COBOL language COPY FROM DICTIONARY statement to create a record description for DATA DIVISION.

   In the COPY FROM DICTIONARY statement, you must include the path name for the table. If you have used the CDO ENTER command, you specify the path name, as shown in the following example:

```
COPY "CDD$DEFAULT.DEPARTMENTS"
     FROM DICTIONARY.
```

If you have not used the CDO ENTER command, you specify the path name, as shown in the following example:

```
COPY "CDD$DEFAULT.PERSONNEL.RDB$RELATIONS.DEPARTMENTS"
        FROM DICTIONARY.
```

3. Open the input file.
4. Start a transaction.
5. Read the input file and use an INSERT statement to store each row.
6. Commit the transaction and close the input file.

Example 6–5 shows the host language COBOL program.

Example 6–5 COBOL Program That Calls an SQL Module to Load Data

```
IDENTIFICATION DIVISION.
PROGRAM-ID.   LOAD_DEPTS.
*
*       Load the DEPARTMENTS table of the personnel database.
*
ENVIRONMENT DIVISION.
INPUT-OUTPUT SECTION.
FILE-CONTROL.

*
*  Declare the input file.
*
        SELECT DEPT-FILE  ASSIGN TO "DEPARTMENTS.DAT"
        ORGANIZATION IS SEQUENTIAL
        ACCESS MODE IS SEQUENTIAL.

DATA DIVISION.
FILE SECTION.
FD DEPT-FILE.
*
*  The COPY FROM DICTIONARY statement copies the Oracle Rdb table definition
*  and uses it as a COBOL record description.  Therefore, the columns in
*  the record description must match the columns in the table exactly.
*  You cannot use the data file sql_depts.dat located in the sample
*  directory, because it contains extra "filler" spaces between columns.

        COPY "CDD$DEFAULT.DEPARTMENTS"
        FROM DICTIONARY.
```

(continued on next page)

Example 6–5 (Cont.) COBOL Program That Calls an SQL Module to Load Data

```
WORKING-STORAGE SECTION.

01 SQLCODE                      PIC S9(9) USAGE IS COMP.
01 FLAGS                        PIC X.
        88 END-OF-FILE          VALUE "Y".
        88 NOT-END-OF-FILE      VALUE "N".

PROCEDURE DIVISION.

*       Program LOAD_DEPTS reads Department data and stores it
*       in the DEPARTMENTS table of the personnel database.

START-UP.

        SET NOT-END-OF-FILE TO TRUE.
        OPEN INPUT DEPT-FILE.

* Call the SQL module procedure SET_TRANSACTION to start the transaction.
        CALL "SET_TRANSACTION" USING SQLCODE.

*       Start Program: DEPTS

MAIN-LINE.

        PERFORM LOAD THRU LOAD-EXIT UNTIL END-OF-FILE.

999-EOJ.

*       End of Program: DEPTS

        DISPLAY "Program: DEPARTMENTS Loaded. Normal End-of-Job".
*       Now commit the transaction.

* Call the SQL module procedure COMMIT_TRANS.
        CALL "COMMIT_TRANS" USING SQLCODE.

        CLOSE DEPT-FILE.

        STOP RUN.
*****************************************************************
*                   <<< Subroutines >>>                         *
*****************************************************************

LOAD.
        PERFORM READ-DEPTS THRU READ-DEPTS-EXIT.
        IF NOT-END-OF-FILE
        THEN
            PERFORM STORE-DEPTS THRU STORE-DEPTS-EXIT
        END-IF.
```

(continued on next page)

Example 6–5 (Cont.) COBOL Program That Calls an SQL Module to Load Data

```
LOAD-EXIT.
        EXIT.

READ-DEPTS.

        READ DEPT-FILE AT END SET END-OF-FILE TO TRUE
                GO TO LOAD-EXIT.

READ-DEPTS-EXIT.
        EXIT.

STORE-DEPTS.

* Call the SQL module procedure INSERT_DEPTS to insert the data into the
* database.

        CALL "INSERT_DEPTS" USING SQLCODE, DEPARTMENTS.
STORE-DEPTS-EXIT.
        EXIT.
```

Example 6–6 shows the module language program that loads the data into the database.

Example 6–6 Loading Data Using an SQL Module

```
-- This SQL module provides the SQL procedures needed by the
-- sql_load_depts.cob program to load data into the DEPARTMENTS table.

-----------------------------------------------------------------------------
-- Header Information Section
-----------------------------------------------------------------------------
MODULE          SQL_LOAD_DEPTS_COB -- Module name
LANGUAGE        COBOL              -- Language of calling program
AUTHORIZATION   SQL_SAMPLE
PARAMETER COLONS

-----------------------------------------------------------------------------
-- DECLARE Statements Section
-----------------------------------------------------------------------------
DECLARE ALIAS PATHNAME personnel    -- Declaration of the database.
```

(continued on next page)

Example 6–6 (Cont.) Loading Data Using an SQL Module

```
-----------------------------------------------------------------------------
-- Procedure Section
-----------------------------------------------------------------------------

-- This procedure uses the executable statement SET TRANSACTION to
-- start a transaction.

PROCEDURE SET_TRANSACTION
    SQLCODE;

    SET TRANSACTION READ WRITE RESERVING
        DEPARTMENTS FOR PROTECTED WRITE;

-- This procedure inserts a row in the DEPARTMENTS table.

PROCEDURE INSERT_DEPTS
    SQLCODE,
     DEPTS_REC RECORD
       FROM 'CDD$DEFAULT.DEPARTMENTS'
       END RECORD;

     INSERT INTO DEPARTMENTS
     VALUES (DEPTS_REC);

-- This procedure commits the transaction.

PROCEDURE COMMIT_TRANS
    SQLCODE;

    COMMIT;
```

6.5.3 Using SQL Precompiled C Programs to Load Data

This section discusses sql_load_jobhist.sc, an SQL precompiled C program that reads data from a single input file and loads the data into a table, JOB_ HISTORY. The sample directory contains an online version of this program.

This program, shown in Example 6–7, performs the following steps:

1. Declares variables to hold the fields from the input file
2. Attaches to the database and starts a read/write transaction
3. Converts text dates to DATE data types
4. Stores data in the JOB_HISTORY table

The error handler checks for several possible errors. More detailed commentary appears in the program in Example 6–7.

Example 6–7 Loading Data Using an SQL Precompiled C Program

```
/*  ABSTRACT:
*
*      This program demonstrates the use of the SQL precompiler for the C
*      language to load an Oracle Rdb database from a stream (flat) file.
*
*      This program attaches to an existing Oracle Rdb database, opens a data
*      file containing job history records, and reads the records,
*      formatting and inserting them into the database until the end of
*      the data file is reached.  Then, the program commits the transaction.
*/

#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#ifdef VMS
#include "sql$sample:sql_load_rtl.sc"
#endif

#ifdef __osf__
#include <sql_load_rtl.sc>
#endif

main( )
{

/* Fields to receive strings read from the sql_jobhist.dat file record. */
    char    jh_id[6];
    char    j_code[5];
    char    ascii_start_date[24];
    char    ascii_end_date[24];
    char    d_code[5];
    char    supr_id[6];

/* File definitions for reading the sql_jobhist.dat file */
    FILE    *jobhist_file;

/* Declarations for error handling */
    int     return_status;
    int     sql_signal();

/* Variables for main program use */
    int             i;          /* loop counter */
```

(continued on next page)

Example 6–7 (Cont.) Loading Data Using an SQL Precompiled C Program

```
/* Define the SQLCA. */
    EXEC SQL INCLUDE SQLCA;

/* Declare the database. */
    EXEC SQL DECLARE ALIAS FILENAME personnel;

/* Set up error handling for failures on execution of SQL statements. */
    EXEC SQL WHENEVER SQLERROR GOTO HANDLE_ERROR;

/* Operator message to the terminal */
    printf("\nProgram: Loading JOB_HISTORY");

/* Open the sequential file containing the job history data records. */
#ifdef VMS
    jobhist_file = fopen("sql$sample:sql_jobhist.dat","r");
#endif

#ifdef __osf__
    jobhist_file = fopen("sql_jobhist.dat","r");
#endif

/* This procedure uses the executable form for starting a transaction.
*/
    EXEC SQL SET TRANSACTION READ WRITE RESERVING
                JOB_HISTORY FOR EXCLUSIVE WRITE;

/* Main loop until data file is empty */
    while (get_line(jobhist_file) != NULL)
        {
        get_field(jobhist_file,jh_id            ,5);
        get_field(jobhist_file,NULL             ,3);
        get_field(jobhist_file,j_code           ,4);
        get_field(jobhist_file,NULL             ,3);
        get_field(jobhist_file,ascii_start_date ,23);
        get_field(jobhist_file,NULL             ,3);
        get_field(jobhist_file,ascii_end_date   ,23);
        get_field(jobhist_file,NULL             ,3);
        get_field(jobhist_file,d_code           ,4);
        get_field(jobhist_file,NULL             ,2);
        get_field(jobhist_file,supr_id          ,5);

/* This compound statement uses the CAST and SUBSTRING functions to
convert the dates to the DATE VMS data type format and then inserts
the row into the table.

In the INSERT statement, the list of names in the VALUES clause corresponds
to the host variables containing the values.  The list of names that follows
the INSERT clause names the columns in the table that are to be inserted.
*/
```

(continued on next page)

Example 6–7 (Cont.) Loading Data Using an SQL Precompiled C Program

```
EXEC SQL
BEGIN
   DECLARE :start_date DATE VMS;
   DECLARE :end_date DATE VMS;
   SET :start_date  = CAST(SUBSTRING(:ascii_start_date FROM 8 FOR 4) ||
         -- Convert the month to a number.
          (CASE SUBSTRING(:ascii_start_date FROM 4 FOR 3)
                  WHEN 'JAN' THEN '01'
                  WHEN 'FEB' THEN '02'
                  WHEN 'MAR' THEN '03'
                  WHEN 'APR' THEN '04'
                  WHEN 'MAY' THEN '05'
                  WHEN 'JUN' THEN '06'
                  WHEN 'JUL' THEN '07'
                  WHEN 'AUG' THEN '08'
                  WHEN 'SEP' THEN '09'
                  WHEN 'OCT' THEN '10'
                  WHEN 'NOV' THEN '11'
                  WHEN 'DEC' THEN '12'
               END) ||
         -- Parse the day, hour, minutes, seconds.
           SUBSTRING(:ascii_start_date FROM 1 FOR 2)  ||
           SUBSTRING(:ascii_start_date FROM 13 FOR 2) ||
           SUBSTRING(:ascii_start_date FROM 16 FOR 2) ||
           SUBSTRING(:ascii_start_date FROM 19 FOR 2) ||
           SUBSTRING(:ascii_start_date FROM 22 for 2)
         AS DATE VMS);

 -- If the end date equals 17-NOV-1858 00:00:00.00, set :end_date
 -- to NULL. If it does not, convert it to DATE VMS format.

    IF :ascii_end_date <> '17-NOV-1858 00:00:00.00'
    THEN
       SET :end_date  = CAST(SUBSTRING(:ascii_end_date FROM 8 FOR 4) ||
         -- Convert the month to a number.
               (CASE SUBSTRING(:ascii_end_date FROM 4 FOR 3)
                  WHEN 'JAN' THEN '01'
                  WHEN 'FEB' THEN '02'
                  WHEN 'MAR' THEN '03'
                  WHEN 'APR' THEN '04'
                  WHEN 'MAY' THEN '05'
                  WHEN 'JUN' THEN '06'
                  WHEN 'JUL' THEN '07'
                  WHEN 'AUG' THEN '08'
                  WHEN 'SEP' THEN '09'
```

(continued on next page)

Example 6–7 (Cont.) Loading Data Using an SQL Precompiled C Program

```
                   WHEN 'OCT' THEN '10'
                   WHEN 'NOV' THEN '11'
                   WHEN 'DEC' THEN '12'
                END) ||
         -- Parse the day, hour, minutes, seconds.
                SUBSTRING(:ascii_end_date FROM 1 FOR 2)  ||
                SUBSTRING(:ascii_end_date FROM 13 FOR 2) ||
                SUBSTRING(:ascii_end_date FROM 16 FOR 2) ||
                SUBSTRING(:ascii_end_date FROM 19 FOR 2) ||
                SUBSTRING(:ascii_end_date FROM 22 for 2)
            AS DATE VMS);

    ELSE
         SET  :end_date = NULL;
    END IF;

   -- Insert the row.

      INSERT INTO JOB_HISTORY
                    (EMPLOYEE_ID, JOB_CODE, JOB_START, JOB_END,
                     DEPARTMENT_CODE, SUPERVISOR_ID)
                    VALUES
                    (:jh_id, :j_code, :start_date, :end_date,
                     :d_code, :supr_id);
 END;
       }

/* Commit the transaction. */
    EXEC SQL COMMIT;

/* Close the sql_jobhist.dat data file. */
    fclose(jobhist_file);

/* Operator prompt message */
    printf("\nProgram: JOB_HISTORY Loaded. Normal End-of-Job");
    exit(1);

/* Error handler for SQL errors */
HANDLE_ERROR:
    sql_signal();
    exit(0);
}
```

6.6 Loading and Unloading Data Using the RMU Load and RMU Unload Commands

You can use the RMU Load command to load data into database tables from sequential flat files or from specially formatted data unload (.unl) files created

with the RMU Unload command. The RMU Load command is useful for the following tasks:

- Loading the initial data
- Loading large quantities of new data
- Restructuring databases
- Loading data from one database into another
- Archiving data

The source of the data can be:

- Tables from other Oracle Rdb databases, including archived databases
- Databases other than Oracle Rdb that can unload their data into sequential flat files
- Tables that have been unloaded to perform a restructuring operation with the RMU Unload command

When you use RMU Load, you can use a single-process or multiprocess load operation. For information about the multiprocess (parallel) load operation, see Section 6.7.

When you load a database using sequential flat files, you need the following:

- A record definition (.rrd) file that contains the metadata or the full path name in the repository where the metadata can be found. The metadata describes the format of the data in the flat file.

  You specify the source of the metadata using the Record_Definition=option qualifier, where the option is one of two arguments:

  - FILE=*filename* for the .rrd file. The .rrd file uses the same record and field definition format as the repository's CDO utility.
  - PATH=*pathname* for a CDO record definition that is to be extracted from the repository
- A sequential flat file containing data to be loaded.

When you load a database using specially formatted files created by the RMU Unload command, you need only the data file (default file type .unl).

6.6.1 Improving Performance While Using the RMU Load Command

The performance of load operations can be affected positively or adversely by a number of factors. See Section 6.1 for general information about improving load performance. Consider the following factors when loading data with the RMU Load command:

- You can use parallel load (a multiprocess load operation) or a single-process load operation. See Section 6.7 for specific information about parallel load operation.
- You should disable triggers by specifying the Notrigger_Relations qualifier of the RMU Load command. If you specify a list of tables, those tables are reserved for write access. If you omit the Trigger_Relations qualifier, the tables are automatically locked as required. In order for the Notrigger_Relations qualifier to automatically disable the defined triggers, the user doing the load operation must have delete access rights to the triggers. Note that trigger definitions are not deleted, but are temporarily disabled for the load operation.
- If you specify the Place qualifier to the RMU Load command, use a hashed index and the PLACEMENT VIA INDEX clause in a storage map statement to significantly enhance the performance of the load operation. The Place qualifier sorts the records as that they are loaded so they are stored as efficiently as possible.
- By using the Commit_Every qualifier, you can specify how many records to load between COMMIT statements. If you are loading large numbers of records using a placement index, you should use one of the following methods:
  - Avoid the use of the Commit_Every qualifier and allocate multiple sortwork files so that all records can be sorted before they are written sequentially, page by page. This is the most efficient method but no records are committed until the load operation completes.
  - Use the Commit_Every qualifier but use the largest value that your system can support. That is, determine how much space your system can spare for sorting and increase the value of the Commit_Every qualifier to make use of that space. This approach means that the records are sorted in smaller sort sets and written sequentially, page by page, for each commit issued. This method is less efficient because the total load operation for multiple commits issued does not perform sequential loading; that is, the same page can be written to more than once with each commit issued.

  The Commit_Every qualifier has the following benefits:

- The .ruj file does not grow as large.
- For non-parallel load operations, you can restart in the middle of a load operation in case of failure.
- It reduces memory usage by flushing contextual information.
- It reduces the number of locks used when the load is not using exclusive mode.

- Constraints adversely affect the performance of the load operation. Use the Constraints=Deferred or Noconstraints qualifiers to specify when Oracle Rdb evaluates constraints during the load operation.

  Use these qualifiers when load performance is your highest priority and you are fairly certain that the data you are loading does not violate any constraints.

  The Constraints=Deferred qualifier specifies that Oracle Rdb evaluates constraints only after all the data in the input file has been loaded. This qualifier is particularly useful when you are loading data into a new table.

  The Noconstraints qualifier specifies that Oracle Rdb does not evaluate constraints during the load operation. Use this qualifier only when you are sure that your data does not violate any constraints and the cost of Constraints=Deferred is too high.

  If you use the Noconstraints qualifier, Oracle Corporation recommends that you issue an RMU Verify command with the Constraints qualifier after you complete the load operation to be certain that the data does not violate any constraints.

  See the *Oracle RMU Reference Manual* for information about using these qualifiers.

- Using the Row_Count option, you can specify the number of rows that are sent between the RMU Load and server processes in a single I/O request, decreasing the communications overhead.

  Digital UNIX

  If you are using Digital UNIX, you should specify this option to improve performance. The optimum number depends on your system resources, but a Row_Count of 500 is a reasonable starting point. ♦

  OpenVMS VAX OpenVMS Alpha

  On OpenVMS, specifying this option is not as crucial. A Row_Count of 50 (the default) may be sufficient. ♦

- On OpenVMS, increase the memory available to the load process by increasing the working set parameters of the load process. Increase the parameters of WSQUOTA, WSEXTENT, and WSMAX. WSEXTENT should be at least twice as large as WSQUOTA to allow the Oracle Rdb sort routines to perform most effectively. ♦
- Using the Exclusive transaction type reduces the number of locks acquired by the load process, reducing the load time. If a lock cannot be acquired, RMU will retry the load. Consider using the Commit_Every and Skip qualifiers to allow restart after a lock conflict occurs or use the exception file to catch the records in conflict and load them again in a second pass.

  If you are using a parallel load operation and constraints are defined on the table and you are not using the Constraints=Deferred or Noconstraints qualifier, you should use the Shared transaction type.
- Using the Batch_Update transaction type reduces the number of locks taken out and disables writes to the .ruj file. Not writing to the .ruj file means the database is unusable in the event of a failure.

  This transaction type yields performance that is similar to the Exclusive transaction type.

  Caution

  Extreme care must be taken when using the Transaction_Type=Batch_Update qualifier. The database should be fully backed up before using the Batch_Update transaction.

For the RMU Load command, you can specify other qualifiers to assist you with the load operation, including the following:

- By specifying the Statistics qualifier with the Log_Commits option, you can display an informational message for each group of stored and committed records based on the value specified with the Commit_Every qualifier.
- By specifying the Statistics qualifier with the Interval option, you can display informational messages at set time-based intervals during the load operation.
- By specifying the Skip=*n* qualifier with a non-parallel load, you can define the number of rows to skip following a failed load operation (*n* equals the number of rows already loaded and committed).

For example, if the RMU Load operation fails part way through the load operation, note the number of rows committed in the message received and you can determine the location of the failure by specifying Commit_Every=1 and Skip=*n* where *n* equals the number of rows already loaded and committed. Then, the load operation commits at every row until it reaches the point of failure.

See the *Oracle RMU Reference Manual* for details on the syntax, a description of the qualifiers used with both the RMU Load and RMU Unload commands, and additional examples of these commands. See Section 6.7 for guidelines specific to parallel load operations.

6.6.2 Understanding the Format of the Record Definition File

To understand the format of the record definition file, you can perform an unload operation and inspect this file. To do this, enter the command shown in Example 6–8 to unload the data from the COLLEGES table and create a colleges.rrd record definition file and a flat data file, colleges.unl.

Example 6–8 Unloading a Table Using the RMU Unload Command

```
$ RMU/UNLOAD /RECORD_DEF=FILE=colleges.rrd mf_personnel -
_$ COLLEGES colleges.unl
%RMU-I-DATRECUNL, 15 data records unloaded
```

The following example shows the contents of the colleges.rrd file. The RMU Unload command presents the definitions in the format used by the repository's CDO utility.

```
DEFINE FIELD COLLEGE_CODE DATATYPE IS TEXT SIZE IS 4.
DEFINE FIELD COLLEGE_NAME DATATYPE IS TEXT SIZE IS 25.
DEFINE FIELD CITY DATATYPE IS TEXT SIZE IS 20.
DEFINE FIELD STATE DATATYPE IS TEXT SIZE IS 2.
DEFINE FIELD POSTAL_CODE DATATYPE IS TEXT SIZE IS 5.
DEFINE RECORD COLLEGES.
   COLLEGE_CODE .
   COLLEGE_NAME .
   CITY .
   STATE .
   POSTAL_CODE .
END COLLEGES RECORD.
```

Note that the .rrd file contains the record definition of each field in the record and the definition of the COLLEGES table specifying the order of the fields. If you specify a field order different from that of the COLLEGES table using the Fields=*field-name-list* qualifier, the field order changes accordingly.

The following example shows the contents of the colleges.unl file. Note that the display may vary depending upon the text editor or display command you use.

```
AU  American University      Washington          DC20016
BATEBates College            Lewiston            ME04240
BOWDBowdoin College          Brunswick           ME04011
CALTCal. Institute of Tech.  Pasadena            CA91125
COLBColby College            Waterville          ME04901
DREWDrew University          Madison             NJ07940
FLU Florida University       Gainesville         FL32601
HVDUHarvard University       Cambridge           MA02138
MIT Mass. Institute of Tech. Cambridge           MA02139
PRDUPurdue University        West Lafayette      IN47907
QUINQuinnipiac College       Hamden              CT06518
STANStanford University      Berkley             CA94305
UME University of Maine      Orono               ME04473
USCAU. of Southern California San Diego          CA92037
YALEYale University          New Haven           CT06520
```

The contents of this file are the same as the contents of the COLLEGES table.

6.6.3 Loading Data into a Database Table from a Flat File

When you use the Record_Definition qualifier of the RMU Load and RMU Unload commands, you can load data from flat files, archive data, or extract data for an application that cannot directly access the Oracle Rdb database. This section explains how to load data from a flat file into a table.

Understanding the format of the record definition file and the format of the flat data file, you can perform a load operation to add several new rows to the COLLEGES table. If all the data is text data, it is easy to add new rows by creating a text file. For example, to add three rows to the COLLEGES table, create a new file named colleges3.unl that contains these rows, as shown in the following example.

```
CU   Cornell University       Ithaca              NY14853
UMCO University of Missouri   Columbia            MO65205
VPI  Virginia Polytechnic Inst Blacksburg         VA24061
```

If the data includes data types other than text, you can create the flat file using an application program.

Use the colleges.rrd file created in the unload operation in the previous section as the record definition file. Enter the command shown in Example 6–9 to load this new data into the COLLEGES table.

Example 6–9 Loading Additional Rows into a Table Using the RMU Load Command

```
$ RMU/LOAD /RECORD_DEFINITION=FILE=colleges.rrd mf_personnel -
_$ COLLEGES colleges3.unl
%RMU-I-DATRECSTO, 3 data records stored
```

To see that these three new rows have been added to the COLLEGES table, use the SQL SELECT statement.

When you use the RMU Load command with the Record_Definition qualifier, the following restrictions apply:

- The number and data types of the fields specified in the database must agree with the number and data types of the fields in the record definition file.
- Fields in the flat record must be data types supported by SQL. Decimal data types, for example, are not supported.

  As another example, Oracle Rdb converts a date written in text format to a DATE data type only if the text is in the following format:

  ```
  YYYYNNDDHHMMSSCC
  ```

  For example, if the date and time is October 15, 1993 at 8:11 AM, you enter the date in the following manner:

  ```
  1993101508110000
  ```

  See the Language and Syntax Elements chapter of the *Oracle Rdb7 SQL Reference Manual* for a list of supported data types.

In addition, the following restrictions apply *unless* you are using the Format=Delimited qualifier:

- The flat data file must be in a fixed record length format.
- Each record must be completely filled; blank fields must be filled with spaces.
- If the last field in the record is character data and the information is shorter than the length of the field, the remainder of the field must be filled with spaces.

If you use the Record_Definition qualifier of the RMU Unload command to unload data from an Oracle Rdb database table, you should be aware of the following:

- You cannot unload columns with the LIST OF BYTE VARYING data type.

- Unless you use the Delimited_Text qualifier, Oracle Rdb converts varying length character strings to fixed length character strings. Their length becomes the largest length possible. For example, if a column in a table is defined as VARCHAR(255), all the instances of the column are converted to a fixed length of 255 characters.
- Unless you use the Delimited_Text qualifier with the Null option, Oracle Rdb converts null values in a column to zero or blank spaces depending upon the data type, because null values are not recognized in flat files.

  See Section 6.6.4 for information about using the Delimited_Text qualifier to load null values.
- If an SQL default value or an RDO missing value is defined for the column, Oracle Rdb unloads the default or missing value, not a null value. Then, if you use RMU Load to load the database, Oracle Rdb loads the default value or missing value, not a null value, into the column.

6.6.4 Loading Null Values

When you use the Record_Definition qualifier, null values for a column are not loaded into an Oracle Rdb database, unless you use the Delimited_Text qualifier with the Null option.

When you specify the Delimited_Text qualifier, the .rrd file should define all fields as text fields, specifying the maximum length of the columns in the data file. You can generate the .rrd file by using the RMU Unload command with the Format=Text or Delimited_Text option to the Record_Definition qualifier.

You can specify that the null representation be a character string or an empty string. In the following jobhist.dat data file, any null values (in the second and fourth rows) are represented by an empty string. That is, there are no characters or spaces between the two comma separators.

```
"00164","SPGM","1994070500000000","1995092100000000","MCBM","00164"
"00164","DMGR","1995092100000000",,"MBMN","00228"
"00165","ASCK","1981030800000000","1990011300000000","PHRN","00201"
"00165","ASCK","1990011300000000",,"ELGS","00276"
```

In the RMU Load command, you specify that empty strings represent null values by using `NULL=""` or just the word `NULL`, as shown in Example 6–10.

Example 6–10 Loading Null Values from Empty Strings

```
$ RMU/LOAD /RECORD_DEFINITION = (FILE=job_history.rrd,  -
_$ FORMAT=DELIMITED_TEXT, NULL="")
_Root: mf_personnel
_Table: JOB_HISTORY
_Input file: jobhist.dat
%RMU-I-DATRECREAD,  4 data records read from input file.
%RMU-I-DATRECSTO,   4 data records stored.
```

You specify how the null value is represented in the flat file by specifying the same characters in the Null option. For example, in the following jobhist.dat data file, null values (in the second and fourth rows) are represented by an asterisk (*):

```
"00164","SPGM","1994070500000000","1995092100000000","MCBM","00164"
"00164","DMGR","1995092100000000",*,"MBMN","00228"
"00165","ASCK","1981030800000000","1990011300000000","PHRN","00201"
"00165","ASCK","1990011300000000",*,"ELGS","00276"
```

Example 6–11 loads the data into the JOB_HISTORY table, specifying an asterisk as the null representation.

Example 6–11 Loading Null Values

```
$ RMU /LOAD /RECORD_DEF = (FILE=job_history.rrd, -
_$ FORMAT=DELIMITED_TEXT, NULL="*")
_Root: mf_personnel
_Table: JOB_HISTORY
_Input file: jobhist.dat
%RMU-I-DATRECREAD,  4 data records read from input file.
%RMU-I-DATRECSTO,   4 data records stored.
```

Note that the representation of the null value in the data file *is not* surrounded by double quotation marks, but that the null value in the command line *is* surrounded by double quotation marks.

In addition to specifying the null representation, you can specify the following delimiter options:

- Field separators, using the Separator option

  The default is a comma (,).
- Strings that mark the beginning of a column value, using the Prefix option

  The default is a double quotation mark (").
- Strings that mark the end of a column value, using the Suffix option

  The default is a double quotation mark (").

- The row termination character, using the Termination option

  The default is the end of the line.

You cannot use the same character string to represent both the null value and delimiter options. For example, if you use an asterisk to represent the null value, you cannot use it to represent the separator. You cannot specify a blank space or spaces as the null representation or as a delimiter option.

If the last columns of a row are null, the data file does not need to specify null representations for those columns. For example, the DEPARTMENTS table contains five columns but some rows in a data file contain only the first three columns. If you specify the null option, Oracle Rdb loads null values in the remaining columns.

The following example shows the depts.dat data file with two rows containing only three columns each:

```
"SDOC","Software Documentation","00375"
"QLTY","Quality Assurance","00418"
```

The following command loads the rows into the database, loading null values into the last two columns of the table:

```
$ RMU/LOAD /RECORD_DEFINITION = (FILE=depts.rrd, -
_$  FORMAT=DELIMITED_TEXT, NULL="")
_Root: mf_personnel
_Table: DEPARTMENTS
_Input file: depts.dat
%RMU-I-DATRECREAD,  2 data records read from input file.
%RMU-I-DATRECSTO,   2 data records stored.
```

6.6.5 Unloading Null Values

As with RMU Load, when you use the Record_Definition qualifier with the RMU Unload command, null values for a column are not loaded into an Oracle Rdb database, unless you use the Delimited_Text qualifier with the Null option.

You can specify that the null representation be a character string or an empty string. Example 6–12 shows how to unload null values from the DEPARTMENTS table and specify that the null values be represented by an asterisk (*).

Example 6–12 Unloading Null Values

```
$ RMU/UNLOAD/RECORD_DEFINITION = (FILE=depts.rrd, -
_$ FORMAT=DELIMITED_TEXT, NULL="*")
_Root: mf_personnel
_Table:  DEPARTMENTS
_Output file: depts.dat
%RMU-I-DATRECUNL,   26 data records unloaded.
```

In the following excerpt of the unloaded depts.dat file, the commas are field separators. The two asterisks at the end of each line are the null representation:

```
"ADMN","Corporate Administration      ","00225",*,*
"ELEL","Electronics Engineering       ","00188",*,*
"ELGS","Large Systems Engineering     ","00369",*,*
"ELMC","Mechanical Engineering        ","00190",*,*
"ENG ","Engineering                   ","00471",*,*
   .
   .
   .
```

6.6.6 Restructuring Databases Using the RMU Load and RMU Unload Commands

You can use the RMU Load and RMU Unload commands to help you restructure a database. For example, suppose you want to expand the COLLEGES table to include all colleges and universities throughout the world. For the United States and Canada alone, this list exceeds 3000 rows, and worldwide it may be close to 12,000 rows.

In researching this task, you discover that the following alterations to the row definition and the column definitions are necessary to internationalize the COLLEGES table:

- Add a country column
- Increase the size of the postal code column
- Increase the size of the college name column to include the entire name of the institution with minimal abbreviations
- Increase the size of the college code column

After you determine what changes to make to the old row and column definitions, you define a new database table, calling it the ALL_COLLEGES table, and a new record definition file, calling it the all_colleges.rrd file. Next, create the flat data file or unload it from an existing database. Then, sort the rows in a specified order perhaps based on the country name and the college

name because, after the load operation, you want to create a sorted index on these two columns to guarantee fast row access for range retrievals.

The following example shows contents of the all_colleges.rrd file:

```
DEFINE FIELD COLLEGE_CODE DATATYPE IS TEXT SIZE IS 8.
DEFINE FIELD COLLEGE_NAME DATATYPE IS TEXT SIZE IS 40.
DEFINE FIELD CITY DATATYPE IS TEXT SIZE IS 20.
DEFINE FIELD POSTAL_CODE DATATYPE IS TEXT SIZE IS 10.
DEFINE FIELD REGION_STATE DATATYPE IS TEXT SIZE IS 20.
DEFINE FIELD COUNTRY_NAME DATATYPE IS TEXT SIZE IS 15.
DEFINE RECORD ALL_COLLEGES.
   COLLEGE_CODE .
   COLLEGE_NAME .
   CITY .
   POSTAL_CODE .
   REGION_STATE .
   COUNTRY_NAME .
END ALL_COLLEGES RECORD.
```

Note that the row size has expanded from 56 bytes to 113 bytes and the table now includes six fields. Suppose you unloaded the data from an existing database table into a flat data file that includes four additional columns. Check that the column sizes and data types match with those in the database. Next, sort the data on the country name and college name fields within the country column using your operating system sort utility. If the column order of the input file does not match that defined for the table in the database, you can use the Corresponding qualifier to specify that Oracle RMU load the columns in the order specified by the .rrd file. To load the ALL_COLLEGES table, enter the command shown in Example 6–13.

Example 6–13 Loading a Table Using the RMU Load Command

```
$ RMU/LOAD /CORRESPONDING /RECORD_DEFINITION=FILE=all_colleges.rrd -
_$ /TRANSACTION_TYPE=EXCLUSIVE /COMMIT_EVERY=500 /LOG_COMMITS -
_$ mf_personnel ALL_COLLEGES all_colleges3.unl
%RMU-I-DATRECSTO, 500 data records stored
%RMU-I-DATRECSTO, 1000 data records stored
   .
   .
   .
%RMU-I-DATRECSTO, 12000 data records stored
%RMU-I-DATRECSTO, 12000 data records stored
```

If you specify the share mode Exclusive in the Transaction_Type qualifier as shown in Example 6–13, the load operation becomes faster because no snapshot file I/O operations occur. Oracle RMU displays a message for each cumulative

group of 500 records stored and committed and the last message indicates that 12,000 data rows are stored and committed to the database table.

Note that the load operation may fail if the column data types and column sizes of the database table do not match the columns in the unloaded flat file. Although Oracle Rdb converts the data as it loads it, the conversion may fail because of incompatible lengths, precision, or values. (See the RMU Load section in the *Oracle RMU Reference Manual* for more information about data type conversion.) Basically, you have the following two choices:

- Modify the database table definition to match the desired columns in the flat file and make further changes within the database itself.
- Modify the desired columns in the flat file to match the database table definition.

The first case is the preferred method because you modify only the domain definitions in the database definition before the load operation to accomplish this task.

6.6.7 Loading and Unloading Data from Oracle Rdb Databases

You can use the RMU Load and RMU Unload commands to restructure Oracle Rdb databases, archive data, and sort data by defining a view with a SORTED BY clause and unloading that view.

When you unload data from an Oracle Rdb database and load the same data into the same or another Oracle Rdb database, consider whether or not to use the Record_Definition qualifier. As Section 6.6.3 explains, this qualifier changes the characteristics of some types of data.

You can use the RMU Load and RMU Unload commands to restructure a database table. For example, suppose that you decide that you do not need an ADDRESS_DATA_2 column in the EMPLOYEES table. You can remove the column and the data, as Example 6–14 shows.

Example 6–14 Restructuring a Table Using the RMU Load and RMU Unload Commands

```
$ ! Unload the data from the table, except for the column ADDRESS_DATA_2.
$ !
$ RMU/UNLOAD/FIELDS=(EMPLOYEE_ID, LAST_NAME, FIRST_NAME, MIDDLE_INITIAL, -
_$  ADDRESS_DATA_1, CITY, STATE, POSTAL_CODE, SEX, BIRTHDAY, STATUS_CODE)
_Root:  mf_personnel
_Table: EMPLOYEES
_Output file:  employees.unl
$ !
$ SQL
SQL> ATTACH 'FILENAME mf_personnel';
SQL> CREATE TABLE EMPLOYEES_NEW
cont>   (
cont>   EMPLOYEE_ID      ID_DOM
cont>     PRIMARY KEY,
cont>   LAST_NAME        LAST_NAME_DOM,
cont>   FIRST_NAME       FIRST_NAME_DOM,
cont>   MIDDLE_INITIAL   MIDDLE_INITIAL_DOM,
cont>   ADDRESS_DATA_1   ADDRESS_DATA_1_DOM,
cont>   CITY             CITY_DOM,
cont>   STATE            STATE_DOM,
cont>   POSTAL_CODE      POSTAL_CODE_DOM,
cont>   SEX              SEX_DOM,
cont>     CHECK          (SEX IN ('M', 'F', '?') )
cont>     CONSTRAINT     EMP_SEX_VAL,
cont>   BIRTHDAY         DATE_DOM,
cont>   STATUS_CODE      STATUS_CODE_DOM,
cont>     CHECK          (STATUS_CODE IN ('0', '1', '2', 'N') )
cont>     CONSTRAINT     EMP_STATUS_CODE_VAL
cont>   );
SQL> COMMIT;
SQL>  EXIT
$
$ RMU/LOAD personnel
_Table: EMPLOYEES_NEW
_Input file: employees.unl
%RMU-I-DATRECSTO, 100 data records stored
```

6.6.8 Loading Data from One Database to Another

You can use the RMU Load and RMU Unload commands to move data from one Oracle Rdb database to another, even if the databases do not have the same tables or the same columns in the tables.

The RMU Extract command makes it easier to unload data from one database and load it into another, as the following steps describe:

1. Use the RMU Extract command to generate a command procedure to unload data from one database. The command procedure contains RMU Unload commands for each table in the database.
2. If any of the tables in the database from which you are unloading data has *more* columns than the corresponding table in the database into which you are loading the data, edit the command procedure to include the Fields qualifier to the RMU Unload command for that table. The Fields qualifier unloads only those fields (columns) that you specify.
3. Execute the command procedure to unload the data from one database. RMU unloads data from each table into a .unl file.
4. Use the RMU Extract command to generate a command procedure to load the data into the other database. The command procedure contains RMU Load commands for each table in the database.
5. If any of the tables in the database from which you are unloading data has *fewer* columns than the corresponding table in the database into which you are loading the data, you must create views in the database and edit the command procedure to load the data into the views.
6. If the names of the tables in the two databases are different, edit the command procedure to change the name of the table in the command procedure so that it corresponds to the name of the table in the new database.
7. Evaluate whether or not constraints in the second database will prevent you from loading data into a table. If they will, temporarily drop the constraints.

   When RMU Extract extracts table definitions, it extracts primary, unique, and not null constraint definitions into CREATE TABLE statements. Because foreign key or check constraints can refer to other tables, RMU Extract extracts the definitions for those constraints into ALTER TABLE statements. The ALTER TABLE statements appear after all the CREATE TABLE statements, so that the command procedure can be executed without editing.
8. Execute the command procedure to load the data.
9. Delete any temporary views.
10. If you dropped any constraints, update the data to meet the conditions of those constraints and add the constraints to the table.

The following examples explain how to unload data from the personnel database and load it into the multischema corporate_data database.

Example 6–15 shows how to use RMU Extract to create a command procedure to unload data from the personnel database.

Example 6–15 Creating a Command Procedure to Unload Data

```
$ RMU/EXTRACT/ITEMS=UNLOAD /OUTPUT= unload.com
_Root File: mf_personnel
```

The command procedure created by the RMU Extract command consists of RMU Unload commands for every table in the database.

Because the databases are not identical, you must make some adjustments after you unload the data. For example, the corporate_data database contains two DEPARTMENTS tables, one in the personnel schema and one in the ACCOUNTING schema. The DEPARTMENTS table in the ACCOUNTING schema contains five columns as does the DEPARTMENTS table in the personnel database; the DEPARTMENTS table in the PERSONNEL schema of the corporate_data database contains only two columns.

Because there are two DEPARTMENTS tables in corporate_data and they do not contain the same columns, you must copy the RMU Unload command for the DEPARTMENTS table and specify a different name for one of the .unl files. Because the DEPARTMENTS table in the personnel database contains *more* columns than the DEPARTMENTS table in the PERSONNEL schema of the corporate_data database, you edit the command procedure to include the Fields qualifier in the command line that unloads that table.

The following example shows an excerpt of the command procedure created by RMU Extract, including the changes made to the command lines that unload data from the DEPARTMENTS table and the RESUMES table:

```
$ ! Unload only two fields from the DEPARTMENTS table.
$ !
$ RMU/UNLOAD personnel.rdb -
        /FIELDS=(DEPARTMENT_CODE,DEPARTMENT_NAME) -
        DEPARTMENTS -
        departments.unl
$ ! Add another command to unload the DEPARTMENTS table to another .unl file.
$ ! Unload all the columns because this .unl file will load the
$ ! DEPARTMENTS table in the ACCOUNTING schema. Give the .unl file
$ ! a different name.
$ !
$ RMU/UNLOAD personnel.rdb -
        DEPARTMENTS -
        departments_accounting.unl
```

```
.
.
.
$ ! Unload only two fields from the RESUMES table.
$ !
$ RMU/UNLOAD personnel.rdb -
        /FIELDS=(EMPLOYEE_ID, RESUME) -
        RESUMES -
        resumes.unl
```

When you run the procedure, Oracle RMU unloads data from all tables in the database (twice from the DEPARTMENTS table), as Example 6–16 shows.

Example 6–16 Unloading Data Using the RMU Unload Command

```
$ @UNLOAD
%RMU-I-DATRECUNL, 3 data records unloaded
%RMU-I-DATRECUNL, 100 data records unloaded
%RMU-I-DATRECUNL, 26 data records unloaded
%RMU-I-DATRECUNL, 26 data records unloaded
%RMU-I-DATRECUNL, 274 data records unloaded
%RMU-I-DATRECUNL, 729 data records unloaded
%RMU-I-DATRECUNL, 15 data records unloaded
%RMU-I-DATRECUNL, 165 data records unloaded
%RMU-I-DATRECUNL, 3 data records unloaded
%RMU-I-DATRECUNL, 3 data records unloaded
$
```

Example 6–17 shows how to use RMU Extract to create a command procedure to load data into a database.

Example 6–17 Creating a Command Procedure to Load Data

```
$ RMU/EXTRACT/ITEMS=LOAD /OUTPUT= load.com
_Root File: corporate_data
```

Because the JOB_HISTORY table in the personnel database has fewer columns than the JOB_HISTORY table in the corporate_data database, you must create a temporary view in corporate_data that contains only as many columns as the JOB_HISTORY table in personnel. The following example shows how to create that view:

```
SQL> ATTACH 'FILENAME corporate_data';
SQL>
SQL> CREATE VIEW administration.personnel.JOB_HIST_TEMP
cont>   AS SELECT
cont>      EMPLOYEE_ID,
cont>      JOB_CODE,
cont>      JOB_START,
cont>      JOB_END,
cont>      DEPARTMENT_CODE,
cont>      SUPERVISOR_ID
cont> FROM ADMINISTRATION.PERSONNEL.JOB_HISTORY;
```

Similarly, you must create a temporary view for the DEGREES table.

Because the WORK_STATUS table in the corporate_data database contains a check constraint with values that are different from the values in the WORK_STATUS table in the personnel database, modify the table in corporate_data and drop the constraint.

Then, edit the command procedure, substituting the name of the view for the name of the table. The command procedure loads the data through the view into the underlying table.

Because some table names are not the same in both databases, you must edit the RMU Load commands for those tables, changing the names of the .unl files in the load procedure to reflect the names of the actual .unl files. The following example shows excerpts of the command procedure with these modifications:

```
$ ! Substitute the name of the view for the name of the table.
$ RMU/LOAD -
        /TRANSACTION_TYPE = EXCLUSIVE -
        corporate_data -
        JOB_HIST_TEMP -
        job_history.unl
   .
   .
   .
$ !
$ ! Substitute the name of the view for the name of the table.
$ RMU/LOAD -
        /TRANSACTION_TYPE = EXCLUSIVE -
        corporate_data -
        DEGREES_TEMP -
        degrees.unl
   .
   .
   .
```

```
$ ! Change the name of the .unl file.
$ RMU/LOAD -
        /TRANSACTION_TYPE = EXCLUSIVE -
        corporate_data -
        PAYROLL -
        jobs.unl
$ !
$ ! Because corporate_data is a multischema database and it has two
$ ! DEPARTMENTS tables in two schemas, Oracle RMU uses the stored name
$ ! DEPARTMENTS1 to designate the DEPARTMENTS table in the ACCOUNTING
$ ! schema. Change the name of the .unl file for this table to reflect
$ ! the name of the .unl file you used in the UNLOAD.COM procedure.
$ RMU/LOAD -
        /TRANSACTION_TYPE = EXCLUSIVE -
        corporate_data  -
        DEPARTMENTS1 -
        departments_accounting.unl
$ !
```

In addition to the changes shown in the preceding example, the commands that load the CANDIDATES and RESUMES tables have been deleted. Because the CANDIDATES table in the corporate_data database contains a primary key, CANDIDATE_ID, that does not exist in the personnel database, you cannot load that table without making further changes to the data. For example, you could modify the table to delete the primary key, load the table with the RMU Load procedure, update the rows to add data for the CANDIDATE_ID column, and then modify the table to add the primary key constraint. However, it is probably easier to load the CANDIDATES table by using an application program rather than the RMU Load command.

Because the RESUMES table in the corporate_data database contains a foreign key based on CANDIDATES_ID and because the command procedure does not load the CANDIDATES table, you cannot load the RESUMES table without making further changes. Again, it is probably easier to load the RESUMES table by using an application program rather than the RMU Load command.

After you edit the command procedure, execute it to load data into the database, as shown in Example 6–18.

Example 6–18 Loading Data Using the RMU Load Command

```
$ @LOAD
%RMU-I-DATRECSTO, 100 data records stored
%RMU-I-DATRECSTO, 26 data records stored
%RMU-I-DATRECSTO, 274 data records stored
%RMU-I-DATRECSTO, 729 data records stored
%RMU-I-DATRECSTO, 15 data records stored
%RMU-I-DATRECSTO, 165 data records stored
%RMU-I-DATRECSTO, 3 data records stored
%RMU-I-DATRECSTO, 15 data records stored
%RMU-I-DATRECSTO, 26 data records stored
```

After you load the data, you can delete the temporary views. If you dropped any constraints, update the data to meet the constraints and define the constraints again.

6.7 Using Parallel Load

With the RMU Load command, you can use multiple processes to load data in parallel. A parallel load operation uses your process to read the input file and uses one or more executors to load the data into the target table. This results in concurrent read and write operations and, in many cases, substantially improves the performance of the load operation.

On OpenVMS, an **executor** is a detached process if you have the OpenVMS DETACH privilege. If you do not have the DETACH privilege, an executor is a subprocess. ♦

On Digital UNIX, an **executor** is a forked (detached) process. ♦

A parallel load operation is effective especially when you have large partitioned tables that contain no indexes (except indexes that follow the partitioning) and no constraints, triggers or list data. (You cannot use parallel load to load list data or security auditing data.)

When you use parallel load, your process serves as the load operation execution manager for the number of executors you specify. RMU Load maps each executor to one or more storage areas (partitions). For example, if you specify two executors, RMU Load takes the following actions:

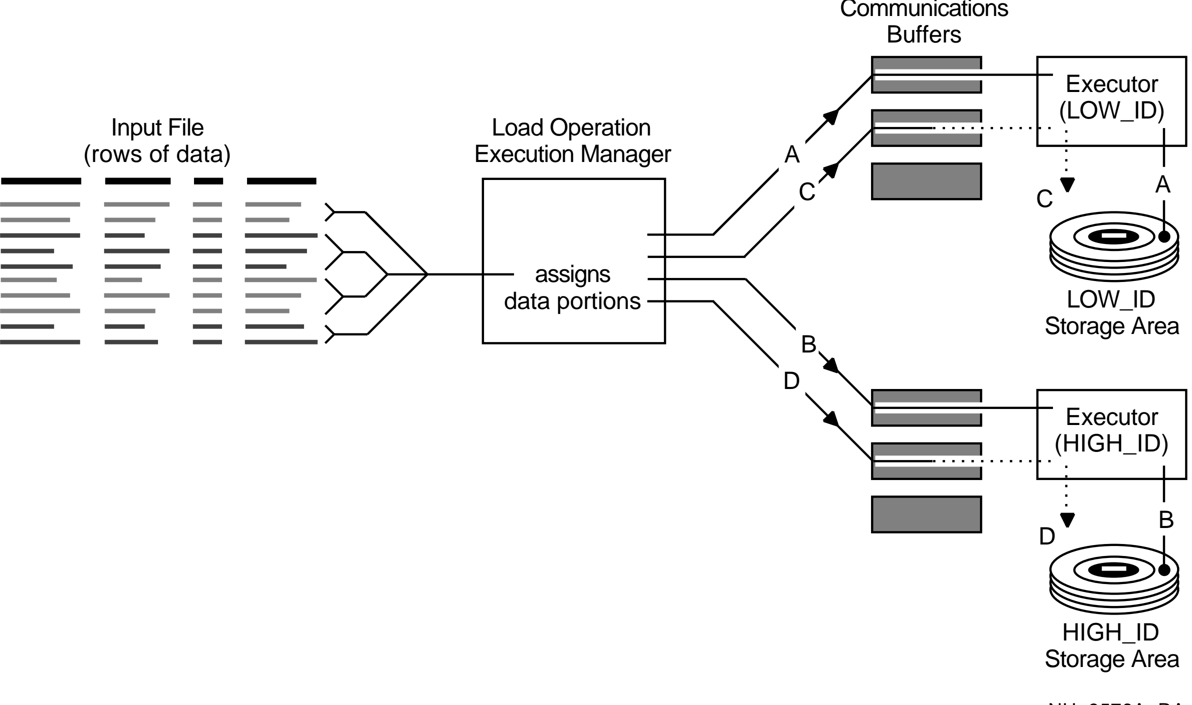
1. The execution manager reads a portion of the input file, determines the appropriate executor (based on the partitioning criteria), and places the rows in a communications buffer for the appropriate executors to load.

   By default, Oracle RMU assigns four communications buffers to each executor and one executor for the load operation.

2. The executors begin to load the data from their first communications buffer, as your process reads the next portion of data.
3. If the execution manager has another portion of data ready for loading before an executor has completed loading its first portion of data (that is, its first communications buffer still contains data), the execution manager places the next portion of data in the second communications buffer of the appropriate executor.
4. The executor begins to load the data from its second communications buffer, as the execution manager reads the next portion of data.
5. If an executor has not completed loading the second portion of data, the execution manager places the next portion of data in the third communications buffer of the appropriate executor.
6. The steps are repeated, using the next available communications buffer, until all data is loaded into the table.

Figure 6–1 demonstrates a parallel load using two executors with three communications buffers each.

Figure 6–1 Using Parallel Load



When you use parallel load, keep in mind the following guidelines:

- You can specify the number of executors and the number of communications buffers. Usually, the number of executors should be the same as the number of partitions for the table. For example, the EMPLOYEES table in the mf_personnel database contains three partitions, EMPIDS_LOW, EMPIDS_MID, and EMPIDS_OVER. To load the EMPLOYEES table, you should specify three executors.
- If the partitioning is random or uses vertical partitioning, Oracle RMU creates only one executor, no matter how many you specify. (The partitioning is random if you created the storage map with the STORE RANDOMLY ACROSS clause.)
- If you are loading data that is already sorted based on the partitions, you should use only one executor. In this situation, using more executors does not speed up the load operation because only one executor at a time is processing data. For example, if the partitioning is based on the

column EMPLOYEE_ID and the data is sorted by EMPLOYEE_ID, the execution manager places all the data from the first portion of data in the communications buffers of the first executor and none in the buffers of second or third executor.

If the data was unloaded using the RMU Unload command and you are loading the data into a table that is partitioned based on the same limits as it was when it was unloaded, you should use only one executor. When RMU Unload unloads data from partitions, it unloads one partition at a time into the data file.

- You cannot use parallel load to load list data or security audit data.

To improve the performance of the parallel load operation, review the guidelines for improving load performance in Section 6.1 and Section 6.6.1. Most of the guidelines, except for the recommendations regarding sorting data before the load, apply to parallel load as well as single-process load operations. In particular, note the following guidelines for parallel load operations:

- Increase the memory available to the load process by increasing the working set parameters of the load process.
- Increase the database buffer count.
- Use a multiprocessor system.
- The performance of the parallel load operation improves if you base the partitioning criteria on one integer column. For example, if you create the storage map with a STORE USING clause that specifies more than one column, it may counteract some of the advantages of parallel load.
- If your system becomes CPU-bound during a parallel load, consider using fewer executors and balancing the storage areas so that each executor has approximately the same amount of work to perform.
- When trying to make the best use of memory, increase the value of Buffer_Count before increasing Row_Count.
- If the table has indexes that use partitioning criteria that does not match the partitioning criteria of the data, consider using the Defer_Index_Updates qualifier. When you have secondary indexes, the executors may lock shared index data as they are loading rows. The Defer_Index_Updates qualifier, which defers the building of secondary indexes until commit time, can result in less locking as well as reduced I/O.

The Defer_Index_Updates qualifier is only useful when there is no other activity in the database table, no constraints are defined on the table or you specify the Noconstraints qualifier, no triggers are defined on the table, and when not all the indexes are unique. Consider using this qualifier for the initial load of a table. If all indexes are partitioned like the data, Oracle Rdb ignores the Defer_Index_Updates qualifier.

If you use the Defer_Index_Updates qualifier when there is other activity in the database table, it may result in lock conflicts, thus terminating the entire operation, rather than a single row. (With the Defer_Index_Updates qualifier, lock conflicts are not detected until commit time.) Consider using the Nodefer_Index_Updates qualifier (the default) for daily updates of a table, when there is likely to be other database activity.

To help troubleshoot the performance of parallel load operations, read Section 6.4 before beginning to load data.

Oracle RMU provides two ways to specify how the parallel load operation should be performed:

- Specifying the options on the RMU command line

  See Section 6.7.1 for more information.

- Specify the options in a plan file

  A **plan file** specifies how a load operation should be performed and the directories in which to place files associated with the load operation. Each portion of the plan file describes the execution of a separate executor process. (You can use a plan file for a non-parallel load as well as a parallel load.)

  See Section 6.7.3 for more information.

6.7.1 Using Parallel Load Without a Plan File

When you use parallel load without a plan file, you specify the parallel load options on the Oracle RMU command line. In addition to other RMU Load qualifiers, you can specify the following options to the Parallel qualifier:

- Executor_Count specifies the number of executors assigned to the load operation.

- Buffer_Count specifies the number of communications buffers assigned to each executor.

Example 6–19 shows how to use parallel load to unload the JOB_HISTORY table of the mf_personnel database. Because JOB_HISTORY is partitioned into three storage areas, the command line specifies three executors. Because JOB_HISTORY contains a non-unique index other than the placement index, the command line specifies the Defer_Index_Updates qualifier. Furthermore, because we are certain that the data does not violate any constraints, the command line specifies the Noconstraints qualifier.

Example 6–19 Using Parallel Load

```
$ RMU/LOAD/PARALLEL= (BUFFER_COUNT=3, EXECUTOR_COUNT=3) -
_$   /DEFER_INDEX_UPDATES /NOCONSTRAINTS -
_$   /COMMIT_EVERY=500  /TRANS=EXCLUSIVE
_Root: mf_personnel
_Table: JOB_HISTORY
_Input file: job_history.unl
%RMU-I-EXECUTORMAP, Executor EXECUTOR_1 (pid: 27C15142) will load storage area
EMPIDS_LOW.
%RMU-I-EXECUTORMAP, Executor EXECUTOR_2 (pid: 27C13D46) will load storage area
EMPIDS_MID.
%RMU-I-EXECUTORMAP, Executor EXECUTOR_3 (pid: 27C1514A) will load storage area
EMPIDS_OVER.

%RMU-I-EXECSTAT0, Statistics for EXECUTOR_1:
%RMU-I-EXECSTAT1,   Elapsed time:  00:00:30.57   CPU time:       2.99
%RMU-I-EXECSTAT2,   Storing time:  00:00:03.94   Rows stored:    807
%RMU-I-EXECSTAT3,   Commit time:   00:00:14.33   Direct I/O:     492
%RMU-I-EXECSTAT4,   Idle time:     00:00:06.11   Early commits:  1

%RMU-I-EXECSTAT0, Statistics for EXECUTOR_2:
%RMU-I-EXECSTAT1,   Elapsed time:  00:00:30.36   CPU time:       3.32
%RMU-I-EXECSTAT2,   Storing time:  00:00:05.69   Rows stored:    944
%RMU-I-EXECSTAT3,   Commit time:   00:00:07.37   Direct I/O:     708
%RMU-I-EXECSTAT4,   Idle time:     00:00:03.12   Early commits:  1

%RMU-I-EXECSTAT0, Statistics for EXECUTOR_3:
%RMU-I-EXECSTAT1,   Elapsed time:  00:00:29.69   CPU time:       1.72
%RMU-I-EXECSTAT2,   Storing time:  00:00:01.35   Rows stored:    157
%RMU-I-EXECSTAT3,   Commit time:   00:00:02.47   Direct I/O:     226
%RMU-I-EXECSTAT4,   Idle time:     00:00:10.91   Early commits:  0

%RMU-I-EXECSTAT5,   Main process idle time:  00:00:25.69
%RMU-I-DATRECREAD,  1908 data records read from input file.
%RMU-I-DATRECSTO,   1908 data records stored.
```

6.7.2 Generating a Plan File with RMU Load

A plan file lets you specify the parallel load options in a file rather than on the Oracle RMU command line. Plan files are useful when you repeatedly load large amounts of data into a table.

Although you can construct your own plan file, it is easier to let Oracle RMU construct an initial plan file for you. To do so, use the List_Plan and Noexecute qualifiers to the RMU Load command. Specify other needed qualifiers, as shown in the Example 6–20.

Example 6–20 Generating a Plan File

```
$ RMU/LOAD/PARALLEL= (BUFFER_COUNT=3, EXECUTOR_COUNT=3)  -
_$ /DEFER_INDEX_UPDATES /NOCONSTRAINTS -
_$ /COMMIT_EVERY=500 -
_$ /LIST_PLAN=JOB_HISTORY / NOEXECUTE
_Root:  mf_personnel
_Table: JOB_HISTORY
_Input file: job_history.unl
```

The example generates the plan file job_history.plan, which contains the qualifiers you specify on the command line and the default values for other qualifiers. The following example shows an excerpt of the plan file:

```
! Plan created on 12-SEP-1995 by RMU/LOAD.

Plan Name = LOAD_PLAN
Plan Type = LOAD


Plan Parameters:
    Database Root File = MF_PERSONNEL
    Table Name = JOB_HISTORY
    Input File = JOB_HISTORY.UNL

    ! Fields = <all>
    Transaction_Type = PROTECTED
    ! Buffers = <default>
    Commit_Every = 500
    Row_Count = 50
    ! Skip = <none>
    NoLog_Commits
    NoCorresponding
    Defer_Index_Updates
    NoConstraints
    Parallel
    NoPlace
    ! Statistics = <none>
    NoTrigger_Relations
End Plan Parameters
```

```
Executor Parameters:
    Executor Name = EXECUTOR_1
    ! Place_Only = <none>
    ! Exception_File = <none>
    ! RUJ Directory = <default>
    Communication Buffers = 3
End Executor Parameters
   .
   .
   .
```

Because Example 6–20 contains the Noexecute qualifier, Oracle Rdb does not execute the plan file; it only generates it. You can use the generated plan file as a starting point for building a load operation that is optimized for your database and system. You can edit the plan file, including specifying locations for the following files for each executor:

- Placement-only files
- Exception files
- The .ruj directory

To reduce disk I/O contention, you should specify that the .ruj directory and placement-only files for each executor reside on separate disks. Consider renaming the executors so that the names can be easily mapped to the storage areas they represent.

The following example shows an excerpt of the edited version of the job_history.plan file:

```
Executor Parameters:
    Executor Name = EXEC_LOW
    ! Place_Only = <none>
    ! Exception_File = <none>
    RUJ Directory = DBA:[PERS_DB.RUJ]
    Communication Buffers = 3
End Executor Parameters
```

In the previous example, the executor name is changed and .ruj directory is specified.

6.7.3 Using Parallel Load with a Plan File

When you have modified the plan file to your satisfaction, you are ready to load the data using the RMU Load Plan command.

Because the plan file contains information that you would normally enter at the command line, you cannot specify RMU Load qualifiers other than List_ Plan and Noexecute. Remember that you can specify command line options in the plan file, along with options that are not available on the command line. Example 6–21 shows how to load a table using the plan file generated and modified in Section 6.7.2.

Example 6–21 Using a Plan File for Parallel Load

```
$ RMU /LOAD /PLAN job_history.plan
%RMU-I-EXECUTORMAP, Executor EXEC_LOW (pid: 27C0BA4E) will load storage area
EMPIDS_LOW.
%RMU-I-EXECUTORMAP, Executor EXEC_MID (pid: 27C13C52) will load storage area
EMPIDS_MID.
%RMU-I-EXECUTORMAP, Executor EXEC_HIGH (pid: 27C0B453) will load storage area
EMPIDS_OVER.

%RMU-I-EXECSTAT0, Statistics for EXEC_LOW:
%RMU-I-EXECSTAT1,   Elapsed time:  00:00:34.52   CPU time:       3.30
%RMU-I-EXECSTAT2,   Storing time:  00:00:05.66   Rows stored:    807
%RMU-I-EXECSTAT3,   Commit time:   00:00:07.86   Direct I/O:     696
%RMU-I-EXECSTAT4,   Idle time:     00:00:08.52   Early commits:  1

%RMU-I-EXECSTAT0, Statistics for EXEC_MID:
%RMU-I-EXECSTAT1,   Elapsed time:  00:00:34.93   CPU time:       3.59
%RMU-I-EXECSTAT2,   Storing time:  00:00:09.02   Rows stored:    944
%RMU-I-EXECSTAT3,   Commit time:   00:00:08.22   Direct I/O:     936
%RMU-I-EXECSTAT4,   Idle time:     00:00:07.49   Early commits:  1

%RMU-I-EXECSTAT0, Statistics for EXEC_HIGH:
%RMU-I-EXECSTAT1,   Elapsed time:  00:00:33.90   CPU time:       1.84
%RMU-I-EXECSTAT2,   Storing time:  00:00:02.78   Rows stored:    157
%RMU-I-EXECSTAT3,   Commit time:   00:00:02.20   Direct I/O:     229
%RMU-I-EXECSTAT4,   Idle time:     00:00:18.94   Early commits:  0

%RMU-I-EXECSTAT5,   Main process idle time:  00:00:28.43
%RMU-I-DATRECREAD,  1908 data records read from input file.
%RMU-I-DATRECSTO,   1908 data records stored.
```

6.8 Modifying Database Definitions Following a Load Operation

You may need to make the following changes to your database definitions following a successful load operation:

- Change the buffer size back to its original defined value if you changed this value in an SQL ALTER DATABASE statement prior to the load operation
- Define other indexes
- Define constraints

- Define table-level constraints and triggers
- Define the COMPUTED BY columns in your tables

As you begin to analyze your database after it has gone into a production environment, you may need to do some additional tuning, depending on the requirements of the application. Some of these changes may include:

- Adding more memory buffers
- Disabling compression
- Modifying the B-tree node characteristics—fill percentage and node size
- Reorganizing specific storage areas

See the *Oracle Rdb7 Guide to Database Performance and Tuning* for additional information on enhancing the performance of your database application.

7

Modifying Databases and Storage Areas

This chapter explains how you can modify database characteristics after the database has been created. This chapter describes the following:

- A summary of options for modifying databases and storage areas
- Which data definitions you can modify while other users are attached to a database
- Modifying the characteristics of a database, including database-wide parameters
- Modifying the requirement for using or not using the repository
- Modifying storage areas, adjusting storage area parameters, and deleting storage areas
- Modifying and deleting indexes
- Modifying and deleting storage maps
- Reorganizing databases
- Making copies of databases
- Moving databases and database files
- Deleting databases
- Migrating databases to different versions of Oracle Rdb

This chapter focuses primarily on modifying (altering or deleting) aspects of the physical database design. For information on modifying database elements such as tables and domains, see Chapter 8.

Most metadata updates are journaled in both recovery-unit journal files (file type .ruj) and after-image journal files (file type .aij). Oracle Rdb creates .ruj files by default; however, it does not create .aij files by default. Section 7.4.1 explains how to create .aij files and enable after-image journaling.

7.1 Modifying Databases and Storage Areas — A Summary

If your database has not been loaded with data, you can easily modify database file characteristics and how tables and indexes are stored in different storage areas of a multifile database. Simply delete an existing database and execute a new CREATE DATABASE statement.

After a database contains data, it may not be convenient to unload each table into data files, delete the database (which also deletes data), enter a new CREATE DATABASE statement, and reload tables from data files. Therefore, Oracle Rdb provides the following commands and statements to modify the storage characteristics of an existing database and reload data according to your altered plan:

- An RMU Unload command followed by an RMU Load command

  You can use the RMU Unload and RMU Load commands to unload and load data while restructuring a database. For example, with these commands you can modify the number and order of columns in a table. Chapter 6 discusses these statements in more detail.

- An EXPORT statement followed by an IMPORT statement

  The EXPORT statement stores all database definitions and data in an interchange file (file type .rbr). The IMPORT statement uses both an .rbr file and the storage adjustments that you specify to create the database again and reload the database.

  When you use the EXPORT and IMPORT statements, you unload and then reload your entire database (unless you use the NO DATA option). However, the IMPORT statement offers you the convenience of specifying only the database elements that you want to modify rather than a complete database definition. Furthermore, when you use the EXPORT and IMPORT statements, you do not have to unload and reload tables from and to multiple data files.

  To ensure that other users do not attach to the database before the IMPORT operation completes, Oracle Rdb restricts other users from accessing the database. (By default, Oracle Rdb uses the RESTRICTED ACCESS clause in the IMPORT statement.)

  You can use the EXPORT and IMPORT statements to add storage maps for tables that contain data.

- An ALTER DATABASE statement

  You can use ALTER DATABASE statements to create new storage areas and modify or drop existing ones.

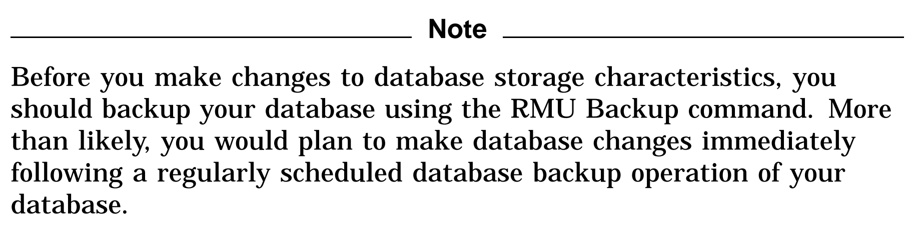
- An ALTER STORAGE MAP statement

You can use ALTER STORAGE MAP statements to make existing storage maps refer to new storage areas that you created in a preceding ALTER DATABASE statement or to modify existing storage maps so that they refer to both existing and new storage areas.

- An ALTER INDEX statement

  You can enter ALTER INDEX statements to modify where indexes are stored.

Table 7–1 summarizes which statements you use to accomplish various objectives.



Note

Before you make changes to database storage characteristics, you should backup your database using the RMU Backup command. More than likely, you would plan to make database changes immediately following a regularly scheduled database backup operation of your database.

You can accomplish all the objectives in the table using the EXPORT and IMPORT statements. However, the EXPORT and IMPORT statements unload and then reload an *entire* database. Storage media limitations and database availability requirements at your site can make a complete reload of a database very inconvenient. Therefore, if you simply want to reorganize data and modify characteristics for a subset of database files, consider the alternatives to the EXPORT and IMPORT statements.

Table 7–1 clarifies when a given objective requires you to use the EXPORT and IMPORT statements and when you can enter other statements to adjust database characteristics.

Table 7–1 Adjusting Storage and Memory Use Parameters

To Perform the Following Operation:	Enter the Following Statements:	In Which Case, You Reload:
Modify database from single-file to multifile, or vice versa	1. EXPORT 2. IMPORT (to define or delete storage areas, maps, and so forth)	Entire database
Modify DICTIONARY clause	1. ALTER DATABASE (specify only DICTIONARY clause)	Not applicable
Modify NUMBER OF USERS and NUMBER OF CLUSTER NODES	1. EXPORT 2. IMPORT (specify new values for NUMBER OF USERS and NUMBER OF CLUSTER NODES)	Entire database
Modify NUMBER OF USERS or NUMBER OF CLUSTER NODES (for multifile only)	1. ALTER DATABASE (specify new clauses directly subordinate to ALTER DATABASE)	Not applicable
Modify BUFFER SIZE, NUMBER OF BUFFERS, NUMBER OF RECOVERY BUFFERS, OPEN, RESERVE JOURNAL or STORAGE AREAS, ADD JOURNAL, ENABLE or DISABLE journaling, journal attributes	1. ALTER DATABASE (specify new clauses directly subordinate to ALTER DATABASE)	Not applicable
Move a table that has a storage map from one storage area to another existing area	1. ALTER STORAGE MAP (omit name of source area and specify name of target area)	Only specified table

(continued on next page)

Table 7–1 (Cont.) Adjusting Storage and Memory Use Parameters

To Perform the Following Operation:	Enter the Following Statements:	In Which Case, You Reload:
Move a table that has a storage map to a newly created area	1. ALTER DATABASE (add definition of storage area) 2. ALTER STORAGE MAP (omit name of source area and specify name of target area) 3. ALTER DATABASE (delete source area if it is empty and will not be used later)	Only specified table
Redistribute a table that has a storage map among original and newly created areas	1. ALTER DATABASE (add definition of storage area) 2. ALTER STORAGE MAP . . . REORGANIZE (specify names of original and new storage areas)	Only specified table
Add a partition to storage map	1. ALTER DATABASE (add definitions of new storage areas) 2. ALTER STORAGE MAP (partition across new areas only) 3. ALTER DATABASE (delete source storage areas if empty of other tables, indexes)	Only specified table

(continued on next page)

Table 7–1 (Cont.) Adjusting Storage and Memory Use Parameters

To Perform the Following Operation:	Enter the Following Statements:	In Which Case, You Reload:
Add a partition to storage map and reorganize storage areas	1. ALTER DATABASE (add definitions of new storage areas) 2. ALTER STORAGE MAP . . . REORGANIZE (partition across original and new areas) 3. ALTER DATABASE (delete source storage areas if empty of other tables, indexes)	Only specified table
Add a partition to storage map that does not contain an overflow partition	1. ALTER DATABASE (add definitions of new storage areas) 2. ALTER STORAGE MAP (add new areas)	Only specified table
Remove overflow partition from storage map	1. ALTER DATABASE (add definitions of new storage areas) 2. ALTER STORAGE MAP (omit OTHERWISE clause and add new areas)	Only specified table
Move a sorted index that was created with a STORE clause from one existing storage area to another	1. ALTER INDEX (specify only target area in STORE clause)	Only specified index
Move a sorted index to a newly created area	1. ALTER DATABASE (add definition of storage area) 2. ALTER INDEX (specify only new area in STORE clause)	Only specified index

(continued on next page)

Table 7–1 (Cont.) Adjusting Storage and Memory Use Parameters

To Perform the Following Operation:	Enter the Following Statements:	In Which Case, You Reload:
Modify or add partitioning to index definition (usually done in conjunction with table partitioning)	1. ALTER DATABASE (add definitions of new storage areas) 2. ALTER INDEX (partition across original new areas only) 3. ALTER STORAGE MAP for storage areas containing table for which index is defined (PLACEMENT VIA INDEX clause parallels newly partitioned index) 4. ALTER DATABASE (delete source storage area or areas if empty of other tables, indexes)	Only specified table, index, or both
Modify or add partitioning to index definition (usually done in conjunction with table partitioning) and reorganize storage areas	1. ALTER DATABASE (add definitions of new storage areas) 2. ALTER INDEX (partition across original and new areas) 3. ALTER STORAGE MAP . . . REORGANIZE for storage areas containing table for which index is defined (specify either original or new storage areas or both; PLACEMENT VIA INDEX clause parallels newly partitioned index) 4. ALTER DATABASE (delete source storage area or areas if empty of other tables, indexes)	Only specified table, index, or both
Modify whether a storage area is a read-only, read/write, or write-once storage area	1. RMU Move_Area or ALTER DATABASE (if *not* moving to or from a write-once storage area, alter the storage area)	Not applicable

(continued on next page)

Table 7–1 (Cont.) Adjusting Storage and Memory Use Parameters

To Perform the Following Operation:	Enter the Following Statements:	In Which Case, You Reload:
Modify EXTENT, ALLOCATION, CHECKSUM, cache name parameters of storage areas	1. ALTER DATABASE (specify new parameters in ALTER STORAGE AREA clause)	Not applicable
Modify SNAPSHOT ALLOCATION, SNAPSHOT EXTENT parameters of storage areas	1. ALTER DATABASE (specify new parameters in ALTER STORAGE AREA clause)	Not applicable
Modify basic characteristics (PAGE FORMAT, PAGE SIZE, INTERVAL, THRESHOLDS) of areas in which rows and indexes are stored	1. ALTER DATABASE (add new storage areas) 2. ALTER INDEX (specify only new areas in STORE clause) 3. ALTER STORAGE MAP (specify only new storage areas)	Only specified index and table
Modify storage map to add a PLACEMENT VIA INDEX clause	1. ALTER DATABASE (add new storage areas) 2. CREATE INDEX or ALTER INDEX (if the index is hashed, specify new areas in STORE clause) 3. ALTER STORAGE MAP (specify only new storage areas and PLACEMENT VIA INDEX clause)	Only specified index, table, or both

(continued on next page)

Table 7–1 (Cont.) Adjusting Storage and Memory Use Parameters

To Perform the Following Operation:	Enter the Following Statements:	In Which Case, You Reload:
Modify storage map to add a PLACEMENT VIA INDEX clause and reorganize	1. ALTER DATABASE (add new storage areas) 2. CREATE INDEX or ALTER INDEX (if the index is hashed, specify original and new areas in STORE clause) 3. ALTER STORAGE MAP . . . REORGANIZE (specify original and new storage areas and PLACEMENT VIA INDEX clause)	Only specified table, index, or both

Some entries in Table 7–1 specify that ALTER STORAGE MAP statements refer only to new areas rather than to areas where rows in a table are currently stored. By referring only to areas where rows are not currently stored, the ALTER STORAGE MAP statement specifies that you want all existing rows reloaded into the new areas. See Section 7.9 for information about how to specify whether all rows are reloaded into new areas or among original and new storage areas.

Note

You can modify some database and storage area characteristics using the RMU Restore command and a database backup file created as part of normal maintenance procedures.

However, the RMU Restore command is not intended to be a database restructuring tool. Any changes that you make to storage parameters using the RMU Restore command affect row storage only when existing rows are updated or new rows inserted. Using the RMU Restore command to modify THRESHOLDS clause values may result in immediate performance improvement for database update operations. Modifying other storage area parameters with the RMU Restore command results in performance improvement only if users most often access newly inserted or updated rows.

You can also modify some database and storage area characteristics using either the RMU Copy_Database or the RMU Move_Area

commands. See Section 7.11.3, Section 7.12, and Section 7.6.4 for more details.

7.2 Modifying Data Definitions While Users Are Attached to the Database

You can make many changes to the metadata even when other users are using the database. Generally, when you create, alter, or delete definitions, it is a good idea to reserve all affected tables using the exclusive share mode. Even if you do not explicitly specify a share mode, Oracle Rdb may automatically promote the share mode of your transaction to protected or exclusive when you modify database definitions. When you create indexes, use the shared data definition share mode, described in Section 3.14.5. See the *Oracle Rdb7 Guide to SQL Programming* for information on transaction share modes and lock types.

Table 7–2 and Table 7–3 list the metadata changes you can make while other users have active transactions on the database.

Additional restrictions regarding concurrent access to the database, as described in the comments column of these tables, may apply to the metadata changes even though the change can be made while the database is online. Note that additional restrictions unrelated to concurrent access are discussed in this manual and in the *Oracle Rdb7 SQL Reference Manual* and *Oracle RMU Reference Manual*.

Table 7–2 Updating Data Definitions While Users Are Attached to the Database

Metadata Update	Concurrency Allowed[1]	Comments
Catalogs CREATE DROP	Yes	You cannot delete a catalog when there are active transactions that access the catalog.

[1] *Concurrency Allowed* means other users can attach to the database while the metadata update is being performed. Note that other restrictions, as described in the Comments column of this table, may apply.

(continued on next page)

Table 7–2 (Cont.) Updating Data Definitions While Users Are Attached to the Database

Metadata Update	Concurrency Allowed[1]	Comments
Collating sequences CREATE ALTER DROP	Yes	You cannot delete a collating sequence if the database or domain in the database uses that collating sequence.
Constraints CREATE DROP	Yes	You cannot delete a constraint when there are active transactions that access the tables involved.
Domains CREATE ALTER DROP	Yes	You cannot alter a domain if stored routines use the domain.
External routines CREATE DROP	Yes	Refers to external procedures and functions.
Indexes CREATE ALTER DROP	Yes	You cannot disable an index or delete an index definition when there are active transactions that access the tables involved.
Modules CREATE DROP	Yes	Modules contain stored procedures and functions.
Outlines CREATE DROP	Yes	
Protection GRANT REVOKE	Yes	Granting or revoking a privilege takes effect after the user detaches and attaches to the database again.
Schemas CREATE DROP	Yes	You cannot delete a schema when there are active transactions that access the schema.
Storage areas RESERVE	No	This change is not journaled.

[1] *Concurrency Allowed* means other users can attach to the database while the metadata update is being performed. Note that other restrictions, as described in the Comments column of this table, may apply.

(continued on next page)

Table 7–2 (Cont.) Updating Data Definitions While Users Are Attached to the Database

Metadata Update	Concurrency Allowed[1]	Comments
CREATE ADD DROP	Yes	Concurrency is allowed if the database root file contains available slots; that is, slots that have been reserved for storage areas but not used. Updates are not seen by users currently attached to the database. New areas are seen when new users attach to the database after the change is committed. These metadata operations complete with an implicit commit operation.
ALTER	See comments	You can modify many of the storage area parameters. See Table 7–3 for specific information.
Storage maps CREATE ALTER DROP	Yes	
Tables CREATE ALTER DROP TRUNCATE	Yes	You cannot delete a table definition when there are active transactions that use the table.
Triggers CREATE DROP	Yes	You cannot delete a trigger definition when there are active transactions that use the trigger or that refer to the tables involved.
Views CREATE DROP	Yes	Deleting a view does not affect active users until you commit your transaction, users detach from the database, and then attach to the database again.
Databases CREATE DROP	No	These metadata updates complete with an implicit commit operation. If a user is attached to the database when you attempt to delete a database, you receive the -SYSTEM-W-ACCONFLICT, file access conflict error message.
ALTER	See comments	You can modify many of the database parameters, including storage area parameters. See Table 7–3 for specific information.

[1] *Concurrency Allowed* means other users can attach to the database while the metadata update is being performed. Note that other restrictions, as described in the Comments column of this table, may apply.

In addition to reviewing the information in Table 7–2, note that concurrent metadata statements may do the following:

- Return a LOCK CONFLICT ON CLIENT error message and cause an update failure if any table, index, constraint, trigger, or storage map is involved in a query and you attempt to create, alter, or drop the table, index, constraint, trigger, or storage map. Note that the query does not

have to be in use currently; it only needs to have been used during the current attach. All users accessing the elements must detach from the database before the create, alter, or drop operation can occur.

- Occasionally interfere with concurrent access to the system tables as well as to the table being modified. This is unavoidable and follows normal concurrent data access rules.
- Return an error message when interference occurs between a concurrent metadata operation and other database operations (such as creating, altering, or dropping an index, constraint, trigger, or storage map). Other users cannot execute a query that involves any metadata used in the CREATE, ALTER, or DROP statement until you have committed or rolled back the metadata update. Oracle Rdb returns a LOCK CONFLICT ON CLIENT error message or a normal deadlock or conflict error message (in the case of nowait transactions) because of the system table concurrency behavior.

Note that the locks involved in controlling updates to the metadata are not subject to the WAIT or NOWAIT clause of the SET TRANSACTION statement.

Table 7–3 shows which database-wide parameters you can modify while other users are attached to the database. Remember that you cannot create or drop a database while one or more users are attached to it.

Table 7–3 Updating to Database-Wide Parameters While Users Are Attached to the Database

Metadata Update	On Line [1]	Comments
Root File Parameters		
Open mode	Yes	Updates are not seen until a database open operation is required.
Wait interval for close	Yes	Updates do not take effect until the database is opened again after the change is completed. However, updates are not seen by users who attached to the database before the update.
Number of users	No	This change is not journaled.
Number of nodes	No	This change is not journaled.
Buffer size	No	

[1] *On Line* means other users can attach to the database while the metadata update is being performed. Other restrictions, as described in the Comments column of this table, may apply.

(continued on next page)

Table 7–3 (Cont.) Updating to Database-Wide Parameters While Users Are Attached to the Database

Metadata Update	On Line [1]	Comments
Root File Parameters		
Number of buffers	Yes	Updates are not seen by users currently attached to the database. Updates are seen when new users attach to the database after the change is completed.
Number of recovery buffers	Yes	Updates take effect when a new database recovery process begins.
Recovery-unit journal location	Yes	
Global buffers enabled or disabled	No	
Number of global buffers	Yes	Updates do not take effect until the database is opened again after the change is completed. However, updates are not seen by users who attached to the database before the update.
Maximum number of global buffers per user	Yes	Updates do not take effect until the database is opened again after the change is completed. However, updates are not seen by users who attached to the database before the update.
Page transfer	Yes	Updates are not seen by users currently attached to the database. Updates are seen when new users attach to the database after the change is completed.
Adjustable lock granularity	No	
Carry-over locks enabled or disabled	No	
Lock timeout interval	Yes	Updates are not seen by users currently attached to the database. Updates are seen when new users attach to the database after the change is completed.
Statistics enabled or disabled	No	
Cardinality collection enabled or disabled	Yes	
Workload collection enabled or disabled	Yes	

[1] *On Line* means other users can attach to the database while the metadata update is being performed. Other restrictions, as described in the Comments column of this table, may apply.

(continued on next page)

Table 7–3 (Cont.) Updating to Database-Wide Parameters While Users Are Attached to the Database

Metadata Update	On Line [1]	Comments
Root File Parameters		
Asynchronous batch-writes	Yes	Updates are not seen by users currently attached to the database. Updates are seen when new users attach to the database after the change is completed.
Asynchronous prefetch	Yes	Updates are not seen by users currently attached to the database. Updates are seen when new users attach to the database after the change is completed.
Detected asynchronous prefetch	Yes	Updates are not seen by users currently attached to the database. Updates are seen when new users attach to the database after the change is completed.
Incremental backup	Yes	
Lock partitioning	No	
Metadata changes enabled or disabled	Yes	Updates are not seen by users currently attached to the database. Updates are seen when new users attach to the database after the change is completed.
Checksum calculation	No	
Reserve row cache slots	No	This change is not journaled.
Row cache enabled or disabled	No	This change is not journaled.
Create or add row cache	Yes	
Alter row cache	No	
Delete row cache	No	
Row cache attributes	No	
Snapshot files enabled or disabled	No	
Snapshot files immediate or deferred	No	
Snapshot checksum calculation	No	
Reserve journal	No	This change is not journaled.
Journaling enabled or disabled	No	

[1] *On Line* means other users can attach to the database while the metadata update is being performed. Other restrictions, as described in the Comments column of this table, may apply.

(continued on next page)

Table 7–3 (Cont.) Updating to Database-Wide Parameters While Users Are Attached to the Database

Metadata Update	On Line [1]	Comments
Root File Parameters		
Add journal	Yes	Online changes are allowed if the database root file contains available slots; that is, slots that have been reserved for journal files but not used.
Alter journal	Yes	
Delete journal	Yes	You cannot delete a journal file while it is in use.
Journal name or file name	No	
Journal allocation	Yes	
Journal backup server	Yes	
Journal backup file name	Yes	
Journal backup file name edit string	Yes	
Journal cache file name	Yes	
Journal extent	Yes	
Journal fast commit	No	
Journal checkpoint interval	No	
Journal checkpoint time	No	
Journal commit to journal optimization	No	
Journal transaction interval	No	
Journal log server	Yes	
Journal notify	Yes	
Journal overwrite	Yes	
Journal shutdown time	Yes	

[1] *On Line* means other users can attach to the database while the metadata update is being performed. Other restrictions, as described in the Comments column of this table, may apply.

(continued on next page)

Table 7–3 (Cont.) Updating to Database-Wide Parameters While Users Are Attached to the Database

Metadata Update	On Line [1]	Comments
Storage Area Parameters		
Reserve storage area	No	This change is not journaled.
Specify default storage area	Yes	
Read or write attribute	Yes	Requires exclusive access to the area.
Journaling enabled or disabled for write-once areas	No	
Allocation	Yes	
Extension enabled or disabled	Yes	Updates are not seen by users currently attached to the database. Updates are seen when new users attach to the database after the change is completed.
Extension options	Yes	
Lock-level options	No	
Thresholds	Yes	Requires exclusive access to the area.
Snapshot file allocation	Yes	Truncating snapshot file blocks read-only transactions.
Snapshot checksum allocation	No	
Snapshot file extension options	Yes	
SPAMs enabled or disabled	Yes	Requires exclusive access to the area. Use the RMU qualifiers Spams or Nospams.
Checksum calculation	No	
Security Parameters		
Audit file name	Yes	Use the RMU Set Audit command.
Alarm name	Yes	Use the RMU Set Audit command.
Audit enabled or disabled	Yes	Use the RMU Set Audit command.
Alarm enabled or disabled	Yes	Use the RMU Set Audit command.

[1] *On Line* means other users can attach to the database while the metadata update is being performed. Other restrictions, as described in the Comments column of this table, may apply.

(continued on next page)

Table 7–3 (Cont.) Updating to Database-Wide Parameters While Users Are Attached to the Database

Metadata Update	On Line [1]	Comments
Security Parameters		
Audit FIRST flag	Yes	Use the RMU Set Audit command.
Audit FLUSH flag	Yes	Use the RMU Set Audit command.
Audit event class flags	Yes	Use the RMU Set Audit command.

[1] *On Line* means other users can attach to the database while the metadata update is being performed. Other restrictions, as described in the Comments column of this table, may apply.

If the database was created or altered to include the DICTIONARY IS REQUIRED clause, you must specify path name access to execute most statements in data definition language.

Most metadata updates are journaled in both .ruj files and .aij files and execute in a read/write transaction. Storage area and database updates complete with an implicit commit operation. All other metadata updates complete with an explicit COMMIT or ROLLBACK statement.

7.3 Freezing Data Definition Changes

After you create all the elements for your database and you are ready to place the database in production, you can ensure that data definitions do not change. To do so, use the METADATA CHANGES ARE DISABLED clause of the ALTER or CREATE DATABASE or IMPORT statement.

When you use the METADATA CHANGES ARE DISABLED clause, Oracle Rdb does not allow data definition statements, except the ALTER DATABASE statement. (ALTER DATABASE is needed for database tuning.) Data definition statements include CREATE, ALTER, and DROP, as well as TRUNCATE TABLE, GRANT, and REVOKE statements.

Example 7–1 shows how to freeze metadata changes.

Example 7–1 Disallowing Data Definition Changes

```
SQL> ALTER DATABASE FILENAME mf_personnel
cont>      METADATA CHANGES ARE DISABLED;
```

The setting takes affect on the next attach to the database, as shown in the following example:

```
SQL> ATTACH 'FILENAME mf_personnel';
SQL> CREATE DOMAIN EMAIL CHAR (30);
%RDB-E-NO_META_UPDATE, metadata update failed
-RDMS-E-NOMETADATA, metadata operations are disabled
```

If you use the METADATA CHANGES ARE DISABLED clause in a CREATE DATABASE statement, Oracle Rdb implicitly enables SYSTEM INDEX COMPRESSION. System index compression is most useful when there are no metadata changes taking place.

To allow metadata changes, use the METADATA CHANGES ARE ENABLED clause.

7.4 Modifying Database Characteristics

You can select database characteristics explicitly or by default when you create or modify your database. You can modify many database characteristics using the ALTER DATABASE statement. However, for single-file databases, you cannot use an ALTER DATABASE statement to modify the following characteristics:

- Number of users
- Number of cluster nodes

To modify these characteristics, you must use the EXPORT and IMPORT statements to specify new values for these parameters.

Before you make modifications to your database, you should enable after-image journaling. After-image journaling stores all database activity in a common file. It provides a method to roll forward all transactions since the last backup operations. Although its use is optional, Oracle Rdb recommends that you enable after-image journaling to record your database transaction activity between full backup operations as part of your database restore and recovery strategies. For information about enabling after-image journaling, see Section 7.4.1.

Note that Oracle Rdb does not journal the following modifications to the database:

- Modifying the number of users
- Modifying the number of nodes
- Reserving slots for storage areas
- Reserving slots for journal files

- Reserving row cache
- Enabling or disabling row cache

Note

If you plan to modify database parameters that are not journaled, Oracle Rdb recommends that you back up your database before attempting these modifications because if the modification fails you can corrupt your database. If you have a backup copy of the database, you can restore the database.

After you have made these modifications, back up your database again in case you later have to restore your database from the backup.

To determine the present settings of the database characteristics, use the SHOW DATABASE and SHOW STORAGE AREAS statements. For example, to display the settings for the mf_personnel database, use the statements shown in Example 7–2.

Example 7–2 Displaying Settings for Database and Storage Area Parameters

```
SQL> ATTACH 'FILENAME mf_personnel';
SQL> SHOW DATABASE RDB$DBHANDLE
Default alias:
    Oracle Rdb database in file mf_personnel
        Multischema mode is disabled
        Number of users:              50
        Number of nodes:              16
        Buffer Size (blocks/buffer):   6
        Number of Buffers:            20
        Number of Recovery Buffers:   20
        Snapshots are Enabled Immediate
        Carry over locks are enabled
        Lock timeout interval is 0 seconds
        Adjustable lock granularity is enabled (count is 3)
        Global buffers are disabled (number is 250, user limit is 5)
                  page transfer via disk)
        Journal fast commit is disabled
                ( checkpoint interval is 0 blocks,
                  checkpoint timed every 0 seconds,
                  no commit to journal optimization,
                  transaction interval is 256 )
```

(continued on next page)

Example 7–2 (Cont.) Displaying Settings for Database and Storage Area Parameters

```
        AIJ File Allocation:          512
        AIJ File Extent:              512
        Statistics Collection is ENABLED
        Unused Storage Areas:          0
        Unused Journals:               1
        System Index Compression is DISABLED
        No Restricted Access
        Journal is Disabled
        Backup Server:   Manual
        Log Server:      Manual
        Overwrite:       Disabled
        Notification:    Disabled
        Asynchronous Prefetch is Enabled (depth is 5)
        Asynchronous Batch Write is Enabled (clean buffers 5, max buffers 4)
        Lock Partitioning is DISABLED
        Incremental Backup Scan Optim uses SPAM pages
        Shutdown Time is 60 minutes
        Unused Cache Slots:         1
        Workload Collection is Disabled
        Cardinality Collection is Enabled
        Metadata Changes are Enabled
        Row Cache is Disabled
        Detected Asynch Prefetch is Disabled
        Default Storage Area RDB$SYSTEM
        Mode is Open Automatic (Wait 0 minutes for close)
        Database Transaction Mode(s) Enabled:
            ALL
        Dictionary Not Required
        ACL based protections
Storage Areas in database with filename mf_personnel
     RDB$SYSTEM                     List storage area.
     EMPIDS_LOW
     EMPIDS_MID
     EMPIDS_OVER
     DEPARTMENTS
     SALARY_HISTORY
     JOBS
     EMP_INFO
     RESUME_LISTS
     RESUMES
Journals in database with filename mf_personnel
     No Journals Found
Cache Objects in database with filename mf_personnel
     No Caches Found
SQL> --
```

(continued on next page)

Example 7–2 (Cont.) Displaying Settings for Database and Storage Area Parameters

```
SQL> -- Display information about all the storage areas
SQL> SHOW STORAGE AREAS *
Storage Areas in database with filename mf_personnel

     RDB$SYSTEM
         List storage area.
         Access is:      Read write
         Page Format:    Uniform
         Page Size:      2 blocks
         Area File:      DBDISK:[PERS_DB]MF_PERS_DEFAULT.RDA;1
         Area Allocation:          861 pages
         Area Extent Minimum:      99 pages
         Area Extent Maximum:      9999 pages
         Area Extent Percent:      20 percent
         Snapshot File:  DBDISK:[PERS_DB]MF_PERS_DEFAULT.SNP;1
         Snapshot Allocation:      248 pages
         Snapshot Extent Minimum:  99 pages
         Snapshot Extent Maximum:  9999 pages
         Snapshot Extent Percent:  20 percent
         Extent :       Enabled
         Locking is Row Level
         No Cache Associated with Storage Area

Database objects using Storage Area RDB$SYSTEM:
Usage            Object Name                     Map / Partition
---------------- -----------------------------------------------------------
Default List Area
Storage Map      CANDIDATES                      CANDIDATES_MAP (1)
List Storage Map                                 LISTS_MAP (2)
System Table Area
Index            COLL_COLLEGE_CODE               (no map)
Index            DEG_COLLEGE_CODE                (no map)
Index            DEG_EMP_ID                      (no map)
Index            EMP_EMPLOYEE_ID                 (no map)
Index            EMP_LAST_NAME                   (no map)
Index            JH_EMPLOYEE_ID                  (no map)
Index            SH_EMPLOYEE_ID                  (no map)

     EMPIDS_LOW
         Access is:      Read write
         Page Format:    Mixed
         Page Size:      2 blocks
         Area File:      DBDISK:[PERS_DB]EMPIDS_LOW.RDA;1
         Area Allocation:          51 pages
         Area Extent Minimum:      99 pages
         Area Extent Maximum:      9999 pages
         Area Extent Percent:      20 percent
         Snapshot File:  DBDISK:[PERS_DB]EMPIDS_LOW.SNP;1
         Snapshot Allocation:      10 pages
         Snapshot Extent Minimum:  99 pages
         Snapshot Extent Maximum:  9999 pages
```

(continued on next page)

Example 7–2 (Cont.) Displaying Settings for Database and Storage Area Parameters

```
        Snapshot Extent Percent:  20 percent
        Extent :      Enabled
        Locking is Row Level
        No Cache Associated with Storage Area

Database objects using Storage Area EMPIDS_LOW:
Usage            Object Name                     Map / Partition
---------------- -------------------------------------------------------------
Index            EMPLOYEES_HASH                  (1)
Storage Map      EMPLOYEES                       EMPLOYEES_MAP (1)
Index            JOB_HISTORY_HASH                (1)
Storage Map      JOB_HISTORY                     JOB_HISTORY_MAP (1)
  .
  .
  .
SQL>
```

You can also use the RMU Dump Header command to display the database settings.

7.4.1 Enabling After-Image Journaling

Oracle Rdb records the actions of successfully completed transactions in after-image journal files (file type .aij). If a media failure or system error results in a corrupt database, you can use after-image journal files to reconstruct a database, or roll forward, from the last database backup. When you create a database, after-image journaling is disabled by default.

When you enable after-image journaling, you must create at least one journal file. Because enabling journaling is an offline operation, to reduce the amount of time the database is unavailable, create only one journal file when you enable journaling. You can create more journal files later, as discussed in Section 7.4.2.

You enable journaling using the JOURNAL IS ENABLED clause of the ALTER DATABASE statement. Example 7–3 shows how you enable after-image journaling, specify an allocation size, and create one journal file.

Example 7–3 Enabling After-Image Journaling

```
SQL> ALTER DATABASE FILENAME mf_personnel_test
cont>      JOURNAL IS ENABLED
cont>      (ALLOCATION IS 1024 BLOCKS)
cont>      ADD JOURNAL JOURN_1 FILENAME USER1:[DB.JOURNAL]JOURN_1.AIJ;
%RDMS-W-DOFULLBCK, full database backup should be done to ensure future recovery
SQL> --
SQL> ATTACH 'FILENAME mf_personnel_test';
SQL> SHOW DATABASE RDB$DBHANDLE
Default alias:
    Oracle Rdb database in file mf_personnel_test
   .
   .
   .
        AIJ File Allocation:          1024
        AIJ File Extent:              512
        Statistics Collection is ENABLED
        Unused Storage Areas:         15
        Unused Journals:               5
        No Restricted Access
        Journal is Enabled
        Journal File:    USER1:[DB.JOURNAL]JOURN_1.AIJ
   .
   .
   .
SQL>
```

Because you created only one journal file, Oracle Rdb creates an extensible file with a default extent size of 512 blocks. If you want multiple fixed-size journal files, create additional journal files, as described in Section 7.4.2.

Before you set the allocation value, you should estimate the amount of data that will be written to the .aij file in one day. Use the Performance Monitor to estimate blocks per transactions (BPT) and transactions per second (TPS). The following formula explains how to calculate the amount of data for one 8-hour day:

```
Amount of Data  =  8 hours x 3600 seconds per hour x BPT X TPS
```

You should set the allocation value to about 25 percent higher than one day's worth of data. For example, if the amount of data written to the .aij file in one day is 80,000 blocks, you should set the allocation to 100,000 blocks.

If you use a single extensible .aij file, you should set the extent value to about 10 percent of the allocation value. For example, if the allocation value is 100,00 blocks, you should set the extent to 10,000 blocks.

7.4.2 Adding After-Image Journal Files

As Section 3.5.4 explains, you can create multiple fixed-size .aij files or one extensible .aij file. If you already have one extensible .aij file, Oracle Rdb converts the existing .aij file to fixed-size when you add at least one more .aij file. Example 7–4 shows how to add two fixed-size .aij files.

Example 7–4 Adding Journal Files

```
SQL> -- Create 2 more journal files.
SQL> --
SQL> ALTER DATABASE FILENAME mf_personnel_test
cont>      ADD JOURNAL JOURN_2 FILENAME USER2:[DB.JOURNAL]JOURN_2.AIJ
cont>      ADD JOURNAL JOURN_3 FILENAME USER3:[DB.JOURNAL]JOURN_3.aij;
SQL> --
SQL> -- Display information about the .aij files.
SQL> ATTACH 'FILENAME mf_personnel_test';
SQL> SHOW JOURNALS *
Journals in database with filename mf_personnel_test
     JOURN_1
        AIJ File Allocation:           1024
        Journal File:    USER1:[DB.JOURNAL]JOURN_1.AIJ;1
        Edit String:     ()
     JOURN_2
        Journal File:    USER2:[DB.JOURNAL]JOURN_2.AIJ;1
        Edit String:     ()
     JOURN_3
        Journal File:    USER3:[DB.JOURNAL]JOURN_3.AIJ;1
        Edit String:     ()
SQL>
```

If you do not specify an allocation for the new .aij files, Oracle Rdb uses the database default values that you specified when you enabled journaling.

You should place each .aij file on a separate device and no .aij file should be on the same device as other database files so that you can more easily recover from failure of the hardware or software.

If you need more slots for .aij files, use the RESERVE JOURNALS clause of the ALTER DATABASE statement. When you reserve more slots, Oracle Rdb adds the number of slots in the RESERVE JOURNALS clause to the number of existing journal slots.

7.4.3 Modifying Allocation Characteristics for After-Image Journal Files

You can use the journal ALLOCATION IS clause to modify the allocation space for the .aij files. However, Oracle Rdb does not modify the allocation value for existing journal files. To modify the allocation value for existing journal files, you must drop the journal files and then add them again.

Example 7–5 sets the .aij file allocation size to 2048 blocks.

Example 7–5 Modifying the Allocation Value for .aij Files

```
SQL> -- Disable journaling before you drop existing journal files.
SQL> ALTER DATABASE FILENAME mf_personnel_test
cont>      JOURNAL IS DISABLED;
SQL> --
SQL> -- Drop the journal files.
SQL> ALTER DATABASE FILENAME mf_personnel_test
cont>      DROP JOURNAL JOURN_1
cont>      DROP JOURNAL JOURN_2
cont>      DROP JOURNAL JOURN_3;
SQL> --
SQL> --  Enable journaling and modify the allocation value.
SQL> ALTER DATABASE FILENAME mf_personnel_test
cont>      JOURNAL IS ENABLED
cont>             (ALLOCATION IS 2048 BLOCKS)
cont>      ADD JOURNAL JOURN_1 FILENAME USER1:[DB.JOURNAL]JOURN_1.AIJ
cont>      ADD JOURNAL JOURN_2 FILENAME USER2:[DB.JOURNAL]JOURN_2.AIJ
cont>      ADD JOURNAL JOURN_3 FILENAME USER3:[DB.JOURNAL]JOURN_3.AIJ;
%RDMS-W-DOFULLBCK, full database backup should be done to ensure future
recovery
```

For detailed information on after-image journaling and journal file attributes, see the *Oracle Rdb7 Guide to Database Maintenance* and the ALTER DATABASE section of the *Oracle Rdb7 SQL Reference Manual*.

7.4.4 Modifying the JOURNAL FAST COMMIT Options

You can enable or disable the journal FAST COMMIT option and set the checkpoint interval, the commit to journal optimization, and the transaction interval with the ALTER DATABASE statement. These options improve the performance of flushing data pages to disk at commit time for each user process based on the specified checkpoint time or interval and for trading off this improved performance with recovery performance.

The commit to journal optimization reduces the I/O to the database root file that is based on the specified transaction TSN interval. To take advantage of the commit to journal optimization, you must disable or defer snapshot files, so that a transaction is committed only by writing to the .aij file. You must enable

after-image journaling to use the journal fast commit and commit to journal performance optimizations.

Example 7–6 shows how to modify the journal fast commit attributes.

Example 7–6 Modifying the JOURNAL FAST COMMIT Attribute

```
SQL> ALTER DATABASE FILENAME mf_personnel_test
cont>      JOURNAL IS ENABLED
cont>         (FAST COMMIT IS ENABLED
cont>          (CHECKPOINT INTERVAL IS 100 BLOCKS,
cont>           COMMIT TO JOURNAL OPTIMIZATION,
cont>           TRANSACTION INTERVAL IS 20));
```

See the *Oracle Rdb7 SQL Reference Manual* for more information on the syntax and descriptions of these options. See the *Oracle Rdb7 Guide to Database Performance and Tuning* for a description of how these options work and information about the recommended values.

7.4.5 Modifying Extent Values for the Database

You can enable and disable database extents using the EXTENT ENABLED or DISABLED clause of the ALTER DATABASE statement. You can control the file space extent of the .rdb file for single-file databases or new storage area files for multifile databases.

You can specify simple control over the file space extent of the .rdb file for single-file databases or new storage area files for multifile databases by specifying the number of pages of each extent in the EXTENT IS extent-pages PAGES clause. The default is 100 pages.

Suppose that you recently added larger disk drives to your system and you allocated additional space for your database from the default of 400 pages to 1000 pages. Example 7–7 shows how to to enable extents and how to double your extent space from 100 to 200 pages for the single-file .rdb file or new storage area files.

Example 7–7 Modifying Extent Values

```
SQL> ALTER DATABASE FILENAME mf_personnel
cont>      EXTENT IS ENABLED
cont>      EXTENT IS 200 PAGES;
```

Suppose instead, that you want to exercise a higher degree of control over your extent growth. To do this, you set a minimum and maximum number of pages for your extents as well as the percent growth of extents, as shown in

Example 7–8. This affects only the single-file .rdb file or storage area files that are subsequently added.

Example 7–8 Modifying Extent Options

```
SQL> ALTER DATABASE FILENAME mf_personnel
cont>      EXTENT IS (MINIMUM OF 350 PAGES,
cont>      MAXIMUM OF 1000 PAGES,
cont>      PERCENT GROWTH IS 30);
```

Example 7–8 sets a minimum extent size of 350 pages, a maximum of 1000 pages, and a percent growth of 30. With growth of 30 percent, the first extent is 30 percent of 1000 pages (the new value for allocated space for your database), resulting in an extent size of 300 pages. However, because you specified a minimum of 350 pages, the extent is actually 350 pages. The extent is the size calculated by the percent growth factor when it falls within the range specified by the minimum and maximum parameters. If it falls outside this range, the minimum or maximum parameters become the extent size.

To control the file space extent of existing storage areas, including the RDB$SYSTEM storage area, for multifile databases, you use the EXTENT ENABLED or DISABLED clause of the ALTER STORAGE AREA clause. For information about enabling and disabling extent values for existing storage areas, see Section 7.6.

7.4.6 Modifying the Maximum Number of Users

To modify the maximum number of users for multifile databases, use the NUMBER OF USERS IS clause in the ALTER DATABASE statement. (For single-file databases, make this modification by first exporting the database using the EXPORT statement and then specifying the modification in the IMPORT statement.) For example, to modify the maximum number of users from 50 (the default) to 75, use the statement shown in Example 7–9.

Example 7–9 Modifying the Maximum Number of Database Users

```
SQL> ALTER DATABASE FILENAME mf_personnel
cont>      NUMBER OF USERS IS 75;
```

Because each attach to the database is equal to one user, an application that attaches to the same database twice is treated as two users.

The ALTER DATABASE statement extends the .rdb file to include 15 additional run-time user process blocks (RTUPBs), for a total of 75. However, if you reduce the maximum number of database users, the .rdb file does not shrink, although the extra RTUPBs are deleted.

7.4.7 Modifying the Maximum Number of Cluster Nodes

OpenVMS VAX OpenVMS Alpha

For multifile databases, you can use the ALTER DATABASE statement to specify the maximum number of nodes that can access your database within a cluster environment. (For single-file databases, make this change by first exporting the database using the EXPORT statement and then specifying the change in the IMPORT statement.)

Use the NUMBER OF CLUSTER NODES IS clause in the ALTER DATABASE statement to limit the allowable number of nodes for multifile databases. The default is 16 nodes. Example 7–10 shows how to increase the number of cluster nodes to 28 nodes.

Example 7–10 Modifying the Maximum Number of Cluster Nodes

```
SQL> ALTER DATABASE FILENAME mf_personnel
cont>      NUMBER OF CLUSTER NODES IS 28;
```

For small systems, modifying the number of cluster nodes to the desired number can save some disk space. You save 1 block of disk space per cluster node below the default of 16. If you have 2 cluster nodes, specifying 2 nodes saves you 14 blocks of disk space.

See the *Oracle Rdb7 Guide to Database Performance and Tuning* for more information on using Oracle Rdb in a cluster environment. ♦

7.4.8 Modifying Database Lock Characteristics

You can modify database lock characteristics with the following clauses of the ALTER DATABASE statement:

- ADJUSTABLE LOCK GRANULARITY IS
- CARRY OVER LOCKS ARE
- LOCK TIMEOUT INTERVAL IS

The adjustable lock granularity capability is enabled by default. This optimization enables Oracle Rdb to request as few locks as possible for database row locking while still ensuring maximum concurrent access to pages within the logical area. Depending on the level of contention for the rows, Oracle Rdb adjusts the level of lock granularity downward until it reaches the single-row level.

The ADJUSTABLE LOCK GRANULARITY IS ENABLED option helps minimize the number of locks needed by using a default value of 3 for the number of page range levels and a fanout factor value of 10 at each level. These intermediate page ranges can be viewed as a three-tier hierarchical level that describes the number of levels in the record lock tree. Thus, 10 pages of records will be in the first (or lowest) level, 100 pages in the second level, and 1000 pages in the third (or highest) level.

All transactions initially request a strong lock at the highest level. If there is no contention, a transaction needs only one lock for all the rows it needs to access. If another transaction needs to access rows from the same area, Oracle Rdb adjusts the locks downward. When contention for pages in the database is very high, adjustable lock granularity can use up CPU time as Oracle Rdb adjusts the lock granularity downward. See the *Oracle Rdb7 Guide to Database Performance and Tuning* for an illustration of adjustable locking levels.

You can control the lock granularity further by specifying the number of page levels by using the COUNT clause. You can specify from 1 to 8 page levels.

If your database has high page contention (many users accessing the same area simultaneously), consider specifying a lower COUNT value. If your database has few users who perform queries that access many widely dispersed rows, specifying a higher COUNT value may result in less locking.

If you specify ADJUSTABLE LOCK GRANULARITY IS DISABLED, Oracle Rdb requests a lock for each database row requested. Oracle Rdb recommends that you start with adjustable lock granularity enabled, using the default COUNT value and then, if CPU time is a problem, adjust the count or disable the adjustable lock granularity and determine if this improves your performance. At the application level, enabling adjustable lock granularity is good for heavy retrieval content where the application processes groups of rows, for example:

```
SQL> SELECT * FROM CURRENT_JOB
cont>     WHERE DEPARTMENT_CODE = 'ELEL';
```

On the other hand, disabling adjustable lock granularity is good for heavy update content when the application processes very specific rows, for example:

```
SQL> SELECT * FROM EMPLOYEES
cont>     WHERE EMPLOYEE_ID = '00164';
```

You can use the Performance Monitor to determine the number of locks used by your database. Page contention that requires changes in locks is indicated by the number of blocking asynchronous system traps (AST) messages. If this number is greater than 20 percent to 25 percent of the number of locks requested, indicating frequent page contention, consider modifying the ADJUSTABLE LOCK GRANULARITY clause. Ideally, you want to reach a balance between the frequency of page contention and the number of locks required.

For example, suppose you evaluated the results of a series of typical transactions with the Performance Monitor command and determined that contention for pages was the problem because the number of blocking AST messages was 35 percent of the number of locks requested. Then, you disable lock granularity as shown in Example 7–11.

Example 7–11 Modifying Adjustable Lock Granularity

```
SQL> ALTER DATABASE FILENAME mf_personnel
cont>      ADJUSTABLE LOCK GRANULARITY IS DISABLED;
```

The CARRY OVER LOCKS clause enables or disables the carry-over lock optimization. By default, carry-over locks are enabled and row and area locks are held across transactions. Because the NOWAIT lock is required when carry-over locks are enabled, your transaction may experience a delay in trying to acquire the NOWAIT lock when a process executes a long statement. To improve lock performance under these circumstances, you can disable carry-over locks as shown in Example 7–12.

Example 7–12 Disabling Carry-Over Locks

```
SQL> ALTER DATABASE FILENAME mf_personnel
cont>      CARRY OVER LOCKS ARE DISABLED;
```

The LOCK TIMEOUT INTERVAL IS clause specifies the number of seconds for processes to wait before timing out during a lock conflict. Oracle Rdb uses the value specified database-wide as the upper limit in determining the timeout interval when compared to a higher value specified in either a SET TRANSACTION WAIT statement or the RDM$BIND_LOCK_TIMEOUT logical name or RDB_BIND_LOCK_TIMEOUT configuration parameter. Example 7–13 sets the value for the lock timeout interval to ten seconds.

Example 7–13 Setting the Lock Timeout Interval

```
SQL> ALTER DATABASE FILENAME mf_personnel
cont>      LOCK TIMEOUT INTERVAL IS 10 SECONDS;
```

For more information on database lock characteristics, see the *Oracle Rdb7 Guide to Database Performance and Tuning*.

7.4.9 Selecting Locking Levels for Storage Areas

Oracle Rdb uses two types of locks for storage areas: row-level locks to provide logical consistency and page-level locks to provide mutual exclusion. The row-level locking and page-level locking mechanisms operate independently of each other.

You can use the following clauses of the ALTER DATABASE statement or ALTER STORAGE AREA clause to alter the locking level for one or more storage areas or for all storage areas:

- LOCKING IS ROW LEVEL

  Specifies that a transaction accessing the storage area uses both row-level and page-level locks. The LOCKING IS ROW LEVEL option is appropriate for most transactions, especially those that lock many rows in one or more storage areas and that are long in duration.

- LOCKING IS PAGE LEVEL

  Specifies that *only* page-level locking be used. When you enable page-level locking for a storage area, transactions accessing the storage area hold only page-level locks and do not request any row-level locks. The page-level locks provide both logical consistency and mutual exclusion.

  Page-level locking is most likely to be beneficial for a partitioned application, in which individual application processes do not access the same data pages at the same time.

Use the LOCKING IS PAGE LEVEL clause carefully, only when you have considered the implications of page-level locking. For more information on when to use the LOCKING IS PAGE LEVEL clause, see the *Oracle Rdb7 Guide to Database Performance and Tuning*.

7.4.10 Enabling or Disabling Global Buffers

You can enable or disable the use of global buffers and set global buffer options such as the total number of global buffers per node and the maximum number of global buffers each user allocates in the ALTER DATABASE statement. Enabling global buffers improves performance because it reduces I/O operations and better utilizes memory. To enable global buffers, enter the statement shown in Example 7–14.

Example 7–14 Enabling Global Buffers by Node

```
SQL> ALTER DATABASE FILENAME mf_personnel
cont>      GLOBAL BUFFERS ARE ENABLED
cont>      (NUMBER IS 60,
cont>       USER LIMIT IS 5);
```

See the *Oracle Rdb7 SQL Reference Manual* for syntax and a description of global buffers and options. See the *Oracle Rdb7 Guide to Database Performance and Tuning* for a discussion about using global buffers and recommendations for setting global buffer options.

7.4.11 Modifying the Buffer Size

You can use the BUFFER SIZE clause of the ALTER DATABASE statement to modify the number of blocks Oracle Rdb allocates for each buffer.

Although the range of values is from 1 to 64, you can specify only a number that accommodates all the page sizes used in the database. For example, if a database contains a page size of 5, you cannot specify a buffer size of 4.

For information on how to determine the buffer size, see the *Oracle Rdb7 Guide to Database Performance and Tuning*.

7.4.12 Modifying the Number of Local Database Buffers

The NUMBER OF BUFFERS clause of the ALTER DATABASE statement lets you specify the number of local database buffers for all users of a multifile database. (To modify the number of buffers for a single-file database, use the EXPORT and IMPORT statements.)

This clause specifies the number of local buffers allocated to a user who attaches to the database. The default is 20 buffers. Example 7–15 shows how to increase the number of local database buffers by 50 percent.

Example 7–15 Modifying the Number of Buffers

```
SQL> ALTER DATABASE FILENAME mf_personnel
cont>      NUMBER OF BUFFERS IS 30;
```

See the *Oracle Rdb7 Guide to Database Performance and Tuning* for information on using the logical name RDM$BIND_BUFFERS or the configuration parameter RDB_BIND_BUFFERS. They allow you to provide an alternative number of buffers at run time. This can be useful when you need to temporarily override the default number of local buffers for a specific task, but in general want to use the default.

7.4.13 Modifying the Number of Database Recovery Buffers

The NUMBER OF RECOVERY BUFFERS IS clause of the ALTER DATABASE statement specifies the number of recovery buffer for all users. This clause specifies the number of buffers to be used by the database recovery (DBR) process when it removes uncommitted changes in the .ruj file. The larger the number of recovery buffers, the faster the recovery process runs. Use as many recovery buffers as fit within the working set of a recovery process. The default is 40 recovery buffers.

Example 7–16 shows how to increase the number of recovery buffers to 50.

Example 7–16 Modifying the Number of Recovery Buffers

```
SQL> ALTER DATABASE FILENAME mf_personnel
cont>      NUMBER OF RECOVERY BUFFERS IS 50;
```

7.4.14 Controlling Snapshot Files

You can use the SNAPSHOT IS ENABLED or SNAPSHOT IS DISABLED clause of the ALTER DATABASE statement to enable or disable snapshot files. You can use the SNAPSHOT ALLOCATION IS option to modify the space allocation, and SNAPSHOT EXTENT IS option to modify extent values of snapshot files.

By default, snapshot files are allowed and not deferred when you create the database. You must use either deferred snapshot files or disabled snapshot files to take advantage of the commit-to-journal optimization described in Section 7.4.4 as part of the journal fast commit option. The default snapshot allocation is 100 pages.

You can set the snapshot allocation to a small value such as three pages. Setting the snapshot allocation to a small value is useful in two cases:

- If you have disabled snapshot files and you want to save some space.

- If the storage area is read-only or write-once, the snapshot file is little used, and you want to save some additional space.

To determine the current snapshot settings for a database, use the SHOW DATABASE and SHOW STORAGE AREAS statements, as shown in Example 7–17.

Example 7–17 Displaying Current Snapshot File Settings

```
SQL> SHOW DATABASE RDB$DBHANDLE
Default alias:
    Oracle Rdb database in file mf_personnel
        Multischema mode is disabled
        Number of users:              50
        Number of nodes:              16
        Buffer Size (blocks/buffer):   6
        Number of Buffers:            20
        Number of Recovery Buffers:   20
        Snapshots are Enabled Immediate
   .
   .
   .
SQL> SHOW STORAGE AREAS *
Storage Areas in database with filename mf_personnel

     RDB$SYSTEM
         List storage area.
         Access is:      Read write
         Page Format:    Uniform
         Page Size:      2 blocks
         Area File:      DBDISK:[PERS_DB]MF_PERS_DEFAULT.RDA;1
         Area Allocation:          861 pages
         Area Extent Minimum:      350 pages
         Area Extent Maximum:      1000 pages
         Area Extent Percent:      30 percent
         Snapshot File:  DBDISK:[PERS_DB]MF_PERS_DEFAULT.SNP;1
         Snapshot Allocation:      248 pages
         Snapshot Extent Minimum:  99 pages
         Snapshot Extent Maximum:  9999 pages
         Snapshot Extent Percent:  20 percent
         Extent :       Enabled
         Locking is Row Level
   .
   .
   .
SQL>
```

Oracle Rdb maintains snapshot file information for each storage area, except when the snapshot files are deferred. When you enable or disable snapshot files, you do it for all storage areas; you cannot enable or disable snapshot files for individual storage areas. For example, to disable snapshot files for the mf_personnel database including all storage areas, use the statement shown in Example 7–18.

Example 7–18 Disabling Snapshot Files

```
SQL> ALTER DATABASE FILENAME mf_personnel
cont>      SNAPSHOT IS DISABLED;
```

Enabling snapshot files lets read-only users access snapshot files and avoid row-locking conflicts with users who are updating the database. When snapshot files are enabled and not deferred, update transactions must write before-images to the snapshot file of each row they update, whether or not there are any active read-only users currently accessing a snapshot file. Read-only transactions use snapshot files to access rows, thereby seeing a consistent view of the database.

If snapshot files are not enabled when you attempt to start a transaction in read-only mode, Oracle Rdb starts a read/write transaction. Any read transactions lock rows and can reduce database performance for read-intensive applications.

Enabling snapshot files can reduce database performance for update-intensive applications. With snapshot files enabled, you create additional I/O operations for update transactions (except for exclusive write and batch-update transactions, neither of which writes before-images to the snapshot file). However, for read-intensive applications, using snapshot files results in good performance.

You may improve performance by scheduling times when you allow read-only access. You can enable snapshot files when update activity is low, allow read-only access for a certain amount of time, and then disable snapshot files again until the next scheduled read-only access time. Using the SNAPSHOT IS ENABLED DEFERRED option saves overhead and improves performance by writing to the snapshot file only when a read-only job is in progress. See Section 7.4.15 for more information on deferred snapshot files.

To find out if any snapshot file transactions (read-only) are in progress, use the RMU Dump Users command as shown in Example 7–19.

Example 7–19 Determining If Snapshot File Transactions Are in Progress

```
$ RMU/DUMP/USERS mf_personnel

Active user with process ID 2080DC8E
    Stream ID is 1
    Monitor ID is 1
    Transaction ID is 17
    Snapshot transaction in progress
    Last Process quiet-point was AIJ sequence 0
    Transaction sequence number is 0:736
```

Example 7–19 shows a user accessing the mf_personnel database through a read-only transaction with a transaction sequence number (TSN) of 0:736. Oracle Rdb uses the TSN to reclaim space in snapshot files by writing over rows bearing a lower TSN than any active transaction.

7.4.15 Using Deferred Snapshot Files

To improve update performance, especially when improved update performance is more important than the immediate execution of a read-only transaction, specify deferred snapshot files. Using the DEFERRED option saves overhead by writing to the snapshot file *only* when a read-only transaction is in progress. Also, you must use either deferred snapshots or disable snapshots to take advantage of the commit to journal optimization described in Section 7.4.4 as part of the journal fast commit option.

To use deferred snapshot files, specify the SNAPSHOTS IS ENABLED DEFERRED clause of the ALTER DATABASE statement. This option tells Oracle Rdb to defer snapshot files throughout the database.

When you use deferred snapshot files, read-only transactions wait for all update transactions to begin writing to the snapshot files before starting. Subsequent read-only transactions do not wait. In this way, the overhead of writing to a snapshot file is used only when it is really needed. After a read-only transaction starts, all subsequent update transactions write before-images of the rows they modify to the snapshot files. When the read-only transaction completes, update transactions cease writing to the snapshot files.

For example, to specify deferred snapshot files for the mf_personnel database, use the statement shown in Example 7–20.

Example 7–20 Specifying Deferred Snapshot Files

```
SQL> ALTER DATABASE FILENAME mf_personnel
cont>      SNAPSHOT IS ENABLED DEFERRED;
```

The default specification for snapshot files is IMMEDIATE. If you accept the default, an update transaction that uses a shared (concurrent) or protected share mode always writes to the snapshot file, even when no read-only transaction is in progress.

For more information on the performance gains of using the DEFERRED clause for snapshot files, see the *Oracle Rdb7 Guide to Database Performance and Tuning*.

7.4.16 Modifying Extent Characteristics for Snapshot Files

You can use the SNAPSHOT EXTENT IS clause to specify the number of pages to be added to the .snp file when the allocation is exceeded. You can also use the extension-options clause of the ALTER DATABASE statement to specify the minimum and the maximum number of pages for the extent and the percent growth of the extent rather than using the SNAPSHOT EXTENT IS option. The statement in Example 7–21 sets the extent size for all new storage area files. It specifies that a minimum of 10 pages and a maximum of 100 pages for the extent size and it expands by increments equal to 10 percent of its present size.

Example 7–21 Modifying Extent Characteristics for Snapshot Files

```
SQL> ALTER DATABASE FILENAME mf_personnel
cont>      SNAPSHOT EXTENT IS
cont>      (MINIMUM OF 10 PAGES,
cont>       MAXIMUM OF 100 PAGES,
cont>       PERCENT GROWTH IS 10);
```

7.4.17 Modifying the Allocation for Snapshot Files

If your snapshot files extend, you can reduce them by specifying an allocation value lower than the current size of the file. Display the database header to determine the current size of your snapshot file, as shown in Example 7–22.

Example 7–22 Determining the Current Size of a Snapshot File

```
$ RMU/DUMP/HEADER mf_personnel
   .
   .
   .
Snapshot area for storage area DEPARTMENTS
    Area ID number is 15
    Filename is DUA0:[ORION.RDO]DEPARTMENTS.SNP;1
    Pages...
      - Page size is 2 blocks
      - Initial data page count was 27
      - Current physical page count is 100    ❸
      - Page checksums are enabled
      - Row level locking is enabled
    Extension...
      - Extends are enabled                   ❶
      - Extend area by 20%, minimum of 99 pages, maximum of 9999 pages
      - Volume set spreading is enabled
      - Area has been extended 1 time         ❷
        .
        .
        .
```

The header in Example 7–22 shows the following for the snapshot file for the DEPARTMENTS area of the mf_personnel database:

❶ EXTENTS are enabled.

❷ The area has been extended one time.

❸ The area has been extended to 100 pages.

To globally reduce or expand the snapshot file for storage areas that are subsequently created, use the SNAPSHOT ALLOCATION IS clause of the ALTER DATABASE statement. For example, to increase the snapshot file from 102 to 200 pages for any new snapshot files, use the statement shown in Example 7–23.

Example 7–23 Increasing the Allocation Size of Snapshot Files

```
SQL> ALTER DATABASE FILENAME mf_personnel
cont>      SNAPSHOT ALLOCATION IS 200 PAGES;
```

For information about the content of data pages in snapshot files, see the *Oracle Rdb7 Guide to Database Maintenance*.

You can truncate the size of a snapshot file while the database is in use, although read-only transactions are blocked while the truncation is taking place.

See the *Oracle Rdb7 SQL Reference Manual* for a complete description of the extent options.

7.5 Modifying the Requirement for Using the Repository

You use the DICTIONARY IS REQUIRED or DICTIONARY IS NOT REQUIRED clause of the ALTER DATABASE statement to change the requirement that metadata be written to the repository. Example 7–24 shows how to make the repository not required.

Example 7–24 Using the DICTIONARY IS NOT REQUIRED Option

```
SQL> ALTER DATABASE FILENAME mf_personnel
SQL>       DICTIONARY IS NOT REQUIRED;
```

If you specify the DICTIONARY IS NOT REQUIRED option and make changes to your metadata, the database metadata and repository are no longer identical and the repository needs updating.

Use the DICTIONARY IS REQUIRED option to make sure that all updates to the metadata also occur in your repository. When you have made the necessary updates, you can disable requiring the repository.

If you invoke the database with the file name when the repository is required, you receive an error message the first time you try to update metadata.

To remove all links between a database and the repository, you can use the DICTIONARY IS NOT USED clause. See Section 10.12.1 for more information.

7.6 Modifying Storage Areas and Storage Area Parameters

This section describes how to modify storage areas and storage area parameters. Before reading this section, you should be familiar with information about creating the database and storage areas in Chapter 3 and Chapter 4.

Although you use the ALTER STORAGE AREA clause of the ALTER DATABASE statement to modify most attributes of storage areas, you must use the EXPORT and IMPORT statements to modify the following attributes:

- PAGE SIZE
- PAGE FORMAT

- THRESHOLDS
- SNAPSHOT FILENAME

Example 7–25 shows how to use the ALTER STORAGE AREA clause to change the extent and allocation of the EMPIDS_OVER storage area.

Example 7–25 Modifying Storage Areas

```
SQL> ALTER DATABASE FILENAME mf_personnel
cont>      ALTER STORAGE AREA EMPIDS_OVER
cont>      EXTENT IS 200 PAGES
cont>      ALLOCATION IS 100 PAGES;
```

Using the EXTENT and ALLOCATION IS clause of the ALTER STORAGE AREA clause, you can change the file space extent and allocation of existing storage areas, including the RDB$SYSTEM storage area.

See Chapter 4 for information on how to calculate the extent and allocation for storage areas.

The following sections describe:

- How to add new storage areas to a multifile database
- How to modify storage areas to cluster rows
- How to use the IMPORT and EXPORT statements to adjust storage area parameters, including the RDB$SYSTEM storage area
- How to move a storage area to a different disk device
- How to change a read/write storage area to a read-only storage area
- How to change a read/write storage area to a write-once storage area
- How to change a read/write storage area to a read-only storage area
- How to delete storage areas

For a more complete discussion of storage design and database characteristics, see Section 4.8 and the *Oracle Rdb7 Guide to Database Performance and Tuning*. See the *Oracle RMU Reference Manual* for reference information about the RMU Backup and RMU Restore commands.

7.6.1 Adding New Storage Areas for Multifile Databases

You can add new storage areas to a multifile database using the ADD STORAGE AREA clause of the ALTER DATABASE statement.

You must have unused storage areas slots reserved in the database root file; otherwise, Oracle Rdb returns an error. You can reserve additional storage area slots using the RESERVE STORAGE AREA clause of the ALTER DATABASE statement. When you reserve more slots, Oracle Rdb adds the number of slots in the RESERVE STORAGE AREA clause to the number of existing storage area slots.

You cannot use the RESERVE STORAGE AREA clause while other users are attached to the database. Also, note that because adding storage areas is an online operation and reserving storage areas is an offline operation, you cannot use the ADD STORAGE AREA clause and the RESERVE STORAGE AREA clause in the same ALTER DATABASE statement.

When you add storage areas, any storage area parameters that you do not set explicitly take the Oracle Rdb defaults, not the global definitions you may have specified in the CREATE DATABASE statement.

For example, if your database users want to keep track of job assignments, you create a new table called JOB_ASSIGNMENTS, and store its rows in a new storage area named JOB_ASSIGNMENTS. Next, you define the necessary options for this new storage area. Table 7–4 shows the columns and keys for the JOB_ASSIGNMENTS table.

Table 7–4 Columns and Keys for the JOB_ASSIGNMENTS Table

Column Name	Data Type	Key
EMPLOYEE_ID	CHAR (5)	Primary Key
JOB_CODE	CHAR (4)	Foreign Key
ASSIGNMENT_NAME	CHAR (8)	Foreign Key
ASSIGNMENT_DESCRIPTION	CHAR (30)	
START_DATE	DATE	
END_DATE	DATE	
ASSIGNMENT_STATUS	CHAR(1)	
SUPERVISOR_ID	CHAR (5)	Foreign key

Table 7–4 shows the columns in the new JOB_ASSIGNMENTS table and the three columns (EMPLOYEE_ID, JOB_CODE, and ASSIGNMENT_NAME) whose values uniquely identify each row. The ASSIGNMENT_NAME column

contains acronyms that describe each assignment. The ASSIGNMENT_DESCRIPTION column describes the assignment, and the ASSIGNMENT_STATUS column describes the status of the assignment.

A storage map, JOB_ASSIGNMENT_MAP, and a sorted index, JOB_ASSIGNMENTS_SORT, assist row storage and retrieval. (See Section 7.7.1 for the index definition and proposed changes and Section 7.9 for the storage map definition and proposed changes.)

Example 7–26 shows how to add a storage area for this table.

Example 7–26 Adding a Storage Area and Specifying Parameters

```
SQL> ALTER DATABASE FILENAME mf_personnel
cont> ADD STORAGE AREA JOB_ASSIGNMENTS FILENAME job_assignments.rda
cont>     ALLOCATION IS 1024 PAGES
cont>     PAGE SIZE IS 4 BLOCKS
cont>     PAGE FORMAT IS MIXED
cont>     THRESHOLDS ARE (70, 85, 95)
cont>     INTERVAL IS 256
cont>     EXTENT IS 512 PAGES
cont>     SNAPSHOT FILENAME job_assignments.snp
cont>     SNAPSHOT ALLOCATION IS 200
cont>     SNAPSHOT EXTENT IS 100;
```

The definition specifies the characteristics for the new storage area, JOB_ASSIGNMENTS. Because the page format is mixed, you can define a hashed index and locate it in this storage area. The page size is set to 4 blocks based on index record and data row sizes. For a description of how to estimate the sizes of hashed index structures and data rows, see Section 4.8.

The extent space for the storage area is 512 pages or half of the size of the space allocation, which provides for a small number of extents to be created if file allocation space is exceeded. Because both read and update transactions are used on the database, the snapshot space allocation and extent size should accommodate read/write as well as read-only transactions. See the *Oracle Rdb7 Guide to Database Performance and Tuning* for more information on the performance benefits of each parameter, based on the database application.

7.6.2 Adjusting Storage Area Parameters to Cluster Rows

You can design a database to cluster related rows in the same storage area. Assume that you have a database that clusters related rows from the EMPLOYEES and JOB_HISTORY tables but stores rows from the SALARY_HISTORY table in a different storage area. If you want to modify data storage to cluster SALARY_HISTORY rows with related rows from the EMPLOYEES and JOB_HISTORY tables, you take the following steps:

1. Recalculate values for ALLOCATION, PAGE SIZE, INTERVAL, and so forth, to be appropriate for the revised design.

   See Section 4.8 and the *Oracle Rdb7 Guide to Database Performance and Tuning* for more information about recalculating optimal values for these parameters.

2. Create new mixed page format storage areas according to your calculations.

3. Modify index STORE clauses and table storage maps to move hashed indexes and data from all three tables to the new storage areas.

   You cannot create a storage map for a table in which data is already stored, unless it is stored in the default or RDB$SYSTEM storage area. However, you can create new indexes and refer to these new indexes in ALTER STORAGE MAP statements. Example 7–27, therefore, only alters storage maps. In two cases, the storage maps refer to hashed indexes for which STORE clauses have been modified. In one case, the storage map refers to a newly created hashed index.

4. Delete the empty storage areas (if desired).

The mf_personnel database currently has EMPLOYEES and JOB_HISTORY data, placed using hashed indexes on EMPLOYEE_ID, in storage areas EMPIDS_LOW, EMPIDS_MID, EMPIDS_OVER. Data for the SALARY_HISTORY table is currently stored in the SALARY_HISTORY storage area. Example 7–27 shows how to change row and hashed index distribution for these three tables.

Example 7–27 Using ALTER DATABASE, ALTER INDEX, and ALTER STORAGE MAP Statements

```
SQL> -- This example adds three new storage areas to the database.  Then,
SQL> -- it alters and creates hashed indexes, and alters storage maps
SQL> -- to cluster related rows, EMPLOYEES, JOB_HISTORY, and SALARY_HISTORY,
SQL> -- in the new storage areas.
```

(continued on next page)

Example 7–27 (Cont.) Using ALTER DATABASE, ALTER INDEX, and ALTER STORAGE MAP Statements

```
SQL> --
SQL> ALTER DATABASE FILENAME mf_personnel
cont> --
cont> ADD STORAGE AREA PERSONNEL_4
cont>     FILENAME PERS4
cont>     PAGE FORMAT IS MIXED
cont>     ALLOCATION IS . . .
cont>     INTERVAL IS . . .
cont>     THRESHOLDS ARE . . .
cont>     PAGE SIZE IS . . .
cont> --
cont> ADD STORAGE AREA PERSONNEL_5
cont>     FILENAME PERS5
cont>     PAGE FORMAT IS MIXED
cont>     ALLOCATION IS . . .
cont>     INTERVAL IS . . .
cont>     THRESHOLDS ARE . . .
cont>     PAGE SIZE IS . . .
cont> --
cont> ADD STORAGE AREA PERSONNEL_6
cont>     FILENAME PERS6
cont>     PAGE FORMAT IS MIXED
cont>     ALLOCATION IS . . .
cont>     INTERVAL IS . . .
cont>     THRESHOLDS ARE . . .
cont>     PAGE SIZE IS . . .
cont> ;
SQL> --
SQL> -- After the ALTER DATABASE statement completes, you must
SQL> -- attach to the database before executing the following
SQL> -- ALTER and CREATE statements:
SQL> --
SQL> ATTACH 'FILENAME mf_personnel';
SQL> --
SQL> ALTER INDEX EMPLOYEES_HASH
cont>      STORE USING (EMPLOYEE_ID)
cont>            IN PERSONNEL_4 WITH LIMIT OF ('00200')
cont>            IN PERSONNEL_5 WITH LIMIT OF ('00399')
cont>      OTHERWISE IN PERSONNEL_6;
```

(continued on next page)

Example 7–27 (Cont.) Using ALTER DATABASE, ALTER INDEX, and ALTER STORAGE MAP Statements

```
SQL> --
SQL> ALTER STORAGE MAP EMPLOYEES_MAP
cont>      STORE USING (EMPLOYEE_ID)
cont>            IN PERSONNEL_4 WITH LIMIT OF ('00200')
cont>            IN PERSONNEL_5 WITH LIMIT OF ('00399')
cont>      OTHERWISE IN PERSONNEL_6
cont>      PLACEMENT VIA INDEX EMPLOYEES_HASH
cont>      REORGANIZE;
SQL> --
SQL> ALTER INDEX JOB_HISTORY_HASH
cont>      STORE USING (EMPLOYEE_ID)
cont>            IN PERSONNEL_4 WITH LIMIT OF ('00200')
cont>            IN PERSONNEL_5 WITH LIMIT OF ('00399')
cont>      OTHERWISE IN PERSONNEL_6;
SQL> --
SQL> ALTER STORAGE MAP JOB_HISTORY_MAP
cont>      STORE USING (EMPLOYEE_ID)
cont>            IN PERSONNEL_4 WITH LIMIT OF ('00200')
cont>            IN PERSONNEL_5 WITH LIMIT OF ('00399')
cont>      OTHERWISE IN PERSONNEL_6
cont>      PLACEMENT VIA INDEX JOB_HISTORY_HASH
cont>      REORGANIZE;
SQL> --
SQL> CREATE INDEX SH_ID_IND ON SALARY_HISTORY (EMPLOYEE_ID)
cont>       TYPE IS HASHED
cont>       STORE USING (EMPLOYEE_ID)
cont>             IN PERSONNEL_4 WITH LIMIT OF ('00200')
cont>             IN PERSONNEL_5 WITH LIMIT OF ('00399')
cont>       OTHERWISE IN PERSONNEL_6;
SQL> --
SQL> ALTER STORAGE MAP SALARY_HISTORY_MAP
cont>      STORE USING (EMPLOYEE_ID)
cont>            IN PERSONNEL_4 WITH LIMIT OF ('00200')
cont>            IN PERSONNEL_5 WITH LIMIT OF ('00399')
cont>      OTHERWISE IN PERSONNEL_6
cont>      PLACEMENT VIA INDEX SH_ID_IND
cont>      REORGANIZE;
SQL> --
SQL>  COMMIT;
```

Section 7.7 and Section 7.9 provide more information about modifying indexes and storage maps.

7.6.3 Adjusting the RDB$SYSTEM Storage Area

You can make adjustments to storage parameters of the RDB$SYSTEM storage area using the ALTER STORAGE AREA clause of the ALTER DATABASE statement or the EXPORT and IMPORT statements. You can use the ALTER STORAGE AREA clause to modify the following characteristics:

- Extent values
- Allocation
- Checksum values
- Row-cache name
- Snapshot allocation, extent, and checksum calculation

To modify other storage parameters of the RDB$SYSTEM storage area, as well as to modify row and hashed index distribution for that storage area, you must use the EXPORT and IMPORT statements rather than the ALTER DATABASE statement.

The IMPORT statement allows only CREATE and DROP clauses (no ALTER clauses). However, if a CREATE clause refers to an existing storage area, index, or storage map, SQL alters the existing storage area, index, or storage map to your specifications before reloading any data.

Example 7–28 shows how modify the page size of the RDB$SYSTEM storage area. It assumes that the database is imported to the same directory from which it was exported, but the existing files have been deleted from that directory.

Example 7–28 Using EXPORT and IMPORT Statements to Modify the RDB$SYSTEM Storage Area

```
SQL> -- The IMPORT statement changes the page size for the RDB$SYSTEM
SQL> -- area.
SQL>
SQL> EXPORT DATABASE FILENAME mf_personnel_mf INTO personnel_mf;
SQL> --
SQL> -- If you import the database into the same directories from
SQL> -- which you exported the database, delete existing database files
SQL> -- before entering the IMPORT statement.
SQL> --
SQL> DROP DATABASE FILENAME mf_personnel;
```

(continued on next page)

Example 7–28 (Cont.) Using EXPORT and IMPORT Statements to Modify the RDB$SYSTEM Storage Area

```
SQL> --
SQL> -- When the IMPORT statement executes, it displays informational
SQL> -- messages.
SQL> --
SQL> IMPORT DATABASE FROM personnel_mf FILENAME mf_personnel
cont>       PAGE SIZE IS 5 BLOCKS;           -- applies to RDB$SYSTEM area
Exported up by Oracle Rdb V7.0-0 Import/Export utility
A component of SQL V7.0-2
Previous name was mf_personnel
   .
   .
   .
IMPORTing STORAGE AREA: RDB$SYSTEM
IMPORTing STORAGE AREA: EMPIDS_LOW
IMPORTing STORAGE AREA: EMPIDS_MID
   .
   .
   .
SQL>
```

In an IMPORT statement, if you do not specify an existing database element, the characteristics of that element do not change. You delete an element by including a DROP section. You modify an element by including a CREATE section that specifies that element's name. If you do include a CREATE section to modify an existing element, the CREATE statement defaults are the SQL defaults rather than characteristics that the element definition includes when the database is exported.

For more information about using the IMPORT and EXPORT statements, see Section 7.11.

7.6.4 Moving Storage Areas

The RMU Move_Area command allows you to move one or more storage areas to different read/write and WORM optical disks. This operation can be performed on line with users attached to the database but excluded from access to storage areas being moved during the move operation. The RMU Move_Area command lets you modify certain storage area and snapshot file parameters. See the *Oracle RMU Reference Manual* for information about which storage area and snapshot file parameters you can modify when you use the RMU Move_Area command.

The RMU Move_Area command, like the RMU Backup command, processes all files simultaneously and eliminates the use of intermediate storage media in these operations.

See Table 9–3 in Section 9.5.1 for information on the privileges required for the RMU Move_Area command.

Example 7–29 shows how you can move the DEPARTMENTS storage area and snapshot file to another disk device in an online operation. When you perform the move area operation on line, you take out an exclusive lock on the storage area until the operation completes.

Example 7–29 Moving a Storage Area and Related Snapshot Files to a Different Disk Device

```
$ RMU/MOVE_AREA mf_personnel /ONLINE /AREA -
_$ /DIRECTORY=$2$DUA1:[PERS.STOR.MFPERS] DEPARTMENTS
```

Using the Directory qualifier moves both the storage area and its associated snapshot file to the specified directory.

If you want to move the DEPARTMENTS snapshot file to a different device from the DEPARTMENTS storage area, specify the Snapshots qualifier, as shown in Example 7–30.

Example 7–30 Moving a Storage Area and Related Snapshot Files to Two Different Disk Devices

```
$ RMU/MOVE_AREA mf_personnel /ONLINE /AREA -
_$ /DIRECTORY=$2$DUA1:[PERS.STOR.MFPERS] -
_$ /SNAPSHOTS=FILE=$2$DUA2:[MFPERS.STOR.MFPERS]DEPARTMENTS.SNP DEPARTMENTS
```

Note

For a single-file database, you must specify the name of the database root file in the Root qualifier. If you omit the root file name, the name defaults to a blank file name, such as .rdb and .snp.

Modifying two storage area characteristics, SPAM threshold values for mixed page format storage areas and page size, with an RMU Move_Area command is not as beneficial as an export and import operation for the following reasons:

- The storage area is not reorganized.

Old data remains in place, based on the previous page size and SPAM thresholds. Rows, if fragmented, are still fragmented.

- Only new data is stored, based on these new page sizes and SPAM threshold values.

  Over time, as data is archived and replaced by new data, this operation gradually reorganizes your database. However, the preferred reorganizing tools are the ALTER DATABASE statement for most database changes and the EXPORT and IMPORT statements for certain changes.

Caution

The RMU Move_Area command has parameter qualifiers with positional semantics. Depending on whether the qualifiers are positioned before the first parameter or after parameters, Oracle RMU uses the qualifiers as global or local qualifiers. See the *Oracle RMU Reference Manual* for more information on the positional semantics of RMU command qualifier parameters.

7.6.5 Moving Read/Write Data to Write-Once Storage Areas

If you have list (segmented string) data in a read/write storage area, you can move the data to a write-once storage area on a write-once, read-many (WORM) optical device.

To move a storage area that contains list data, you must use the RMU Move_Area command. You cannot use the SQL ALTER STORAGE MAP statement.

In the sample mf_personnel database, the RESUME column of the RESUMES table is stored in RESUME_LISTS, a read/write storage area. Example 7–31 shows how to change the storage area from a read/write storage area to a write-once storage area by moving the storage area, along with the list data stored in it, to a WORM device.

Example 7–31 Moving List Data from a Read/Write Storage Area to a Write-Once Storage Area

```
SQL> -- Use SQL to display the characteristics of the storage area.
SQL> --
SQL> ATTACH 'FILENAME mf_personnel';
SQL>  SHOW STORAGE AREA RESUME_LISTS
         Access is:      Read write
         Page Format:    Mixed
         Page Size:      6 blocks
         Area File:      DBDISK:[PERS_DB]RESUME_LISTS.RDA;3
         Area Allocation:          30 pages
         Area Extent Minimum:      99 pages
         Area Extent Maximum:      9999 pages
         Area Extent Percent:      20 percent
         Snapshot File:  DBDISK:[PERS_DB]RESUME_LISTS.SNP;2
   .
   .
   .
SQL> EXIT
$
$ ! Use the RMU Move_Area command to move the storage area to a WORM device.
$ ! Specify the Worm qualifier to change the storage area to a write-once
$ ! storage area.
$ !
$ RMU/MOVE_AREA mf_personnel /ONLINE /AREA -
_$   RESUME_LISTS /WORM /FILE = ODA0:[DATABASE]RESUME_LISTS -
_$   /SNAPSHOTS=(ALLOCATION=3)
$
$ SQL
SQL> ATTACH 'FILENAME mf_personnel';
SQL> --
SQL> -- The storage area is now a write-once storage area on a WORM device:
SQL> --
SQL>  SHOW STORAGE AREA RESUME_LISTS
     RESUME_LISTS
         Access is:      Write once
         Page Format:    Mixed
         Page Size:      6 blocks
         Area File:      ODA0:[DATABASE]RESUME_LISTS.RDA;1
         Area Allocation:          30 pages
         Area Extent Minimum:      99 pages
         Area Extent Maximum:      9999 pages
         Area Extent Percent:      20 percent
```

(continued on next page)

Example 7–31 (Cont.) Moving List Data from a Read/Write Storage Area to a Write-Once Storage Area

```
         Snapshot File:  DBDISK:[PERS_DB]RESUME_LISTS.SNP;2
   .
   .
   .
SQL> EXIT
```

When you move a storage area to a WORM device, do not move the snapshot file because the snapshot file must reside on a read/write device. The associated snapshot file for the relocated storage area remains in its original location when you use the File qualifier to move the storage area.

Remember that you should allocate a small number of pages to the snapshot file, as mentioned in Section 4.5.

7.6.6 Moving Data from a Write-Once Storage Area

You can move a write-once storage area from a WORM device and change it to a read/write or read-only storage area. Example 7–32 shows how to make this change with the RESUME_LISTS storage area.

Example 7–32 Moving List Data from a Write-Once Storage Area to a Read/Write Storage Area

```
$ RMU/MOVE_AREA mf_personnel/ONLINE /AREA  RESUME_LISTS -
_$   /NOWORM /FILE = DBDISK:[PERS_DB]RESUME_LISTS /SPAMS
```

When you move list data from a write-once storage area to a read/write storage area, make sure you enable the creation of SPAM pages using the Spams qualifier.

7.6.7 Adding List Data to Write-Once Storage Areas

You can alter a storage map to add new list data to an existing or a new write-once storage area. Suppose you add the column EMPLOYEE_PICTURE to the RESUMES table and that the column is a LIST data type. You can add a storage area on the WORM device and modify the storage map so that the EMPLOYEE_PICTURE data is stored in the new storage area.

Example 7–33 shows how to add a column of the LIST data type, add a storage area, and alter the storage map.

Example 7–33 Adding New List Data to a Write-Once Storage Area

```
SQL> -- Create a write-once storage area.
SQL> --
SQL> ALTER DATABASE FILENAME mf_personnel
cont> ADD STORAGE AREA RESUME_LISTS2
cont>      FILENAME ODA0:[DATABASE]RESUME_LISTS2
cont>      PAGE FORMAT IS MIXED
cont>      SNAPSHOT ALLOCATION IS 3
cont>      WRITE ONCE;
SQL> --
SQL> ATTACH 'FILENAME mf_personnel';
SQL> --
SQL> -- Add a new column to the RESUMES table.
SQL> --
SQL> ALTER TABLE RESUMES
cont>       ADD COLUMN EMPLOYEE_PICTURE LIST OF BYTE VARYING;
SQL> --
SQL> SHOW STORAGE AREA RESUME_LISTS2
     RESUME_LISTS2
         Access is:      Write once
         Page Format:    Mixed
         Page Size:      2 blocks
         Area File:      ODA0:[DATABASE]RESUME_LISTS.RDA;1
   .
   .
   .
SQL> --
SQL>  SHOW STORAGE MAP LISTS_MAP
     LISTS_MAP
 For Lists
 Store clause:          STORE LISTS IN RESUME_LISTS for (RESUMES)
                         IN RDB$SYSTEM
SQL> ALTER STORAGE MAP LISTS_MAP
cont>      STORE LISTS IN RESUME_LISTS
cont>      FOR (RESUMES.RESUME)
cont>      IN RESUME_LISTS2
cont>      FOR (RESUMES.EMPLOYEE_PICTURE)
cont>      IN RDB$SYSTEM;
```

(continued on next page)

Example 7–33 (Cont.) Adding New List Data to a Write-Once Storage Area

```
SQL> SHOW STORAGE MAP LISTS_MAP
     LISTS_MAP
 For Lists
 Store clause:          STORE LISTS IN RESUME_LISTS
          FOR (RESUMES.RESUME)
          IN RESUME_LISTS2
          FOR (RESUMES.EMPLOYEE_PICTURE)
          IN RDB$SYSTEM
SQL>
```

7.6.8 Modifying Read/Write Storage Areas to Read-Only Storage Areas

If you have stable data you do not expect to change, you can change the access to read-only by modifying the storage area for the data to a read-only storage area. Example 7–34 shows how to change the DEPARTMENTS storage area to read-only access.

Example 7–34 Modifying a Read/Write Storage Area to Read-Only Access

```
SQL> ALTER DATABASE FILENAME mf_personnel
cont>      ALTER STORAGE AREA DEPARTMENTS READ ONLY;
```

To add rows to a table stored in a read-only storage area, just change the read-only attribute to read/write using the ALTER DATABASE statement. See the *Oracle Rdb7 Guide to Database Maintenance* for developing backup and restore operation strategies for databases containing read-only storage areas.

7.6.9 Deleting Storage Areas

To delete a storage area, you use the DROP STORAGE AREA clause of the ALTER DATABASE statement. You can specify that the delete is restricted or cascading. When you use a restricted delete, Oracle Rdb deletes only the storage area. When you use a cascading delete, Oracle Rdb deletes the storage area and deletes or modifies all objects that refer to the storage area.

Consider the following restrictions before deleting a storage area:

- You cannot delete a storage area if it is the DEFAULT STORAGE AREA, the default LIST STORAGE AREA, or the RDB$SYSTEM storage area.
- If you use the RESTRICT keyword, you cannot delete a storage area if any database object, such as a storage map, refers to the area.
- If you use the RESTRICT keyword, you cannot delete a storage area if there is data in it.

- If you use the CASCADE keyword, Oracle Rdb modifies all objects that refer to the storage area so that they no longer refer to it. However, Oracle Rdb does not drop objects if doing so would make the database inconsistent. Keep in mind the following points:
  - If the storage area is the only area in the storage map, Oracle Rdb drops the storage area and all objects that refer to it.
  - If the storage area is not the only area in the storage map *and* the storage map is defined as PARTITIONING IS NOT UPDATABLE (that is, the area is strictly partitioned), Oracle Rdb drops the storage area and all objects that refer to it.
  - If the storage area contains only an index or part of an index, but no matching data for the table, Oracle Rdb does not drop the area because doing so would make the database inconsistent.
  - If a hashed index and a table are in the same storage area and the mapping for the hashed index is not the same as the mapping for the table, Oracle Rdb does not drop the storage area.
  - If a storage area contains a table that contains constraints, Oracle Rdb drops the area if doing so will maintain the consistency of the database.

    For example, the foreign key constraint JH_DEPT_CODE_FOREIGN refers to the DEPARTMENTS table. Because the entire table is stored in one storage area, Oracle Rdb drops the DEPARTMENTS table, including the constraint JH_DEPT_CODE_FOREIGN, when you drop the DEPARTMENTS storage area. The database remains consistent.

    On the other hand, Oracle Rdb will not drop the EMPIDS_LOW storage area, even if it is defined as PARTITIONING IS NOT UPDATABLE. The EMPIDS_LOW storage area contains parts, but not all, of the EMPLOYEES and JOB_HISTORY tables. Foreign key constraints, including DEGREES_FOREIGN1, refer to the EMPLOYEES table. Because dropping the storage area would result in deleting only part of the data in the table, Oracle Rdb cannot drop the foreign key constraints and still maintain the consistency of the database.
- Because dropping storage areas is an online operation and reserving storage areas is an offline operation, you cannot use the DROP STORAGE AREA clause and the RESERVE STORAGE AREA clause in the same ALTER DATABASE statement.

Example 7–35 shows that you cannot delete the DEPARTMENTS storage area using the RESTRICT keyword because other objects refer to it.

Example 7–35 Deleting a Storage Area Using the RESTRICT Keyword

```
SQL> -- When you use the RESTRICT keyword, Oracle Rdb does not
SQL> -- let you drop a storage area if other objects refer to it:
SQL> ALTER DATABASE FILENAME mf_personnel
cont>      DROP STORAGE AREA DEPARTMENTS RESTRICT;
%RDB-F-BAD_DPB_CONTENT, invalid database parameters in the database parameter
block (DPB)
-RDMS-E-MAPREFAREA, error deleting storage area DEPARTMENTS referenced by
DEPARTMENTS_MAP
```

Because you must consider the restrictions before deleting a storage area with the RESTRICT keyword, you should use the following procedure:

1. Determine if there are any placement indexes in the storage area.

   If there are placement indexes, start a read/write transaction to modify the storage map and specify the NO PLACEMENT VIA INDEX clause.

   You cannot delete an placement index to which a storage map refers. You must modify the storage map and either specify the name of another placement index or specify NO PLACEMENT VIA INDEX before you can delete the index.

2. Determine if there are any tables in the storage area. If there are none, continue to step 3; otherwise perform either step 2a or 2b and then step 2c:

   a. Start a read/write transaction and use the ALTER STORAGE MAP statement to relocate the table or tables into other existing storage areas.

   b. Start a read/write transaction and delete all tables in the storage area to empty the storage area of data.

   c. Start a read/write transaction and delete the storage map that refers to the storage area.

   Note that the DROP TABLE CASCADE statement automatically deletes all view definitions based on the table, indexes defined for any column or columns in the table, any constraints that refer to the table, and all triggers that refer to the table. SQL drops any PLACEMENT VIA INDEX clause from the storage map definition.

3. Start a read/write transaction to delete the storage area.

To drop a storage area and all objects that refer to it, use the CASCADE keyword. Oracle Rdb drops the area quickly and removes it from all usage by the database.

When you use the CASCADE keyword to drop the DEPARTMENTS storage area, Oracle Rdb deletes the DEPARTMENTS_MAP storage map because the DEPARTMENTS storage area is the only area mentioned in the storage map. Because the DEPARTMENTS storage area refers to the DEPARTMENTS table, which in turn refers to the DEPARTMENTS_INDEX and a foreign key constraint that refers to the JOB_HISTORY table, Oracle Rdb deletes the table, index, and foreign key constraint, as shown in Example 7–36.

Example 7–36 Deleting a Storage Area Using the CASCADE Keyword

```
SQL> -- To delete the area and all objects that refer to it, use the CASCADE
SQL> -- keyword:
SQL> ALTER DATABASE FILENAME mf_personnel
cont>      DROP STORAGE AREA DEPARTMENTS CASCADE;
SQL> --
SQL> -- Use SHOW statements to see if the table and the DEPARTMENTS_MAP
SQL> -- storage map still exist.
SQL> ATTACH 'FILENAME mf_personnel';
SQL> SHOW TABLE DEPARTMENTS;
No tables found
SQL> SHOW STORAGE MAP DEPARTMENTS_MAP
 No Storage Maps Found
```

Example 7–37 shows that Oracle Rdb does not delete a storage area if the storage map contains more than one area and is defined as PARTITIONING IS UPDATABLE.

Example 7–37 Attempting to Delete an Updatable Storage Area

```
SQL> ALTER DATABASE FILENAME mf_personnel
cont>      DROP STORAGE AREA EMPIDS_LOW CASCADE;
%RDB-E-NO_META_UPDATE, metadata update failed
RDMS-E-NODACSTRICT, PARTITIONING UPDATABLE specified for storage map
EMPLOYEES_MAP
```

As explained earlier, even if the EMPIDS_LOW storage area is defined as PARTITIONING IS NOT UPDATABLE, Oracle Rdb will not drop the area because doing so would make the database inconsistent. In Example 7–38, EMPIDS_LOW_STRICT is a storage area that is similar to EMPIDS_LOW, except that EMPIDS_LOW_STRICT is defined as PARTITIONING IS NOT UPDATABLE. Oracle Rdb does not drop the storage area; doing so would violate a constraint, thus make the database inconsistent.

Example 7–38 Attempting to Delete a Storage Area

```
SQL>  ALTER DATABASE FILENAME mf_personnel
cont>       DROP STORAGE AREA EMPIDS_LOW_STRICT CASCADE;
%RDB-E-INTEG_FAIL, violation of constraint SALARY_HISTORY_STR_FOREIGN1 caused
operation to fail
-RDB-F-ON_DB, on database SQL_USER1:[GREMBOWICZ.V70]MF_PERSONNEL.RDB;1
```

Caution

When you delete a storage area and snapshot files are enabled, Oracle Rdb automatically deletes the snapshot file for that storage area. Do not delete a snapshot file using an operating system command because you will corrupt the database. Because the .rdb file is not updated when you delete any database file using operating system commands and still contains the pointers to all deleted database files, your database is corrupt. Because Oracle Rdb cannot find the deleted snapshot file, you must restore and recover the database to its most current state.

7.7 Modifying Indexes

You can modify the following characteristics of indexes:

- The description
- Run-length compression
- In the STORE clause, the columns whose values are used as limits for partitioning the index across multiple storage areas
- In the STORE clause, whether the index is located in one storage area or across multiple storage areas based on a specified maximum value for each storage area
- For sorted indexes, the node size, its percent fill value, and usage type (whether for update or query)
- For hashed indexes, whether an ordered or scattered placement algorithm is used
- That the index is disabled
- The storage area that holds the index.

You cannot change a sorted index to a hashed index or vice versa, a unique index to a duplicate index or vice versa, or the columns that comprise the key for the index. You cannot modify whether a sorted index is ranked or nonranked nor how duplicates are stored.

7.7.1 Modifying Sorted Indexes

This section discusses why and how you modify sorted index definitions.

For sorted indexes, you must balance the needs of update-intensive transactions versus read-intensive transactions in the structure of the index:

- Read-intensive transactions are improved by minimizing I/O operations. To do this, specify index characteristics that create large index nodes and high fill factors. This results in fewer B-tree levels but at the expense of increasing lock conflicts for update-intensive transactions.
- Update-intensive transactions are improved by minimizing lock conflicts. To do this, specify index characteristics that create small index nodes and low fill factors. This results in more B-tree levels but at the expense of increasing I/O operations for read-intensive transactions.

When you tune and optimize I/O operations and locking for both read-intensive and update-intensive transactions, you may need to modify indexes frequently to achieve the right balance. The ALTER INDEX statement lets you experiment with these options.

Return to the example, described in Section 7.6.1, of adding the JOB_ASSIGNMENTS table to the mf_personnel database. You define a sorted index, JOB_ASSIGNMENTS_SORT, to include the columns EMPLOYEE_ID, JOB_CODE, and ASSIGNMENT_NAME to form a three-part multisegmented key for this table. Assume that the update volume exceeds by a factor of 100 the read-only transaction volume, and that a sorted index is defined to optimize update use of the database. The following example shows the index definition:

```
SQL> CREATE UNIQUE INDEX JOB_ASSIGNMENTS_SORT ON JOB_ASSIGNMENTS
cont>       (ASSIGNMENT_NAME, JOB_CODE, EMPLOYEE_ID)
cont>       TYPE IS SORTED
cont>       NODE SIZE 200
cont>       PERCENT FILL 50;
```

This index is calculated to have a node size slightly larger than the minimum recommended when the percent fill value is set to 50 percent. These calculations are based on formulas presented in the Section 4.7 for improving sorted index performance for update-intensive applications. The combination of the low percent fill value (50) and small node size (200) creates the potential for fewer lock conflicts for update-intensive transactions, but may increase I/O operations because the B-tree levels may become numerous due to many

duplicate node records. Note that because the STORE clause is not specified, the the index is stored in the default RDB$SYSTEM storage area.

Because the primary purpose of this index is to facilitate update-intensive transactions, you could make a trade-off to reduce the potential for increased I/O operations, especially as the database grows. Assume that several top-level managers are the primary readers. You want performance to be reasonably good for these readers, yet you want to minimize the potential lock conflicts for your update-intensive transactions. You decide to make the changes shown in Example 7–39 using the ALTER INDEX statement.

Example 7–39 Modifying an Index Definition

```
SQL> ALTER INDEX JOB_ASSIGNMENTS_SORT
cont>      NODE SIZE 1950
cont>      USAGE UPDATE;
```

When the minimum value for the node size was originally specified, a minimum of three entries fit in an index node. By modifying this value to a larger size of 1950 bytes, 15 entries should fit in an index node. This value is based partly on the fact that the usage type of UPDATE has a default percent fill value of 70 percent. This index structure change may result in fewer B-tree levels and reduce I/O operations. But with larger node records, the potential for locking a node record increases by five times if a "key" column is updated for an employee, for example, if the ASSIGNMENT_NAME column is changed for an employee.

Some experimentation with these index characteristics is necessary to minimize I/O operations so that performance is good and lock conflicts are acceptable. Use the RMU Analyze command with the Indexes qualifier to measure the results of using different values for these two characteristics.

7.7.2 Modifying Hashed Indexes

This section describes why and how you modify hashed index definitions.

Return again to the example of adding the JOB_ASSIGNMENTS table to the mf_personnel database. To reduce I/O operations, you define one or more hashed indexes based on the type of queries that are made. Assume that you expect the tables to grow enormously over time. You also determine that one common transaction makes queries based on an exact match on the EMPLOYEE_ID first, followed by an exact match on the JOB_ASSIGNMENT column. As a result, you define the two hashed indexes as shown in the following example:

```
SQL> CREATE UNIQUE INDEX JOB_ASSIGN_EMPID_HASH ON JOB_ASSIGNMENTS
cont>       (EMPLOYEE_ID)
cont>       TYPE IS HASHED
cont>       STORE IN EMP_JOBHIST_JOBASSIGN;
SQL>
SQL> CREATE INDEX JOB_ASSIGNMENTS_HASH ON JOB_ASSIGNMENTS
cont>       (ASSIGNMENT_NAME)
cont>       TYPE IS HASHED
cont>       STORE IN JOB_ASSIGNMENTS;
```

The JOB_ASSIGN_EMPID_HASH hashed index definition does not permit duplicate index records, is stored in the EMP_JOBHIST_JOBASSIGN storage area, and has the column EMPLOYEE_ID defined as the key. The JOB_ ASSIGNMENTS_HASH hashed index permits duplicate index records, stores the index structure within the JOB_ASSIGNMENTS storage area, and has the column ASSIGNMENT_NAME defined as the key.

Placing the index structure for the JOB_ASSIGN_EMPID_HASH hashed index in the same storage area as the data rows achieves the maximum performance gains for use of a hashed index. To maximize this benefit, carefully calculate the storage area parameters. See the Section 4.8 to determine how to calculate the amount of space that a hashed index and associated data rows use on a page. As a result of your calculations, you may want to create new storage areas. You might determine that, to further reduce disk contention for the EMP_JOBHIST_JOBASSIGN storage area, you should horizontally partition the rows across two storage areas and specify mixed page formats, new page sizes, SPAM intervals, and SPAM thresholds, among other things.

If you are also using a sorted index, such as the JOB_ASSIGNMENTS_SORT index defined in Section 7.7.1, you can make allowances for it as well, such as relocating it.

One additional consideration is to cluster the related tables, EMPLOYEES, JOB_HISTORY, and JOB_ASSIGNMENT, within these two new storage areas to control row placement and reduce I/O operations related to reading the parent and child rows. All these requirements need to be balanced against each other and one or more strategies chosen, based on the predominant types of transactions. See Section 7.6 for more information on these topics.

When you want to spread one or more tables over two or more storage areas (horizontal or vertical partitioning), you can modify the index with the STORE clause in the ALTER INDEX statement. Moving your hashed indexes into these storage areas helps minimize I/O operations. The STORE clause permits you to specify the storage area in which to store the index structure. The maximum value of the index key defined in the USING clause determines in which storage area the row is initially stored. For example, to spread the JOB_ASSIGN_EMPID_HASH hashed index over two storage areas, EMP_

JOBHIST_JOBASSIGN_LOW and EMP_JOBHIST_JOBASSIGN_HIGH, modify the index as shown in Example 7–40.

Example 7–40 Partitioning a Hashed Index Across Two Storage Areas

```
SQL> ALTER INDEX JOB_ASSIGN_EMPID_HASH
cont>      STORE USING (EMPLOYEE_ID)
cont>            IN EMP_JOBHIST_JOBASSIGN_LOW WITH LIMIT OF ('00499')
cont>      OTHERWISE IN EMP_JOBHIST_JOBASSIGN_HIGH;
```

If an index does not contain an overflow partition (defined by the OTHERWISE clause), you can add new partitions to the index without reorganizing the storage areas. Note that this is true of sorted indexes as well as hashed indexes. Example 7–41 shows how to add another partition, PERSONNEL_4, to the EMP_MAP storage map.

Example 7–41 Adding Partitions to Indexes Without Overflow Areas

```
SQL> ALTER INDEX EMP_HASH_INDEX
cont>      STORE USING (EMPLOYEE_ID)
cont>            IN PERSONNEL_1 WITH LIMIT OF ('00399')
cont>            IN PERSONNEL_2 WITH LIMIT OF ('00699')
cont>            IN PERSONNEL_3 WITH LIMIT OF ('10000')
cont>            IN PERSONNEL_4 WITH LIMIT OF ('10399');
```

Because Oracle Rdb does not need to move or reorganize data, you can quickly modify indexes that do not contain overflow partitions.

You should create new storage maps or modify existing storage maps when you make the following types of new index definitions or changes:

- If you create a new index to replace an existing index and the columns that comprise the index change, modify the storage map definition.
- If you modify the storage characteristics of the index such as the column name or names of the USING argument, the partitioning limit values, or the storage area name or names, modify the storage map definition if you want data rows to be stored accordingly.
- If you modify an existing hashed index definition to partition index rows across multiple storage areas, modify the existing storage map definition to also store data rows across the same storage areas if you want to maximize retrieval performance.
- If you want to relocate existing data rows that are partitioned across multiple storage areas to one storage area to match the modified index definition, modify the existing storage map definition.

- If you modify either the USING column-name value or the WITH LIMIT OF values, you can reorganize the data and index records within the storage area or areas.

See Section 7.9 and the *Oracle Rdb7 SQL Reference Manual* for information on modifying storage maps.

When you create new indexes, Oracle Rdb stores the index records according to the new index definition, but it still stores data rows according to your current (unmodified) storage map definition, or by default in the RDB$SYSTEM or default storage area when no storage map definition exists.

For example, if you create a new index to change a single-segmented key to a multisegmented key in a hashed index, but neglect to make similar changes in the ALTER STORAGE MAP statement, Oracle Rdb places your data in the storage area according to the old index definition, not the new index definition. Furthermore, you cannot delete the old index because Oracle Rdb still refers to it.

If you omit the STORE IN clause in the ALTER STORAGE MAP statement, Oracle Rdb moves your data to the default or RDB$SYSTEM storage area. Consequently, Oracle Rdb might place data rows where they are not desired.

If you want to delete the old index but do not want to specify a new index in the PLACEMENT VIA INDEX clause, specify the NO PLACEMENT VIA INDEX clause in the ALTER STORAGE MAP statement to delete the reference to the old index. Then, delete the old index.

7.7.3 Disabling Indexes

Occasionally, an index becomes corrupt, unsuitable for query optimizations, or unnecessary due to a change in business requirements. If the index is part of a large table in a very large database, deleting the index can take time, especially if the index is stored in a mixed storage area. By using the MAINTENANCE IS DISABLED clause of the ALTER INDEX statement, you can permanently disable an index. Later, when the table can be taken off line, you can delete the index with a DROP INDEX statement.

When an index is disabled, the optimizer does not use it (it is not loaded into the symbol tables for the table), Oracle Rdb does not maintain it (no updates are applied to it), and the EXPORT statement ignores it. Disabling the maintenance of an index breaks the connection between the index and the table, so the index is no longer used. After you disable an index, you cannot enabled it again.

In Example 7–42, the ALTER INDEX statement disables maintenance of the EMP_EMPLOYEE_ID index in the mf_personnel database.

Example 7–42 Disabling the Maintenance of an Index

```
SQL> ATTACH 'FILENAME mf_personnel';
SQL> --
SQL> --  Display information on the EMP_EMPLOYEE_ID index:
SQL> SHOW INDEX EMP_EMPLOYEE_ID
Indexes on table EMPLOYEES:
EMP_EMPLOYEE_ID                with column EMPLOYEE_ID
  No Duplicates allowed
  Type is Sorted
  Compression is DISABLED
SQL> --
SQL> -- Use the MAINTENANCE IS DISABLED clause to disable
SQL> -- maintenance for the EMP_EMPLOYEE_ID index:
SQL> ALTER INDEX EMP_EMPLOYEE_ID
cont>      MAINTENANCE IS DISABLED;
SQL> COMMIT;
SQL> --
SQL> -- Show that maintenance has been disabled for the index:
SQL> SHOW INDEX EMP_EMPLOYEE_ID
Indexes on table EMPLOYEES:
EMP_EMPLOYEE_ID                with column EMPLOYEE_ID
  No Duplicates allowed
  Type is Sorted
  Compression is DISABLED
  Index is no longer maintained
SQL>
```

You cannot specify any other clauses when you specify the MAINTENANCE IS DISABLED clause.

7.8 Deleting Indexes

You may want to delete an index for the following reasons:

- By monitoring index use you determine that the index offers little performance advantage or actually degrades database performance.

  Chapter 3 provides some reasons why indexes should be deleted. In addition, the *Oracle Rdb7 Guide to Database Performance and Tuning* explains how to analyze the usefulness of an index.

- You want to delete a column on which an index is based.
- You want to change whether the index allows or disallows duplicate values in the index key.

- You want to change the index structure from sorted to hashed, or vice versa.
- You want to change to or from a ranked sorted index.
- You want to change duplicate compression characteristics.

To delete an index, use the DROP INDEX statement as shown in the following example:

```
SQL> DROP INDEX EMP_LAST_NAME;
```

SQL automatically deletes any indexes associated with a table when you delete the table.

If a storage map refers to the index, you cannot delete that index. To delete the index, first you must take one of the following steps:

- Specify a new index in the PLACEMENT VIA INDEX clause of the ALTER STORAGE MAP statement.
- Specify the NO PLACEMENT VIA INDEX clause in the ALTER STORAGE MAP statement.
- Specify the MAINTENANCE IS DISABLED clause of the ALTER INDEX statement.

For more information about situations when database activity precludes using the DROP INDEX statement, see the *Oracle Rdb7 SQL Reference Manual*.

To delete an index in a multischema database, you must qualify the name of the index with the names of the catalog and schema that contain it:

```
SQL> DROP INDEX ADMINISTRATION.PERSONNEL.EMP_EMPLOYEE_ID;
```

For more information on qualifying names of elements in multischema databases, see Section 5.5.

7.9 Modifying Storage Maps

Using the ALTER STORAGE MAP statement, you can modify the following characteristics of storage maps:

- Moving the rows for a table from one storage area to another
- The way rows are stored: by index to select the target page or randomly
- Adding new storage areas to the storage map
- Reorganizing data rows to new target pages within a storage area, to new target pages across storage areas, or to new target pages in new storage areas

- Columns comprising the key that determines the storage area in which a row is initially stored
- Which tables are stored in a storage area or how a table is partitioned across multiple storage areas
- Disabling or enabling compression when the row is stored
- SPAM threshold values for new logical areas with uniform format pages
- A NOT UPDATABLE storage map to an UPDATABLE one. (You cannot change from an UPDATABLE storage map to a NOT UPDATABLE one.)

In addition, you can use a combination of CREATE STORAGE MAP and ALTER STORAGE MAP statements to move tables that contain data from the default storage area to separate storage areas. See Section 7.9.1 for more information.

You can also use a combination of CREATE STORAGE MAP and ALTER STORAGE MAP statements to move certain system tables to separate storage areas. See Section 7.9.2 for more information.

Keep in mind the following points when you modify storage maps:

- You cannot modify the vertical partitioning of a storage map.
- If you modify the storage area in which a table is stored using the STORE clause, Oracle Rdb moves data from the old storage area to the new storage area.
- If you remove a storage area from the list of storage areas in the STORE IN clause, Oracle Rdb moves all rows in that storage area.
- If you modify compression, Oracle Rdb reads and stores all rows again. Oracle Rdb stores new data according to your new definitions.
- You cannot modify threshold values for existing storage areas, as Example 7–49 demonstrates.

Table 7–5 summarizes how rows are moved (among storage areas or among pages within storage areas) when you use the REORGANIZE clause alone and in combination with each of the options that you can use in an ALTER STORAGE MAP statement.

Table 7–5 Summary of Modifying Storage Map Options and Effect on Rows (Moved/Not Moved)

Specified Option	Effect on Rows When Specifying REORGANIZE	Effect on Rows When Not Specifying REORGANIZE
Omit original storage area names and name new storage areas	All rows are moved to new storage areas	All rows are moved to new storage areas
Omit some original storage area names and name some new storage areas	All rows are read and restored among original and new storage areas	Only rows in omitted storage areas are moved to new storage areas; rows in original, existing storage areas are not moved
Only specify the WITH LIMIT OF values whether or not new values are given	All rows are read and restored With AREAS option[1] With PAGES option [2]	Old rows stay where they are; only new rows are stored according to the new limit values
In storage map that contains no overflow partition, specify original and new storage areas using WITH LIMIT OF clause	New rows are stored according to the LIMIT OF values	Same
Remove overflow partition from storage map and specify original and new storage areas using WITH LIMIT OF clause	All rows are read and restored	Only rows in omitted storage area are moved to new storage areas; rows in original, existing storage areas are not moved
In storage map that contains only one area, reorganize rows within the area.	Rows are reorganized within the area.	No effect.
Only modify the COMPRESSION option	All rows are read, the compression characteristic changed, and all rows restored With AREAS option[1] With PAGES option [2]	Same

[1]With the AREAS option, all rows are checked to see if they are in the right storage area; if some are not, they are moved.

[2]With the PAGES option, all rows are checked to see if they are in the right storage area; if some are not, they are moved; then all rows are checked to see if any should be moved within each storage area and these rows are moved if there is space on or closer to the new target page.

(continued on next page)

Table 7–5 (Cont.) Summary of Modifying Storage Map Options and Effect on Rows (Moved/Not Moved)

Specified Option	Effect on Rows When Specifying REORGANIZE	Effect on Rows When Not Specifying REORGANIZE
Only modify the index name in the PLACEMENT VIA INDEX option	All rows are read and restored With AREAS option[1] With PAGES option[2]	Only new rows are stored based on the new index name
Only modify the column names in the USING option	All rows are read and restored With AREAS option[1] With PAGES option[2]	Only new rows are stored based on the new column names

[1]With the AREAS option, all rows are checked to see if they are in the right storage area; if some are not, they are moved.

[2]With the PAGES option, all rows are checked to see if they are in the right storage area; if some are not, they are moved; then all rows are checked to see if any should be moved within each storage area and these rows are moved if there is space on or closer to the new target page.

The following examples show how to use the ALTER STORAGE MAP statement to modify the storage area in which data is located or to specify a particular change in a storage map definition to indicate how new rows are to be stored.

Suppose you modify the location of a table from one storage area to another; it may be advisable to relocate any indexes defined for the same storage area with the ALTER INDEX statement. Similarly, if you partition an index across two or more storage areas, it may be desirable to reflect this modification with the ALTER STORAGE MAP statement.

For example, in the first case, modifying the sorted index JOB_ ASSIGNMENTS_SORT requires that you modify the storage map JOB_ ASSIGNMENTS_MAP. The following example shows the original storage map definition:

```
SQL> CREATE STORAGE MAP JOB_ASSIGNMENTS_SORT_MAP FOR JOB_ASSIGNMENTS
cont>       STORE IN JOB_ASSIGNMENTS;
```

Note that the PLACEMENT VIA INDEX clause is not specified: you can force a faster load of your data by omitting this clause and letting Oracle Rdb use its own set of row placement algorithms.

To move the JOB_ASSIGNMENTS table to a new storage area, you merely omit the name of the old storage area and indicate the name of the new storage area in the ALTER STORAGE MAP statement. For example, if you name the new storage area JOB_ASSIGNMENTS_STOREAREA, all the rows are moved to this new storage area, as shown in Example 7–43.

Example 7–43 Modifying the STORE Clause of the Storage Map Definition

```
SQL> ALTER STORAGE MAP JOB_ASSIGNMENTS_SORT_MAP
cont>      STORE IN JOB_ASSIGNMENTS_STOREAREA;
```

As mentioned previously, to force faster data loading, do not specify the PLACEMENT VIA INDEX clause. After the data is moved to the new storage area, you can modify the storage map and specify the PLACEMENT VIA INDEX clause. This allows all new rows to be placed with the characteristics defined in the JOB_ASSIGNMENTS_SORT index as shown in Example 7–44.

Example 7–44 Specifying the PLACEMENT VIA INDEX Option in a Storage Map Definition

```
SQL> ALTER STORAGE MAP JOB_ASSIGNMENTS_SORT_MAP
cont>      PLACEMENT VIA INDEX JOB_ASSIGNMENTS_SORT;
```

In Example 7–40, the index JOB_ASSIGN_EMPID_HASH is partitioned into two storage areas to cluster, by EMPLOYEE_ID, the EMPLOYEES, JOB_HISTORY, and the JOB_ASSIGNMENT rows on the same page to improve retrieval performance. To store all three tables and associated hashed indexes in these same two storage areas, modify the storage map JOB_ASSIGN_EMPID_HASH_MAP.

The following example shows the original definition for the JOB_ASSIGN_EMPID_HASH_MAP storage map:

```
SQL> CREATE STORAGE MAP JOB_ASSIGN_EMPID_HASH_MAP FOR JOB_ASSIGNMENTS
cont>       STORE IN JOB_ASSIGN
cont>       PLACEMENT VIA INDEX JOB_ASSIGN_EMPID_HASH;
```

You specify the partition with the ALTER STORAGE MAP statement as shown in Example 7–45.

Example 7–45 Adding a Storage Area to a Storage Map Definition

```
SQL> ALTER STORAGE MAP JOB_ASSIGN_EMPID_HASH_MAP
cont>      STORE USING (EMPLOYEE_ID)
cont>            IN EMP_JOBHIST_JOBASSIGN_LOW WITH LIMIT OF ('00499')
cont>      OTHERWISE IN EMP_JOBHIST_JOBASSIGN_HIGH
cont>      PLACEMENT VIA INDEX JOB_ASSIGN_EMPID_HASH;
```

Suppose you add a new storage area, EMP_JOBHIST_JOBASSIGN_MID, to partition the rows across three storage areas to further reduce disk contention. To ensure that all rows are reorganized into three distinct groupings based on the value for EMPLOYEE_ID, you must specify the REORGANIZE clause in the ALTER STORAGE MAP statement as shown in Example 7–46.

Example 7–46 Reorganizing Rows Across Old and New Storage Areas

```
SQL> ALTER STORAGE MAP JOB_ASSIGN_EMPID_HASH_MAP
cont>      STORE USING (EMPLOYEE_ID)
cont>            IN EMP_JOBHIST_JOBASSIGN_LOW WITH LIMIT OF ('00200')
cont>            IN EMP_JOBHIST_JOBASSIGN_MID WITH LIMIT OF ('00400')
cont>      OTHERWISE IN EMP_JOBHIST_JOBASSIGN_HIGH
cont>      PLACEMENT VIA INDEX JOB_ASSIGN_EMPID_HASH
cont>      REORGANIZE;
```

If you do not specify the REORGANIZE option, Oracle Rdb does not move existing rows and stores only new rows according to the new limit values.

If a storage map does not contain an overflow partition (defined by the OTHERWISE clause), you can add new partitions to the storage map without reorganizing the storage areas. Example 7–47 shows how to add another partition to the EMP_MAP storage map.

Example 7–47 Adding Partitions to Storage Maps Without Overflow Areas

```
SQL> ALTER STORAGE MAP EMP_MAP
cont>      STORE USING (EMPLOYEE_ID)
cont>            IN PERSONNEL_1 WITH LIMIT OF ('00399')
cont>            IN PERSONNEL_2 WITH LIMIT OF ('00699')
cont>            IN PERSONNEL_3 WITH LIMIT OF ('10000')
cont>            IN PERSONNEL_4 WITH LIMIT OF ('10399');
```

Because the original storage map did not contain an OTHERWISE clause, you do not need to reorganize the storage areas.

If a storage map contains an overflow partition and you want to alter the storage map to rid it of the overflow partition, you do not need to use the REORGANIZE clause. Oracle Rdb moves the existing data to the appropriate storage area.

If a storage map contains an overflow partition and you want to alter the storage map to change the overflow partition to a partition defined with the WITH LIMIT OF clause, you must use the REORGANIZE clause to move existing data that is stored in the overflow partition to the appropriate storage area.

For example, suppose the JOB_HISTORY table contains a row with an EMPLOYEE_ID of 10001 and the JH_MAP storage map is defined as shown in the following example:

```
SQL> SHOW STORAGE MAP JH_MAP
     JH_MAP
 For Table:              JOB_HISTORY
 Compression is:         ENABLED
 Store clause:           STORE USING (EMPLOYEE_ID)
                               IN PERSONNEL_1 WITH LIMIT OF ('00399')
                               IN PERSONNEL_2 WITH LIMIT OF ('00699')
                         OTHERWISE IN PERSONNEL_3
```

If you want to change the PERSONNEL_3 storage area from an overflow partition to a partition with a limit of 10000, and add the partition PERSONNEL_4, you must use the REORGANIZE clause to ensure that Oracle Rdb moves existing rows to the new storage area. Example 7–48 shows the ALTER STORAGE MAP statement that accomplishes this change.

Example 7–48 Removing Overflow Partitions and Moving Existing Data

```
SQL> ALTER STORAGE MAP JH_MAP
cont>      STORE USING (EMPLOYEE_ID)
cont>            IN PERSONNEL_1 WITH LIMIT OF ('00399')
cont>            IN PERSONNEL_2 WITH LIMIT OF ('00699')
cont>            IN PERSONNEL_3 WITH LIMIT OF ('10000')
cont>            IN PERSONNEL_4 WITH LIMIT OF ('10399')
cont>      REORGANIZE;
```

When you alter a storage map, you can specify threshold values for new logical areas in uniform areas, but you cannot modify threshold values for existing areas, as Example 7–49 shows.

Example 7–49 Specifying Threshold Values for New Areas

```
SQL> -- The existing storage map specifies two areas.
SQL> SHOW STORAGE MAP  TEST_MAP
     TEST_MAP
 For Table:             TEST_TAB
 Placement Via Index:   EMP_IND
 Compression is:        ENABLED
 Partitioning is:       UPDATEABLE
 Store clause:          STORE USING (EMPLOYEE_ID)
              IN TEST_AREA1 (THRESHOLDS ARE (70,80,90))
                 WITH LIMIT OF ('00399')
              IN TEST_AREA2 (THRESHOLDS ARE (70,80,90))
                 WITH LIMIT OF ('00699')
SQL>
SQL> -- Try to modify the threshold values of all existing areas.
SQL> ALTER STORAGE MAP TEST_MAP
cont>      STORE USING (EMPLOYEE_ID)
cont>        IN TEST_AREA1 (THRESHOLDS ARE (70,80,90))
cont>           WITH LIMIT OF ('00400')
cont>        IN TEST_AREA2 (THRESHOLDS ARE (70,80,90))
cont>           WITH LIMIT OF ('00600');
%RDB-E-NO_META_UPDATE, metadata update failed
-RDMS-E-THRESHAREEXI, illegal thresholds usage - area TEST_AREA exists, and
cannot have THRESHOLDS respecified
SQL>
SQL> -- Specify threshold values for only the new area.
SQL> ALTER STORAGE MAP TEST_MAP
cont>      STORE USING (EMPLOYEE_ID)
cont>        IN TEST_AREA
cont>           WITH LIMIT OF ('00200')
cont>        IN TEST_AREA2
cont>           WITH LIMIT OF ('00400')
cont>        IN TEST_AREA3 (THRESHOLDS ARE (70,80,90))
cont>           WITH LIMIT OF ('00600');
```

7.9.1 Creating Storage Maps for Tables That Contain Data

If a table contains data, you usually cannot create a storage map for that table. However, when the table is not explicitly mapped to a storage area (that is, it is located in the default storage area or RDB$SYSTEM storage area), you can create a storage map for the table.

When you create a storage map for such a table, note the following:

- The CREATE STORAGE MAP statement cannot contain COMPRESSION, THRESHOLD, or PLACEMENT VIA INDEX clauses.

- The CREATE STORAGE MAP statement can refer to only one storage area and that area must be the area (the default area or the RDB$SYSTEM area) in which the table is currently stored.

As a result, the new storage map simply describes the current mapping and may not provide you with the mapping you want. After you create the storage map, you can modify it, specifying characteristics and reorganizing the data into more than one storage area.

Suppose that the default storage area contains the table WORKING_BUDGET, which contains data, and you want to partition the table across two storage areas. First, you create the storage area, then you modify it, as shown in Example 7–50.

Example 7–50 Creating a Storage Map for Tables Containing Data

```
SQL> -- Create the storage map.
SQL> CREATE STORAGE MAP WORK_BUDGET_MAP
cont>       FOR WORKING_BUDGET
cont>       STORE IN RDB$SYSTEM;
SQL> --
SQL> ALTER STORAGE MAP WORK_BUDGET_MAP
cont>      STORE USING (CUR_YEAR)
cont>            IN BUDGET_AREA1 WITH LIMIT OF (400000)
cont>            IN BUDGET_AREA2  WITH LIMIT OF (800000);
```

7.9.2 Moving Certain System Tables to Separate Storage Areas

You can move certain optional system tables from the default or RDB$SYSTEM storage area to separate storage areas by creating and then altering a storage map for the table. You must use only certain specified names for the storage map.

Table 7–6 lists the system tables that you can move, the required name for the storage map for each table, and the purpose of the table.

Table 7–6 Optional System Tables and Their Storage Map Names

Table Name	Storage Map Name	Associated Feature
RDB$CATALOG_SCHEMA	RDB$CATALOG_SCHEMA_ MAP	Multischema databases
RDB$CHANGES	RDB$CHANGES_MAP	Replication Option for Rdb

(continued on next page)

Table 7–6 (Cont.) Optional System Tables and Their Storage Map Names

Table Name	Storage Map Name	Associated Feature
RDB$CHANGES_MAX_TSER	RDB$CHANGES_MAX_TSER_ MAP	Replication Option for Rdb
RDB$SYNONYMS	RDB$SYNONYMS_MAP	Multischema databases
RDB$TRANSFERS	RDB$TRANSFERS_MAP	Replication Option for Rdb
RDB$TRANSFER_RELATIONS	RDB$TRANSFER_RELATIONS_ MAP	Replication Option for Rdb
RDB$WORKLOAD	RDB$WORKLOAD_MAP	Workload Collection

To move these system tables, take the following steps:

1. Attach to the database.

   To create a storage map for the RDB$CATALOG_SCHEMA or RDB$SYNONYMS table, you must attach using the MULTISCHEMA IS OFF clause, as shown in the following example. You should not execute any queries on the database, because this may cause the system table to be locked.

   ```
   SQL> ATTACH 'FILENAME MS_TEST2 MULTISCHEMA IS OFF';
   ```

2. Create a storage map for the system table.

   The storage map must be a simple storage map which only lists the name of the storage area in which the table resides. The following example shows how to create a storage map for the RDB$CATALOG_SCHEMA system table:

   ```
   SQL> CREATE STORAGE MAP RDB$CATALOG_SCHEMA_MAP
   cont>       FOR RDB$CATALOG_SCHEMA
   cont>       STORE IN RDB$SYSTEM;
   ```

   The following restrictions apply when you create a storage map for these special system tables:

   - The storage map cannot change the compression attributes.
   - The storage map cannot specify the logical area thresholds.
   - The storage map cannot be placed via an index.
   - The storage map can refer to only one storage area, which must be mapped to the default storage area or the RDB$SYSTEM storage area.
   - The storage map cannot partition the table vertically.

3. After you create the storage map, you can use the ALTER STORAGE MAP statement to move the table to another area, as shown in the following example:

```
SQL> ALTER STORAGE MAP RDB$CATALOG_SCHEMA_MAP
cont>      STORE IN rdb_cat_schema_area;
```

   The following restrictions apply when you modify a storage map for these special system tables:

   - The storage map cannot be placed via an index.
   - The storage map can contain only one storage area.
   - The storage map cannot partition the table vertically.
   - The ALTER STORAGE MAP operation may require exclusive access to the database, as well as the table, while the table data is relocated.

The EXPORT statement does not export the storage maps for system tables. Therefore, if you export and import your database, you must repeat these steps to remap any of these system tables. This restriction may be removed in a future version of Oracle Rdb.

7.10 Deleting Storage Maps

The DROP STORAGE MAP statement deletes storage maps. Example 7–51 shows an example of this statement.

Example 7–51 Deleting a Storage Map

```
SQL> DROP STORAGE MAP TEST_STORAGE_HASH_MAP;
```

If you attempt to delete a storage map that refers to a storage area that contains data, you receive an error message.

7.11 Reorganizing Databases

You can reorganize the database with the EXPORT and IMPORT statements. Two types of reorganization are possible:

- Reorganizing a single-file database into a multifile database with at least one storage area file

  Section 7.11.1 describes how to reorganize the single-file personnel database into the multifile mf_personnel database.

- Reorganizing a multifile database

You can modify specific options for a previously defined multifile database. Changes can be minor or extensive. Examples shown in Section 7.11.2 illustrate typical changes.

7.11.1 Reorganizing a Single-File Database into a Multifile Database

You can change a single-file database into a multifile database. This reorganization is accomplished in much the same way as if you were to define a new multifile database using a CREATE DATABASE statement. In fact, using the IMPORT statement with the WITH EXTENSIONS option makes a copy of the database in an intermediate form called an interchange (.rbr) file, which is a special type of flat file. Specifying and defining the appropriate IMPORT statement clauses creates the multifile database in the same way as the CREATE DATABASE statement does when multifile options are specified.

First, examine the single-file database and devise a plan for reorganizing it. Table 7–7 shows how you might map the tables of the single-file personnel database to storage area files.

Table 7–7 Tables, Storage Areas, and Storage Maps for the Multifile mf_personnel Database

Multifile mf_personnel Database Tables	Storage Areas	Storage Maps
EMPLOYEES JOB_HISTORY	EMPIDS_LOW (EMP_ID <=200)	EMPLOYEE_MAP JOB_HISTORY_MAP
EMPLOYEES JOB_HISTORY	EMPIDS_MID (EMP_ID >200 and EMP_ID <=400)	EMPLOYEE_MAP JOB_HISTORY_MAP
EMPLOYEES JOB_HISTORY	EMPIDS_OVER (EMP_ID >400)	EMPLOYEE_MAP JOB_HISTORY_MAP
SALARY_HISTORY	SALARY_HISTORY	SALARY_HISTORY_MAP
DEPARTMENTS	DEPARTMENTS	DEPARTMENTS_MAP
COLLEGES	COLLEGES	COLLEGES_MAP
DEGREES	DEGREES	DEGREES_MAP
WORK_STATUS	WORK_STATUS	WORK_STATUS_MAP
RESUMES	RESUMES	RESUMES_MAP
RESUMES	RESUME_LISTS	LISTS_MAP
JOBS	JOBS	JOBS_MAP

(continued on next page)

Table 7–7 (Cont.) Tables, Storage Areas, and Storage Maps for the Multifile mf_personnel Database

Multifile mf_personnel Database Tables	Storage Areas	Storage Maps
System tables	RDB$SYSTEM	By default

Second, determine which clauses of the IMPORT statement you use to define storage areas, indexes, and storage maps. Remember, the IMPORT statement is similar to the statements CREATE DATABASE, CREATE STORAGE AREA, CREATE STORAGE MAP, CREATE INDEX, and so on.

Third, use the EXPORT statement to create an interchange (.rbr) file from the single-file personnel database as shown in Example 7–52.

Example 7–52 Creating an Interchange File Using the EXPORT Statement

```
SQL> EXPORT DATABASE FILENAME personnel.rdb INTO newmf_personnel.rbr;
```

Because the WITH EXTENSIONS option is the default, you do not need to specify it. The WITH EXTENSIONS option specifies that information is retained to create the same physical structure as was contained in the original file.

Fourth, implement the physical design changes as part of the IMPORT statement.

1. Define the storage areas.
2. Define the indexes.
3. Define the storage maps that point the data rows to specific storage areas with the STORE clause.

   When you want to cluster rows (child rows with parent rows) for optimizing row access, use the PLACEMENT VIA INDEX clause. Importing data without specifying the PLACEMENT VIA INDEX clause usually causes the load operation to go faster because Oracle Rdb uses its own efficient set of row placement algorithms to accomplish this.

Because of the length and complexity of the IMPORT statement, you should create it as an SQL procedure file in a text editor. From within SQL, execute the procedure that contains the IMPORT statement by preceding the name of the SQL file with the at sign (@). For example, `SQL> @import_newmf_pers`.

Example 7–53 shows the IMPORT statement definitions for the table DEPARTMENTS.

Example 7–53 Reorganizing a Database Using the IMPORT Statement

```
SQL> @import_newmf_pers

IMPORT DATABASE FROM newmf_personnel.rbr FILENAME newmf_personnel.rdb
--
--  Specify storage area options.
--  Specify import options.  (None specified in this example.)
   .
   .
   .
-- Define the DEPARTMENTS storage area.
-- Specify metadata options.
--
CREATE STORAGE AREA DEPARTMENTS FILENAME departments.rda
       ALLOCATION IS 25 PAGES
       PAGE FORMAT IS MIXED
       SNAPSHOT FILENAME departments.snp
       SNAPSHOT ALLOCATION IS 10 PAGES
   .
   .
   .
-- Define a sorted index for the DEPARTMENTS table.
--
CREATE UNIQUE INDEX DEPARTMENTS_INDEX
       ON DEPARTMENTS
       (DEPARTMENT_CODE)
       TYPE IS SORTED
       STORE IN DEPARTMENTS
--
--Define the storage map for the DEPARTMENTS table.
--
CREATE STORAGE MAP DEPARTMENTS_MAP
       FOR DEPARTMENTS
       STORE IN DEPARTMENTS
       PLACEMENT VIA INDEX DEPARTMENTS_INDEX
   .
   .
   .
-- Specify database-wide options or accept the defaults.
--
-- End the procedure with a semicolon.
;
```

Example 7–53 shows how to define a storage area, a sorted index, and a storage map for the DEPARTMENTS table. You carry out this same sequential procedure for the remainder of the planned storage areas, indexes, storage maps, and so on, that make up the newmf_personnel database. You use the ALTER TABLE, ALTER DOMAIN, and other ALTER statements to tailor the database.

7.11.2 Reorganizing a Database for Special Use

You can use many ALTER statements to modify your database, but the resulting changes may force you to create a new database based on the changes you desire. The EXPORT and IMPORT statements are designed to create new databases when modifications to the physical design of your old database affects the way rows are stored in storage areas. The EXPORT and IMPORT statements let you make desired changes and import the data into a new database simultaneously.

For example, assume that you want to modify a single-file database into a multifile database and you want to compare the performance benefits of two physical database designs:

- A multifile physical design that combines several tables in the same storage area and uses both hashed and sorted indexes
- A multifile physical design that places each table in its own storage area and uses only sorted indexes

The databases all contain the same data. You want to test the two new physical designs against a benchmark set of transactions derived from the transaction analysis phase of your logical design. Your goal may be to reduce record lock conflicts on sorted indexes while improving data retrieval by storing tables together.

To test these two physical designs, first create the databases, using the following steps:

1. Reorganize a copy of the single-file database into a multifile database that uses storage areas with uniform pages and sorted indexes.
2. Reorganize a copy of this multifile database so that storage areas that contain two tables and the related hashed indexes are created with a mixed page format, while other storage areas that use only a sorted index are created with uniform page format.

Assume that you have already performed Step 1 and named the database mf_uniform_personnel.rdb. It contains a storage area for each table.

For this example, look at just two tables: EMPLOYEES and JOB_HISTORY. To improve data retrieval, combine both tables into a single storage area. To reduce record lock conflicts, create two hashed indexes, one based on the EMPLOYEE_ID column for the EMPLOYEES table, and the other based on the EMPLOYEE_ID column for the JOB_HISTORY table. Then, map both tables to the same storage area, and specify the PLACEMENT VIA INDEX clause using each hashed index.

You begin Step 2 with the EXPORT statement, as shown in Example 7–54, to create the interchange file mf_unimix_personnel.rbr from the multifile database, mf_uniform_personnel.

Example 7–54 Creating an Interchange File

```
SQL> EXPORT DATABASE FILENAME mf_uniform_personnel.rdb INTO
cont>       mf_unimix_personnel.rbr;
```

Next, import the mf_unimix_personnel.rbr interchange file and create the new multifile database, newmf_personnel.rdb, specifying the changes desired. For illustration, only modifications to the EMPLOYEES and JOB_HISTORY tables are shown in Example 7–55. To keep the example simple, no partitioning of rows is specified and rows for both tables are stored in the defined EMPIDS_ ALL storage area. The IMPORT statement in Example 7–55 defines the storage area, EMPIDS_ALL; the two hashed indexes, EMPLOYEES_HASH and JOB_HISTORY_HASH; and the two storage maps, EMPLOYEES_MAP and JOB_HISTORY_MAP.

Example 7–55 Using an IMPORT Statement to Reorganize a Database

```
IMPORT DATABASE FROM mf_unimix_personnel.rbr
       FILENAME newmf_personnel.rdb
--
CREATE STORAGE AREA EMPIDS_ALL FILENAME empids_all.rda
       ALLOCATION IS 50 PAGES
       PAGE FORMAT IS MIXED
       SNAPSHOT_FILENAME empids_all.snp
       SNAPSHOT ALLOCATION IS 10 PAGES
```

(continued on next page)

Example 7–55 (Cont.) Using an IMPORT Statement to Reorganize a Database

```
   .
   .
   .
--
CREATE UNIQUE INDEX EMPLOYEES_HASH
       ON EMPLOYEES
       (EMPLOYEE_ID)
       TYPE IS HASHED
       STORE IN EMPIDS_ALL
--
CREATE STORAGE MAP EMPLOYEES_MAP FOR EMPLOYEES
       STORE IN EMPIDS_ALL
       PLACEMENT VIA INDEX EMPLOYEES_HASH
--
-- Cluster the rows to place EMPLOYEES and JOB_HISTORY rows
-- with the same EMPLOYEE_ID on the same database page.
--
CREATE UNIQUE INDEX JOB_HISTORY_HASH
       ON JOB_HISTORY
       (EMPLOYEE_ID)
       TYPE IS HASHED
       STORE IN EMPIDS_ALL
--
CREATE STORAGE MAP JOB_HISTORY_MAP FOR JOB_HISTORY
       STORE IN EMPIDS_ALL
       PLACEMENT VIA INDEX JOB_HISTORY_HASH
   .
   .
   .
-- End the procedure with a semicolon.
;
```

Finally, to complete the physical database design change, you may need to cluster together additional tables into other storage areas with additional hashed indexes defined. These changes accomplish the planned modification of the physical design in addition to reloading the data into the newly created database, newmf_personnel.rdb.

Another change that could enhance performance is to partition rows across several storage areas. You implement this change by using the EXPORT and IMPORT statements. Additional information on performance benefits of these and other changes, along with information on measuring performance improvements using the Oracle Rdb Performance Monitor and the RMU

Analyze command, is discussed in the *Oracle Rdb7 Guide to Database Performance and Tuning*.

7.11.3 Creating a Copy of the Database

The RMU Copy_Database command allows you to create duplicates of your database. Like the RMU Restore command, the RMU Copy_Database command permits the modification of certain database-wide, storage area, and snapshot file parameters when the copy operation is performed. These changes include the following:

- Specifying the .aij files or specifying no .aij files
- Requesting that the page checksum be verified for each page of each area that is copied
- Specifying new locations for the root file, each storage area, and each snapshot file
- Specifying a new value for database parameters such as the maximum number of cluster nodes or maximum number of users
- Specifying a new file allocation size for each snapshot file
- Specifying new SPAM thresholds for each mixed storage area that is copied

In addition, to control how the database is copied, you can specify the following options:

- The RMU Copy_Database operation be performed on line, while other users are attached to the database

  All storage areas are locked for read-only access, so the operation is compatible with all other types of transactions except those that require exclusive access.
- The maximum time the copy database operation waits for the quiet-point lock during an online copy database operation

  If you do not specify the Lock_Timeout qualifier, the copy database operation waits indefinitely.
- An option file if one is used to perform this operation
- A different number of buffers to be allocated for each file that is copied to improve the performance of this operation

The RMU Copy_Database command, like the RMU Backup command, processes all files simultaneously and eliminates the use of intermediate storage media in these operations.

See Section 9.5 for information on the privileges required for the RMU Copy_Database command.

Example 7–56 shows how you can make a copy of the mf_personnel database and put all the files on one disk device.

Example 7–56 Copying a Database

```
$ RMU/COPY_DATABASE mf_personnel /DIRECTORY=DUA1:[DUPMF_PERSONNEL]
```

Example 7–57 shows how you can make a duplicate of the mf_personnel database that has the same contents but a different identity from the original database, in online mode with a lock timeout interval of 20 seconds. Example 7–57 also moves the RESUME_LISTS storage area to a WORM optical disk, disables the use of SPAM pages for the RESUME_LISTS storage area, and specifies a smaller file allocation size for the snapshot file.

Example 7–57 Copying the Database and Moving the RESUME_LISTS Storage Area to a WORM Optical Disk Device

```
$ RMU/COPY_DATABASE mf_personnel /ONLINE /LOCK_TIMEOUT=20 /DUPLICATE -
_$ RESUME_LISTS /FILE=WORM1:[PERS.STOR.MFPERS] /WORM /NOSPAMS
_$              /SNAPSHOT=(ALLOCATION=3)
```

Note that the associated snapshot file for the copied storage area remains in its original location when you use the File qualifier to copy the storage area. You can observe the updates to the root file by displaying the root file header contents using the RMU Dump Header command and inspecting the information for the specified storage area and its associated snapshot file.

Modifying the storage area characteristics (such as SPAM threshold values for mixed page format storage areas and page size) with an RMU Copy_Database command is not as useful as an export and import operation for the following reasons:

- The storage area is not reorganized.

  Old data remains in place, based on the previous page size and SPAM thresholds. Rows, if fragmented, are still fragmented.

- Only new data is stored based on any new page sizes and SPAM threshold values.

Over time, as data is archived and replaced by new data, this operation gradually reorganizes your database. However, the preferred reorganizing tools are the ALTER DATABASE statement for most database changes and the EXPORT and IMPORT statements for certain changes.

Caution

The RMU Copy_Database command has parameter qualifiers with positional semantics. Depending on whether the qualifiers are positioned before the first parameter or after parameters, Oracle RMU uses the qualifiers as global or local qualifiers. See the *Oracle RMU Reference Manual* for more information on the positional semantics of RMU command qualifier parameters.

7.11.4 Creating a Copy of an Empty Database

You can create a duplicate copy of an empty database if you want to:

- Experiment with the physical design for a prototype database application.
- Plan a major change to an existing production application, making sure that the parameters you specify for storage areas and storage maps are sound.

To create a duplicate copy of the database, you can either:

- Export the database, specify that it contains no data, and import it to create the duplicate copy as shown in Example 7–58.
- Use an existing .rbr file for the database that contains data, import it, and specify that it contains no data as shown in Example 7–59.

Example 7–58 Copying an Empty Database Using the EXPORT Statement

```
SQL> EXPORT DATABASE FILENAME mf_personnel INTO nodata_mf_personnel
cont>       WITH EXTENSIONS NO DATA;
```

Example 7–59 Copying an Empty Database Using the IMPORT Statement

```
SQL> IMPORT DATABASE INTO mf_personnel_test FILENAME data_mf_personnel
cont>       NO ACL NO DATA TRACE;
```

The TRACE clause provides summary statistics of the I/O operations and CPU time required for an import operation along with the number of faults used, as shown in Example 7–60. This is useful information for such actions as loading data into tables, defining indexes, and defining constraints. A starting message is written before each of these operations, followed by a summary of the number of data segment I/O operations (DIOs), CPU time, and page faults for the operation. Unusually large values may indicate problems.

Example 7–60 Using the TRACE Clause in an IMPORT Operation to Check the Number of I/O Operations and CPU Time Required

```
SQL> IMPORT DATABASE FROM mfpers.rbr FILENAME mf_personnel.rdb TRACE;
   .
   .
   .
Importing table EMPLOYEES
Completed EMPLOYEES.  DIO = 182, CPU = 0:00:01.60, FAULTS = 21
Starting INDEX definition EMP_EMPLOYEE_ID
Completed EMP_EMPLOYEE_ID.  DIO = 25, CPU = 0:00:00.35, FAULTS = 2
   .
   .
   .
Completed import. DIO = 3530, CPU = 0:00:32.97, FAULTS = 2031
SQL>
```

Note

The NO DATA option is not compatible with repository databases (cdd$database.rdb).

If you attempt to export a cdd$database.rdb database with the NO DATA option or to import an .rbr file generated from a cdd$database.rdb file using the NO DATA option, SQL issues an error message stating that the NO DATA option is not valid for repository databases.

7.12 Moving Databases and Database Files

You can easily move the database root file as well as the storage areas and snapshot files to different disks or different systems with the RMU Move_Area command.

Example 7–61 shows how to move the root file and all the storage areas and snapshot files. It invokes an options file that contains new file specifications for the storage areas. (If your database has many storage areas, entering the changes in an options file makes it easier to edit those changes.)

The following example shows the contents of the MF_MOVE_OPTIONS.OPT options file:

```
RDB$SYSTEM      /FILE= USER1:[DB]MFPERS_DEFAULT.RDA -
                /SNAPSHOT=(FILE = USER1:[DB]MF_PERS_DEFAULT.SNP)
EMPIDS_LOW      /FILE= USER2:[DB]EMPIDS_LOW.RDA -
                /SNAPSHOT=(FILE= USER2:[DB]EMPIDS_LOW.SNP)
EMPIDS_MID      /FILE= USER3:[DB]EMPIDS_MID.RDA  -
                /SNAPSHOT=(FILE=USER3:[DB]EMPIDS_MID.SNP)
EMPIDS_OVER     /FILE= USER4:[DB]EMPIDS_OVER.RDA -
                /SNAPSHOT=(FILE=USER4:[DB]EMPIDS_OVER.SNP)
EMP_INFO        /FILE= USER2:[DB]EMPS_INFO.RDA  -
                /SNAPSHOT=(FILE=USER2:[DB]EMPS_INFO.SNP)
JOBS            /FILE= USER2:[DB]JOBS.RDA  -
                /SNAPSHOT=(FILE=USER2:[DB]JOBS.SNP)
RESUMES         /FILE= USER4:[DB]RESUMES.RDA  -
                /SNAPSHOT=(FILE=USER4:[DB]RESUMES.SNP)
RESUME_LISTS    /FILE= USER4:[DB]RESUME_LISTS.RDA  -
                /SNAPSHOT=(FILE=USER4:[DB]RESUME_LISTS.SNP)
SALARY_HISTORY  /FILE= USER4:[DB]SALARY_HISTORY.RDA  -
                /SNAPSHOT=(FILE=USER4:[DB]SALARY_HISTORY.SNP)
DEPARTMENTS     /FILE= USER4:[DB]DEPARTMENTS.RDA  -
                /SNAPSHOT=(FILE=USER4:[DB]DEPARTMENTS.SNP)
```

Example 7–61 Moving a Database

```
$ RMU/MOVE_AREA /AREA /ROOT = USER1:[DB]MF_PERSONNEL   -
_$              mf_personnel  /OPTION=MF_MOVE_OPTIONS.OPT
```

If you store the database definitions in the repository and you move the database root file, you must change the file name as it is stored in the repository. Section 10.14 shows how to change the database file name in the repository.

Remember that you can also use the RMU Backup and RMU Restore commands to move databases in their entirety, to move one or more files to other disks to redistribute the input/output loading, and to perform regular backup and restore operations of Oracle Rdb databases.

The EXPORT and IMPORT statements can also safely move multifile databases. Use the EXPORT and IMPORT statements when you need to move between systems with different versions of the database software or to restructure the database files.

For more information about moving databases and database files, see the *Oracle RMU Reference Manual*. For information about moving storage area files, see Section 7.6.4.

7.13 Deleting Databases, Database Files, and Repository Definitions

To delete a database, including both database files and the associated definitions stored in the repository, use the DROP DATABASE statement. Because this statement deletes all database files, you cannot roll back the operation.

You can specify a path name, a file name, or an alias in the DROP DATABASE statement. The following example specifies the file name:

```
SQL> DROP DATABASE FILENAME mf_personnel_test;
```

OpenVMS VAX OpenVMS Alpha

If the database is stored in the repository, make sure you delete the database in the repository as well. You can delete the database by path name or, if the database is already attached by path name in an ATTACH or CREATE DATABASE statement, you can delete the database by its alias:

```
SQL> DROP DATABASE
cont> PATHNAME SYS$COMMON:[CDDPLUS]DEPT32.FIELDMAN.mf_personnel_test;
   .
   .
   .
SQL> ATTACH 'FILENAME mf_personnel_test';
SQL> DROP DATABASE ALIAS RDB$DBHANDLE;
```

As the previous example shows, SQL uses RDB$DBHANDLE as the default alias. ♦

For more information about the DROP DATABASE statement, see the *Oracle Rdb7 SQL Reference Manual*.

8
Modifying Database Elements

This chapter describes how you can use SQL to modify the definition of database elements. It describes how to modify or delete the following elements:

- Domains
- Tables and their columns
- Constraints
- Triggers
- Views
- Schemas
- Catalogs

For information on modifying aspects of the physical database design and database-wide characteristics such as journaling, buffer size, and storage area parameters, see Chapter 7.

You can change the definitions of many database elements while other users are attached to the database. For information about whether you need exclusive access to the database, see Table 7–2.

8.1 Modifying and Deleting Domains

You can modify a domain definition to change the data type, default value, formatting options, or constraint or you can drop a default value or constraint. You can delete a domain definition when columns no longer refer to it.

To modify an existing domain definition, use the ALTER DOMAIN statement. You can use the SHOW DOMAIN statement to find out which elements have been defined and how you defined them. Example 8–1 shows how to add a default value to the domain JOB_CODE_DOM.

Example 8–1 Modifying Domain Definitions to Add Default Values

```
SQL> SHOW DOMAIN JOB_CODE_DOM
JOB_CODE_DOM                    CHAR(4)
 Comment:       standard definition of job code
SQL> ALTER DOMAIN JOB_CODE_DOM
cont>      SET DEFAULT '?';
SQL> SHOW DOMAIN JOB_CODE_DOM
JOB_CODE_DOM                    CHAR(4)
 Comment:       standard definition of job code
 Oracle Rdb default: ?
SQL> COMMIT;
```

If you change or add a default value for a domain, the change has no effect on any existing data in the database; that is, the rows already stored in the database with columns that contain the *old* default value are not changed.

If you no longer need a default value on a domain, you can delete it by using the DROP DEFAULT clause of the ALTER DOMAIN statement, as shown in Example 8–2.

Example 8–2 Dropping the Default Value from a Domain

```
SQL> -- Drop the default value.
SQL> --
SQL> ALTER DOMAIN JOB_CODE_DOM
cont>      DROP DEFAULT;
SQL> SHOW DOMAIN JOB_CODE_DOM
JOB_CODE_DOM                    CHAR(4)
 Comment:       standard definition of job code
SQL>
```

To modify a domain constraint, first you must drop the domain constraint using the DROP ALL CONSTRAINTS clause of the ALTER DOMAIN statement. Then, you add the new domain constraint using the ADD constraint clause. Example 8–3 shows how to modify the domain constraint for the DATE_DOM domain.

Example 8–3 Modifying Domains to Change Domain Constraints

```
SQL> SHOW DOMAIN DATE_DOM
DATE_DOM                        DATE ANSI
 Oracle Rdb default: NULL
 CHECK:  (DATE_DOM > DATE'1900-01-01' OR
                    DATE_DOM IS NULL)
```

(continued on next page)

Example 8–3 (Cont.) Modifying Domains to Change Domain Constraints

```
SQL> -- Drop the constraint.
SQL> ALTER DOMAIN DATE_DOM
cont>  DROP ALL CONSTRAINTS;
SQL> --
SQL> SHOW DOMAIN DATE_DOM
DATE_DOM                        DATE ANSI
 Oracle Rdb default: NULL
SQL> -- Add the new domain constraint definition.
SQL> --
SQL> ALTER DOMAIN DATE_DOM
cont>       ADD CHECK (VALUE > DATE'1925-01-01')
cont>       NOT DEFERRABLE;
```

When you add (or modify) a domain constraint, SQL propagates the new constraint definition to all the columns that are based on the domain. If columns that are based on the domain contain data that does not conform to the constraint, SQL returns an error. For example, if the column BIRTHDAY in the EMPLOYEES table is based on the domain DATE_DOM and contains a date prior to 1925-01-01, SQL returns the following error:

```
%RDB-E-NO_META_UPDATE, metadata update failed
-RDMS-E-NOT_VALID_FR, field BIRTHDAY in relation EMPLOYEES fails validation
```

If you modify a domain, for example changing the data type, Oracle Rdb checks any constraints that refer to that domain. If the alter operation violates a constraint, you must drop the constraint before you modify the domain.

To delete a domain, you must enter a DROP DOMAIN statement, as shown in the following example:

```
SQL> DROP DOMAIN STANDARD_DATE_DOM;
```

To modify or delete a domain in a multischema database, you must qualify the name of the domain with the names of the catalog and schema that contain it.

```
SQL> ALTER DOMAIN ADMINISTRATION.PERSONNEL.CODE CHAR(5);
```

For more information on qualifying names of elements in multischema databases, see Section 5.5.

For more information on the ALTER DOMAIN and DROP DOMAIN statements, see the *Oracle Rdb7 SQL Reference Manual*.

8.2 Modifying and Deleting Tables

You can use the ALTER TABLE statement to add, modify, and remove columns, to add or remove constraints, to specify primary and foreign keys, and to specify table-specific constraints. You do this by specifying one or more ADD, ALTER, or DROP clauses in the statement. For example, to remove a column from a table, use a statement similar to the following:

```
SQL> ALTER TABLE WORK_STATUS DROP STATUS_NAME;
```

In the ADD column-name clause, you can include the same sort of information (such as data type, default values, and column constraints) that you can for column specifications in the CREATE TABLE statement. In the ALTER COLUMN clause, you can modify column data type, domain name, default value, or column formatting clauses and you can add column constraints. You cannot specify COMPUTED BY in an ALTER column-definition clause.

To add a table constraint, use the ADD CONSTRAINT clause.

To remove a column and any data stored in the column from the table, use the DROP column-name clause.

To delete a column or table constraint from the table definition, use the DROP CONSTRAINT clause of the ALTER TABLE statement. To delete a constraint that was created using the RDO DEFINE CONSTRAINT statement, use the DROP CONSTRAINT statement.

If a column is based on a domain definition, you should modify the domain definition rather than the table definition, using a statement similar to the following:

```
SQL> ALTER DOMAIN STATUS_TYPE_DOM CHAR(5);
```

Remember that modifying a domain definition changes characteristics of all columns that are based on that domain.

8.2.1 Deleting Tables

To delete an entire table and all its data, use the DROP TABLE statement. For example:

```
SQL> DROP TABLE DEGREES;
```

When you delete a table, you can specify whether the deletion is restricted or cascading. By default (or when you use the RESTRICT keyword), when you delete a table, you erase the table definition, all its data, its storage map, and any index definitions that are based on the table.

If you use the CASCADE keyword, you erase all definitions that refer to the table, as well as the table definition, all its data, and index definitions. In addition, if a module (stored procedure or function) refers to the table, Oracle Rdb marks the module as invalid.

Example 8–4 shows what happens when you delete a table and use the CASCADE keyword, and when you delete a table and do not use the CASCADE keyword.

Example 8–4 Deleting Tables

```
SQL> -- Drop the SALARY_HISTORY table using the CASCADE keyword
SQL> -- to drop all the dependent elements.
SQL> --
SQL> DROP TABLE SALARY_HISTORY CASCADE;
View CURRENT_INFO is also being dropped.
View CURRENT_SALARY is also being dropped.
Constraint SH_EMPL_ID_FOREIGN is also being dropped.
Index SH_EMPLOYEE_ID is also being dropped.
Trigger EMPLOYEE_ID_CASCADE_DELETE is also being dropped.
SQL> --
SQL> -- To restore the deleted definitions, you must roll back the
SQL> -- changes to the database.
SQL> --
SQL> ROLLBACK;
SQL> --
SQL> SET TRANSACTION READ WRITE
cont> RESERVING SALARY_HISTORY FOR EXCLUSIVE WRITE;
SQL> --
SQL> -- Drop only the SALARY_HISTORY table.  Do not use the CASCADE keyword.
SQL> --
SQL> DROP TABLE SALARY_HISTORY;
%RDB-E-NO_META_UPDATE, metadata update failed
-RDMS-F-TRGEXI, relation SALARY_HISTORY is referenced in trigger
EMPLOYEE_ID_CASCADE_DELETE
-RDMS-F-RELNOTDEL, relation SALARY_HISTORY has not been deleted
SQL> --
SQL> -- Oracle Rdb does not delete the table, because other definitions refer
SQL> -- to the table.  To delete the table, delete those definitions first.
```

If you use the CASCADE keyword and the table that you delete is referred to by a computed by column in another table, Oracle Rdb generates the value NULL for that column. You can modify the table to drop the column or modify the column to change the definition.

8.2.2 Deleting Tables Quickly

If you want to quickly delete the data in a table, but you want to maintain the metadata definition of the table (perhaps to reload the data into a new partitioning scheme), you can use the TRUNCATE TABLE statement.

The TRUNCATE TABLE statement deletes all the data from a table as quickly as possible. It is equivalent to a DROP TABLE CASCADE statement followed by a CREATE TABLE statement, including the reconstruction of the target and referencing metadata. It does not fire any triggers that perform a delete operation. However, after the TRUNCATE operation, Oracle Rdb revalidates any constraints which refer to the table. If any constraint fails, Oracle Rdb rolls back the TRUNCATE statement.

Example 8–5 shows how to delete the data from the SALARY_HISTORY table with the TRUNCATE TABLE statement.

Example 8–5 Using the TRUNCATE TABLE Statement to Delete Data from Tables

```
SQL> TRUNCATE TABLE SALARY_HISTORY;
SQL>
SQL> -- The table still exists, but the rows are deleted.
SQL> SELECT * FROM SALARY_HISTORY;
0 rows selected
```

8.2.3 Modifying Tables That Are Used in Views or Have Indexes

To delete a column to which a view, index, trigger, or constraint definition refers, you must first delete the dependent definition. The same restriction applies in some cases when you modify a column. You can edit and create dependent definitions again after you change the table. Example 8–6 illustrates some restrictions you may encounter when modifying tables.

Example 8–6 Modifying Tables That Contain Views and Indexes

```
SQL> -- Delete the LAST_NAME column form the EMPLOYEES table.
SQL> --
SQL> ALTER TABLE EMPLOYEES DROP LAST_NAME;
%RDB-E-NO_META_UPDATE, metadata update failed
-RDMS-F-FLDINVIEW, field LAST_NAME is referenced in view CURRENT_SALARY
-RDMS-F-RELFLDNOD, field LAST_NAME has not been deleted from relation
EMPLOYEES
```

(continued on next page)

Example 8–6 (Cont.) Modifying Tables That Contain Views and Indexes

```
SQL> --
SQL> -- You must delete the view before you can delete the LAST_NAME column.
SQL> -- However, you cannot delete the CURRENT_SALARY view until you
SQL> -- delete the view that is based on it.
SQL> --
SQL> DROP VIEW CURRENT_SALARY;
%RDB-E-NO_META_UPDATE, metadata update failed
-RDMS-F-VIEWINVIEW, view CURRENT_SALARY is referenced by view CURRENT_INFO
-RDMS-F-VIEWNOTDEL, view CURRENT_SALARY has not been deleted
SQL> -- Drop all dependent views.
SQL> DROP VIEW CURRENT_INFO;
SQL> DROP VIEW CURRENT_SALARY;
SQL> DROP VIEW CURRENT_JOB;
SQL> --
SQL> ALTER TABLE EMPLOYEES DROP LAST_NAME;
%RDB-E-NO_META_UPDATE, metadata update failed
-RDMS-F-FLDINUSE, field LAST_NAME is referenced in index EMP_LAST_NAME
-RDMS-F-RELFLDNOD, field LAST_NAME has not been deleted from relation
EMPLOYEES
SQL> --
SQL> -- You must drop the dependent index.
SQL> DROP INDEX EMP_LAST_NAME;
SQL> ALTER TABLE EMPLOYEES DROP LAST_NAME;
SQL> --
SQL> -- Undo the deletions.
SQL> --
SQL> ROLLBACK;
```

8.2.4 Modifying Columns

You can modify the data type, domain name, default value, or formatting clauses of columns by using the ALTER TABLE statement. You can also add and delete columns in a table. However, you cannot modify an existing column to COMPUTED BY or change from one COMPUTED BY clause to another. You must drop the column, and then add the modified column.

Keep in mind that when you delete a column and replace it with another column using the ALTER TABLE statement, you lose all data stored in the existing columns. Furthermore, you cannot specify the position of the new column in the table. New columns are added to the end of the table. You control display position for new columns only by specifying where you want them in a SELECT statement that accesses the table after the definition changes.

You can change the data type or domain name of a column without losing any data. Example 8–7 shows how to change the size of a column and how to add a column to a table.

Example 8–7 Modifying and Deleting Columns

```
SQL> SET TRANSACTION READ WRITE
cont>    RESERVING EMPLOYEES FOR EXCLUSIVE WRITE;
SQL> --
SQL> -- Assume you want to replace the two current address columns
SQL> -- with one.  You have decided that one (slightly larger)
SQL> -- address column is sufficient for data.
SQL> --
SQL> SHOW DOMAIN ADDRESS_DATA_1_DOM
ADDRESS_DATA_1_DOM              CHAR(25)
 Comment:       standard definition for street addresses
 Oracle Rdb default:
SQL> --
SQL> -- Make the domain slightly larger.
SQL> --
SQL> ALTER DOMAIN ADDRESS_DATA_1_DOM CHAR(30);
SQL> --
SQL> SHOW DOMAIN ADDRESS_DATA_1_DOM
ADDRESS_DATA_1_DOM              CHAR(30)
 Comment:       standard definition for street addresses
 Oracle Rdb default:
SQL> --
SQL> -- Delete the second address column and add a new column
SQL> -- called PHONE to store a telephone extension number.
SQL> --
SQL> ALTER TABLE EMPLOYEES DROP ADDRESS_DATA_2
cont>                      ADD PHONE CHAR(7);
SQL> --
SQL> -- You can delete the domain ADDRESS_DATA_2 at this point
SQL> -- because no other columns in the database refer to it.
SQL> --
SQL> DROP DOMAIN ADDRESS_DATA_2_DOM;
SQL> --
SQL> -- The SHOW statement shows the effects of the table modification; the
SQL> -- new column is added to the end of the list of columns in the table.
SQL> --
SQL> SHOW TABLE EMPLOYEES
Information for table EMPLOYEES

Comment on table EMPLOYEES:
personal information about each employee
```

(continued on next page)

Example 8–7 (Cont.) Modifying and Deleting Columns

```
Columns for table EMPLOYEES:
Column Name                     Data Type        Domain
-----------                     ---------        ------
EMPLOYEE_ID                     CHAR(5)          ID_DOM
 Primary Key constraint EMPLOYEES_PRIMARY_EMPLOYEE_ID
LAST_NAME                       CHAR(14)         LAST_NAME_DOM
FIRST_NAME                      CHAR(10)         FIRST_NAME_DOM
MIDDLE_INITIAL                  CHAR(1)          MIDDLE_INITIAL_DOM
ADDRESS_DATA_1                  CHAR(30)         ADDRESS_DATA_1_DOM
CITY                            CHAR(20)         CITY_DOM
STATE                           CHAR(2)          STATE_DOM
POSTAL_CODE                     CHAR(5)          POSTAL_CODE_DOM
SEX                             CHAR(1)          SEX_DOM
BIRTHDAY                        DATE             DATE_DOM
STATUS_CODE                     CHAR(1)          STATUS_CODE_DOM
PHONE                           CHAR(7)
   .
   .
   .
SQL> -- Show the data in the EMPLOYEES table.  Any data in the column
SQL> -- ADDRESS_DATA_2 has been lost.  The new column appears last.
SQL> --
SQL> SELECT * FROM EMPLOYEES LIMIT TO 1 ROW;
 EMPLOYEE_ID   LAST_NAME       FIRST_NAME   MIDDLE_INITIAL
   ADDRESS_DATA_1             CITY                  STATE   POSTAL_CODE   SEX
      BIRTHDAY      STATUS_CODE   PHONE
 00164         Toliver         Alvin        A
   146 Parnell Place          Chocorua              NH      03817         M
      28-Mar-1947   1             NULL

1 row selected
SQL> --
SQL> -- Change the display position for the columns.
SQL> --
SQL> SELECT EMPLOYEE_ID, LAST_NAME, ADDRESS_DATA_1, CITY, PHONE
cont> FROM EMPLOYEES LIMIT TO 1 ROWS;
 EMPLOYEE_ID   LAST_NAME        ADDRESS_DATA_1
   CITY                  PHONE
 00164         Toliver          146 Parnell Place
   Chocorua              NULL

1 rows selected
SQL> ROLLBACK;
```

The following sections provide more information about modifying columns in a table.

8.2.5 Modifying Column Data Types

Be careful if you modify columns to change their data types or size. You should choose data types compatible with data already stored in those columns and column sizes large enough for existing values. In addition, if columns are based on domain definitions, you should add or modify domain definitions to make the change in data type or size. If you are creating a new domain definition to support your data type change, use the ALTER TABLE statement to make the column definition refer to the new domain.

When you change data types, SQL automatically converts data to the new storage format. Remember that if you change data type or size for a column, you may affect existing applications that use the column. As a result, you may need to adjust program parameters explicitly declared in source files, or you may need to edit, compile, and link modules that are part of the application, or both.

Example 8–8 illustrates data type changes.

Example 8–8 Changing Data Types in a Table

```
SQL> -- Assume, for now, that these columns are not based on domains and that
SQL> -- it is appropriate to make data type changes directly in the column
SQL> -- definition rather than in the domain definition.  Changing columns
SQL> -- from a text to a numeric data type may create problems.
SQL> --
SQL> SET TRANSACTION READ WRITE
cont> RESERVING EMPLOYEES FOR EXCLUSIVE WRITE;
SQL> --
SQL> -- ADDRESS_DATA_1 contains nondigit characters whose meaning is lost if
SQL> -- they are converted to a number that does not fit in the numeric
SQL> -- column you define.  SQL displays a warning message to alert you
SQL> -- to a potential problem.
SQL> --
SQL> ALTER TABLE EMPLOYEES ALTER COLUMN ADDRESS_DATA_1 INTEGER;
%SQL-W-INC_DAT_TYP,  Altering column ADDRESS_DATA_1 to an incompatible
datatype may cause data loss
SQL> --
SQL> -- The errors returned on the SELECT statement verify that the data type
SQL> -- conversion is inappropriate.
SQL> --
SQL> SELECT ADDRESS_DATA_1 FROM EMPLOYEES;
%RDB-E-ARITH_EXCEPT, truncation of a numeric value at runtime
-OTS-F-INPCONERR, input conversion error
SQL> --
```

(continued on next page)

Example 8–8 (Cont.) Changing Data Types in a Table

```
SQL> -- To avoid corrupting your data, enter a ROLLBACK statement.
SQL> --
SQL> ROLLBACK;
SQL> --
SQL> -- Text columns that contain only digits can be converted to numeric
SQL> -- columns, but results can be unexpected.  A common problem occurs
SQL> -- with leading zeros that have significance to the user.  Again,
SQL> -- SQL returns a warning on the ALTER TABLE statement to alert you
SQL> -- to a potential problem.  In this case, no errors are returned on
SQL> -- the SELECT statement because the conversion did not change
SQL> -- numeric values.
SQL> --
SQL> SET TRANSACTION READ WRITE
cont> RESERVING EMPLOYEES FOR EXCLUSIVE WRITE;
SQL> --
SQL> SELECT POSTAL_CODE FROM EMPLOYEES LIMIT TO 3 ROWS;
 POSTAL_CODE
 03817
 03817
 03301
3 rows selected
SQL> ALTER TABLE EMPLOYEES ALTER COLUMN POSTAL_CODE QUADWORD;
%SQL-W-INC_DAT_TYP,  Altering column POSTAL_CODE to an incompatible datatype
may cause data loss
SQL> --
SQL> SELECT POSTAL_CODE FROM EMPLOYEES LIMIT TO 3 ROWS;
 POSTAL_CODE
 3817
 3817
 3301
3 rows selected
SQL> --
SQL> ROLLBACK;
SQL> --
SQL> -- Reduce the size of the STATUS_TYPE column using the ALTER COLUMN
SQL> -- clause.
SQL> --
SQL> ALTER TABLE COLLEGES
cont>      ALTER COLUMN COLLEGE_NAME CHAR(20);
%SQL-W-CHR_TOO_SHO,  Character length of column COLLEGE_NAME is too short
```

(continued on next page)

Example 8–8 (Cont.) Changing Data Types in a Table

```
SQL> --
SQL> -- Data in the column is now too long for the column.  SQL truncates
SQL> -- the data.
SQL> SELECT COLLEGE_NAME FROM COLLEGES LIMIT TO 1 ROWS;
 COLLEGE_NAME
 American Univer
1 row selected
```

The ALTER TABLE statement does not immediately change the data type of data that is already stored. The storage format of data for any particular row is not changed until you update that row; however, any new rows that you insert are stored in the new storage format. When you display data using the SELECT statement, the SELECT statement converts the data into the new data type for display purposes, but it does not change the format of the stored data.

8.2.6 Modifying Columns That Include Date-Time Data Types

You can modify the data types of columns and domains that have the date-time data type. However, you should be aware of the following points:

- You can modify DATE VMS to CHAR, VARCHAR, DATE ANSI, TIME, or TIMESTAMP.
- You can modify DATE ANSI to CHAR, VARCHAR, TIMESTAMP, or DATE VMS.
- You can modify TIME to CHAR, VARCHAR, TIMESTAMP, or DATE VMS. Modifying TIME to TIMESTAMP adds the current date to the time. Modifying TIME to DATE VMS adds 17-NOV-1858 to the time.
- You can modify TIMESTAMP to CHAR, VARCHAR, TIME, or DATE.
- You can modify YEAR-MONTH intervals to TEXT or another form of YEAR TO MONTH interval.
- Attempting to modify a YEAR-MONTH interval to a DAY-TIME interval causes the following error message:

  ```
  SQL> ALTER DOMAIN STANDARD_INTERVAL_YM INTERVAL DAY(8) DEFAULT NULL;
  %RDB-E-NO_META_UPDATE, metadata update failed
  -RDB-E-CONVERT_ERROR, invalid or unsupported data conversion
  -RDMS-F-INV_DATE_CHG, invalid field datatype change to/from datetime
  ```

- You can modify DAY-TIME intervals to TEXT to a form of DAY-TIME interval containing one or more of the fields DAY, HOUR, MINUTE, or SECOND.
- When you modify the data type of a domain or column that is defined with a default value, you must specify a new default value if the current one does not match the new data type of the column or domain. For example, the following are valid alter operations:

```
SQL> CREATE DOMAIN STANDARD_INTERVAL_YM INTERVAL YEAR(4) TO MONTH
cont> DEFAULT INTERVAL '0000-00' YEAR(4) TO MONTH;
SQL>
SQL> CREATE DOMAIN STANDARD_INTERVAL_DT INTERVAL HOUR(6) TO SECOND(0)
cont> DEFAULT null;
SQL>
SQL> ALTER DOMAIN STANDARD_INTERVAL_YM INTERVAL MONTH(5)
cont> DEFAULT INTERVAL '00000' MONTH(5);
SQL>
SQL> ALTER DOMAIN STANDARD_INTERVAL_DT INTERVAL DAY(5) TO MINUTE
cont> DEFAULT INTERVAL '1:0:0' DAY(5) TO MINUTE;
```

8.2.7 Adding, Modifying, and Dropping Default Values from a Column

You can add a default value to an existing column, modify the default value of an existing column, or drop the default value. However, doing so has no effect on the values stored in existing rows, as Example 8–9 demonstrates.

Example 8–9 Modifying the Default Value of an Existing Column

```
SQL> -- Assume that the EMPLOYEES table uses the default value "?" for
SQL> -- the SEX column and that a row in the EMPLOYEES table uses that value.
SQL> --
SQL> SELECT EMPLOYEE_ID, FIRST_NAME, LAST_NAME, SEX
cont> FROM EMPLOYEES WHERE SEX = '?';
 EMPLOYEE_ID   FIRST_NAME   LAST_NAME        SEX
 00799         Green        Pat              ?
1 row selected
SQL> --
SQL> -- Now, change the default value of the SEX column.
SQL> --
SQL> ALTER TABLE EMPLOYEES
cont>  ALTER SEX CHAR(7)
cont>        SET DEFAULT 'Unknown';
SQL> --
```

(continued on next page)

Example 8–9 (Cont.) Modifying the Default Value of an Existing Column

```
SQL> -- The value stored in the SEX column for Pat Green did not change.
SQL> SELECT EMPLOYEE_ID, FIRST_NAME, LAST_NAME, SEX
cont> FROM EMPLOYEES WHERE EMPLOYEE_ID = '00799';
 EMPLOYEE_ID   FIRST_NAME   LAST_NAME        SEX
 00799         Green        Pat              ?
1 row selected
SQL>
```

When you add a column to a table and specify a default value for the column, SQL stores the default value in the newly added column of all the previously stored rows. Likewise, if the newly added column is based upon a domain that specifies a default value, SQL stores the default value in the column of all the previously stored rows.

Example 8–10 shows that when you add a column that specifies a default value, SQL stores the default value in the column of all the previously stored rows.

Example 8–10 Adding Columns with Default Values to Tables

```
SQL> -- Add the column PHONE and specify a default value.
SQL> --
SQL> ALTER TABLE EMPLOYEES
cont>     ADD PHONE CHAR(7) SET DEFAULT 'Unknown';
SQL> --
SQL> -- The result table shows that the rows contain the default value
SQL> -- of the PHONE column.
SQL> --
SQL> SELECT LAST_NAME, PHONE FROM EMPLOYEES LIMIT TO 3 ROWS;
 LAST_NAME        PHONE
 Toliver          Unknown
 Smith            Unknown
 Dietrich         Unknown
3 rows selected
SQL> --
SQL> ROLLBACK;
```

Because SQL updates data when you add a column with a default value other than NULL, the ALTER TABLE statement may take some time to complete if the table contains many rows. (If you specify a default value of NULL, SQL does not modify the data because SQL automatically returns a null value in columns that have no actual value stored in them.) If you want to add more than one column with default values, add them in one ALTER TABLE

statement. When you do so, SQL scans the table data once instead of many times.

Be aware that adding data in the form of a column with a default value may result in fragmented records. For information about locating and correcting record fragmentation, see the *Oracle Rdb7 Guide to Database Performance and Tuning* and the *Oracle Rdb7 Guide to Database Maintenance*.

If you do not want the default value to be propagated to existing rows, you can take the two-step approach shown in Example 8–11. First, you add the column without specifying a default value. Then, you modify the column to specify a default value. SQL does not change the data in existing rows. However, any rows that are inserted after the change contain the default value if you do not specify a value for that column.

Example 8–11 Adding Columns Without Propagating Default Values to Previously Stored Rows

```
SQL> -- Add the column PHONE.  Do not specify a default value.
SQL> --
SQL> ALTER TABLE EMPLOYEES
cont>    ADD PHONE CHAR(7);
SQL> SELECT LAST_NAME, PHONE FROM EMPLOYEES LIMIT TO 3 ROWS;
 LAST_NAME        PHONE
 Toliver          NULL
 Smith            NULL
 Dietrich         NULL
3 rows selected
SQL> --
SQL> -- Modify the column, adding a default value.
SQL> --
SQL> ALTER TABLE EMPLOYEES
cont>    ALTER COLUMN PHONE
cont>      DEFAULT 'Unknown';
SQL> --
SQL> --  SQL does not store the default value in existing rows.
SQL> --
SQL> SELECT LAST_NAME, PHONE FROM EMPLOYEES LIMIT TO 3 ROWS;
 LAST_NAME        PHONE
 Toliver          NULL
 Smith            NULL
 Dietrich         NULL
3 rows selected
```

(continued on next page)

Example 8–11 (Cont.) Adding Columns Without Propagating Default Values to Previously Stored Rows

```
SQL> --
SQL> -- Insert a new row and do not specify a value for the PHONE column.
SQL> --
SQL> INSERT INTO EMPLOYEES
cont>      (EMPLOYEE_ID, LAST_NAME, FIRST_NAME, STATUS_CODE)
cont> VALUES ('00100', 'Jones', 'Helen', '1');
SQL> --
SQL> -- The new row uses the default value.
SQL> --
SQL> SELECT LAST_NAME, PHONE FROM EMPLOYEES WHERE LAST_NAME='Jones';
 LAST_NAME        PHONE
 Jones            Unknown
1 row selected
SQL>
```

You can drop the default value from a column by using the DROP DEFAULT clause, as shown in Example 8–12.

Example 8–12 Dropping the Default Value from a Column

```
SQL> -- Drop the default value for the column PHONE.
SQL> --
SQL> ALTER TABLE EMPLOYEES
cont>  ALTER COLUMN PHONE
cont>        DROP DEFAULT;
```

Remember that the default value for a column is *not* the same as the missing value that you can specify using the RDO interface.

8.2.8 Modifying the Name of a Table or the Name or Position of a Column

To change the names of tables or columns or to change the position of a column in a table, you create a new table, transfer data (if any) from the old table to the new table, and then delete the old table.

You can easily control column position using SELECT statements or view definitions. Remember this when you add columns to a table with data stored in it. For an established database, adjustments in names or position are often made using view definitions. Changing names of tables and columns is rarely a trivial task in an established database. For example, note the steps you take to change the table name EMPLOYEES to PERSONAL_DATA in the personnel database:

1. Use the CREATE TABLE statement to define the table again with PERSONAL_DATA as the table name.
2. Insert data from the EMPLOYEES table in the new PERSONAL_DATA table.

   The *Oracle Rdb7 Introduction to SQL* contains examples of inserting rows from one table to another table.
3. Delete and create again any views based on EMPLOYEES so that the views specify PERSONAL_DATA instead.
4. Delete indexes for EMPLOYEES and create them again as indexes for PERSONAL_DATA.
5. Delete the EMPLOYEES table using the CASCADE keyword.

   If other database definitions (indexes, views, constraints, storage maps) refer to the EMPLOYEES table, those other definitions are also deleted when you delete the table. In addition, if a module (stored procedure) refers to the table, Oracle Rdb marks the module as invalid.
6. Create again any other database definitions that refer to EMPLOYEES so that they refer to PERSONAL_DATA.
7. Modify any applications that refer to EMPLOYEES so that they refer to PERSONAL_DATA instead.

   For precompiled host language programs and SQL modules, this means compiling and linking source files again.

As you can see, changing names of tables and columns is usually not worth the time and effort, except when you first create a database. At that time, you may have no data stored, no application dependencies, and few cross-definition dependencies. It is also the only time when you do not have to manage changes to the database in a way that does not seriously disrupt the activities and needs of database users.

If users complain that table and column names in an established database are inappropriate or misleading, one solution is to create a view with more helpful names and ask users to access the view.

8.2.9 Modifying and Deleting Tables in Multischema Databases

To modify or delete a table in a multischema database, you must qualify the name of the table with the names of the catalog and schema that contain it.

```
SQL> DROP TABLE ADMINISTRATION.ACCOUNTING.DEPARTMENTS;
```

For more information on qualifying names of elements in multischema databases, see Section 5.5.

8.3 Modifying and Deleting Constraints

To modify a constraint, you must drop the constraint and create it again. To drop a constraint or create a new constraint for an existing table, you use the ALTER TABLE statement.

If you modify a column, for example to change the data type of a column, Oracle Rdb checks any constraints that refer to that column. If the alter operation violates a constraint, you must drop the constraint before you modify the column.

Example 8–13 shows how to modify constraints and how to delete a constraint.

Example 8–13 Modifying and Deleting Constraints

```
SQL> -- Change the name of the STATUS_NAME_VALUES constraint and its
SQL> -- definition by deleting the constraint and adding it again with a
SQL> -- new name.  When you refer to definitions of table constraints in
SQL> -- the ALTER TABLE statement, the keyword CONSTRAINT is required
SQL> -- after the keywords DROP or ADD.
SQL> --
SQL> ALTER TABLE WORK_STATUS
cont>      DROP CONSTRAINT STATUS_NAME_VALUES;
SQL> --
SQL> ALTER TABLE WORK_STATUS ADD CONSTRAINT
cont>       CONSTRAINT STATUS_NAME_VAL
cont>       CHECK  ( STATUS_NAME IN ('ACTIVE', 'INACTIVE') )
cont>       DEFERRABLE;
SQL> SHOW TABLE WORK_STATUS
   .
   .
   .
Table constraints for WORK_STATUS:
STATUS_NAME_VAL
 Check constraint
 Table constraint for WORK_STATUS
 Evaluated on COMMIT
 Source:
        CHECK          ( STATUS_NAME IN ('ACTIVE', 'INACTIVE') )
   .
   .
   .
SQL> ROLLBACK;
```

When you add or modify constraints that refer to other tables, you should include the other tables in the RESERVING clause of the statement that specifies the transaction. However, if you do not explicitly reserve tables

referred to by constraints (and triggers), SQL automatically locks those tables at the time the constraint refers to them.

If you want to change a reference to another table, first you must delete the constraint, then create it again using the ALTER COLUMN clause of the ALTER TABLE STATEMENT, as shown in Example 8–14.

Example 8–14 Modifying Constraints That Refer to Other Tables

```
SQL> -- Delete the constraint.
SQL> --
SQL> ALTER TABLE SALARY_HISTORY
cont>      DROP CONSTRAINT SH_EMPLOYEE_ID_IN_EMP_REF;
SQL> --
SQL> -- Add the constraint.
SQL> --
SQL> ALTER TABLE SALARY_HISTORY
cont>      ALTER COLUMN EMPLOYEE_ID
cont>         REFERENCES EMPLOYEES (EMPLOYEE_ID)
cont>         CONSTRAINT SH_EMP_ID_IN_EMP_REF;
```

Although Oracle Rdb generally evaluates constraints at commit or execution time, if you modify a constraint in a table that contains data, Oracle Rdb evaluates the constraint at definition time to make sure the table does not contain any rows that violate the constraint. See the *Oracle Rdb7 Guide to SQL Programming* for more information about constraint violation.

The DROP CONSTRAINT clause of the ALTER TABLE statement removes either a column or table constraint from the table definition. The ADD CONSTRAINT clause creates a new table constraint.

For more information about constraints and for additional examples using the CREATE TABLE, ALTER TABLE, and DROP TABLE statements, see the *Oracle Rdb7 SQL Reference Manual*.

8.4 Modifying and Deleting Triggers

To modify a trigger definition, first you must delete the trigger definition, then create it again. To delete a trigger definition, use the DROP TRIGGER statement.

When you use the DROP TRIGGER statement:

- You must execute the statement in a read/write transaction. If you issue this statement when there is no active transaction, SQL starts a transaction with characteristics specified in the most recent DECLARE TRANSACTION statement.

- To delete a trigger, you must have DELETE access to the table for which the trigger is defined.
- Other users can be attached to the database when you issue the DROP TRIGGER statement.
- You cannot execute the DROP TRIGGER statement when the RDB$SYSTEM storage area is set to read-only. You must first set RDB$SYSTEM to read/write. See the description of the ALTER DATABASE statement in the *Oracle Rdb7 SQL Reference Manual* for more information on the RDB$SYSTEM storage area.

Example 8–15 shows how to delete a trigger definition, then create it again.

Example 8–15 Modifying and Deleting Triggers

```
SQL> -- Delete the trigger definition:
SQL> --
SQL> DROP TRIGGER EMPLOYEE_ID_CASCADE_DELETE;
SQL> --
SQL> -- Create the trigger again with a new definition.
SQL> -- If an employee is terminated, remove all associated rows from the
SQL> -- DEGREES, JOB_HISTORY, and SALARY_HISTORY tables.
SQL> --
SQL> CREATE TRIGGER EMPLOYEE_ID_CASCADE_DELETE
cont>   BEFORE DELETE ON EMPLOYEES
cont>   (DELETE FROM DEGREES D
cont>     WHERE D.EMPLOYEE_ID = EMPLOYEES.EMPLOYEE_ID)
cont>      FOR EACH ROW
cont>   (DELETE FROM JOB_HISTORY JH
cont>      WHERE JH.EMPLOYEE_ID = EMPLOYEES.EMPLOYEE_ID)
cont>      FOR EACH ROW
cont>   (DELETE FROM SALARY_HISTORY SH
cont>      WHERE SH.EMPLOYEE_ID = EMPLOYEES.EMPLOYEE_ID)
cont>      FOR EACH ROW
cont> --
cont> -- If an employee is terminated and that employee is the manager
cont> -- of a department, set the MANAGER_ID to 'Open' for that department.
cont> --
cont>   (UPDATE DEPARTMENTS D  SET D.MANAGER_ID = 'Open'
cont>      WHERE D.MANAGER_ID = EMPLOYEES.EMPLOYEE_ID)
cont>      FOR EACH ROW;
```

8.5 Deleting Views

You may want to delete a view from the database for the following reasons:

- The view is no longer needed by users.
- Users have asked for changes to the view, perhaps adding a check option clause, or an additional column or column support clause, such as EDIT STRING or LIMIT TO.

  You cannot modify views. To create the view again with the same name, you must first delete it.
- The view refers to a table column whose data type you want to modify.

  In this case, you must delete the view, change the table column, and then create the view again.
- You want to delete a table column to which a view refers.

  In this case, you must delete the view before deleting the column from the table.

If other view definitions depend upon the view that you want to delete, you must delete those definitions first, as shown in the following example:

```
SQL> Delete the view.
SQL> --
SQL> DROP VIEW CURRENT_JOB;
%RDB-E-NO_META_UPDATE, metadata update failed
-RDMS-F-VIEWINVIEW, view CURRENT_JOB is referenced by view CURRENT_INFO
-RDMS-F-VIEWNOTDEL, view CURRENT_JOB has not been deleted
SQL> --
SQL> -- You cannot delete the view because other views refer to it.
SQL> -- Delete the views that refer to CURRENT_JOB.
SQL> --
SQL> DROP VIEW CURRENT_INFO;
SQL> --
SQL> -- Now, drop the CURRENT_JOB view.
SQL> --
SQL> DROP VIEW CURRENT_JOB;
```

If you specify the CASCADE keyword, SQL automatically deletes any views that refer to the deleted view. In the interactive environment, SQL displays informational messages that this is being done. Then, you can choose to roll back, display, and perhaps delete and create those views again so that they no longer depend on the element you want to delete. This may be desirable when a view is based on several views or tables and you are deleting only one of them. The informational messages for the DROP TABLE and DROP VIEW statements provide an audit of element dependencies that you may find useful. You do not receive these messages when you use SQL in a program.

```
SQL> -- Delete the view and all views that refer to it.
SQL> --
SQL> DROP VIEW CURRENT_JOB CASCADE;
View CURRENT_INFO is also being dropped.
SQL>
```

If, when you created the CURRENT_INFO view, you stored it in the repository as well as in the database file, the DROP statement in the preceding examples would not delete the view in the repository. Chapter 10 includes more information about how you correct this error.

For more information about situations when database users are affected by view deletion, see the *Oracle Rdb7 SQL Reference Manual*.

To delete a view in a multischema database, you must qualify the name of the view with the names of the catalog and schema that contain it.

```
SQL> DROP VIEW  ADMINISTRATION.PERSONNEL.REVIEW_DATE;
```

For more information on qualifying names of elements in multischema databases, see Section 5.5.

8.6 Deleting Schemas in Multischema Databases

If you have a multischema database, you can delete one or all schemas in the database using the DROP SCHEMA statement. If the schema contains other elements, you cannot delete it until you drop all the elements it contains.

You can use the CASCADE keyword to delete a schema, all the elements it contains, and any elements that depend on the schema. In fact, the deletion continues to cascade until SQL finds no further dependencies. The following example illustrates dropping the schema ACCOUNTING:

```
SQL> DROP SCHEMA ADMINISTRATION.ACCOUNTING
cont>     CASCADE;
Table ADMINISTRATION.ACCOUNTING.DAILY_HOURS is also being dropped.
Table ADMINISTRATION.ACCOUNTING.DEPARTMENTS is also being dropped.
View ADMINISTRATION.PERSONNEL.CURRENT_INFO is also being dropped.
Constraint ADMINISTRATION.ACCOUNTING.DEPARTMENTS_FOREIGN1 is also being
dropped.
Constraint ADMINISTRATION.ACCOUNTING.DEPARTMENTS_NOT_NULL1 is also being
dropped.
Constraint ADMINISTRATION.ACCOUNTING.DEPARTMENTS_UNIQUE1 is also being
dropped.
Trigger ADMINISTRATION.PERSONNEL.EMPLOYEE_ID_CASCADE_DELETE is also being
dropped.
Table ADMINISTRATION.ACCOUNTING.PAYROLL is also being dropped.
```

```
Constraint ADMINISTRATION.ACCOUNTING.PAYROLL_PRIMARY_JOB_CODE is also being
dropped.
   .
   .
   .
Domain ADMINISTRATION.ACCOUNTING.BUDGET is also being dropped.
Domain ADMINISTRATION.ACCOUNTING.CODE is also being dropped.
SQL>
```

Be careful when you drop a schema using the CASCADE keyword—you may drop elements in other schemas. For example, if a view in another schema depends upon a table in the schema you are dropping, SQL drops the view. However, if a cascading delete drops a domain on which tables in other schemas depend, SQL does not drop those tables. Instead, SQL modifies the table definitions by explicitly declaring the data type and other characteristics in place of the domain definition.

You cannot drop the system schema RDB$SCHEMA.

8.7 Deleting Catalogs in Multischema Databases

If you have a multischema database, you can delete one or all catalogs in the database using the DROP CATALOG statement. If the catalog contains other elements, you cannot delete it until you drop all the elements it contains.

You can use the CASCADE keyword to delete a catalog, all the elements it contains, and any other elements that depend on elements in the catalog. In fact, the deletion continues to cascade until SQL finds no further dependencies. The following example shows what happens when you delete the catalog ADMINISTRATION:

```
SQL> DROP CATALOG ADMINISTRATION
cont>     CASCADE;
Table ADMINISTRATION.ACCOUNTING.DAILY_HOURS is also being dropped.
Table ADMINISTRATION.ACCOUNTING.DEPARTMENTS is also being dropped.
View ADMINISTRATION.PERSONNEL.CURRENT_INFO is also being dropped.
Constraint ADMINISTRATION.ACCOUNTING.DEPARTMENTS_FOREIGN1 is also being
dropped.
Constraint ADMINISTRATION.ACCOUNTING.DEPARTMENTS_NOT_NULL1 is also being
dropped.
   .
   .
   .
Domain ADMINISTRATION.PERSONNEL.SALARY is also being dropped.
Domain ADMINISTRATION.PERSONNEL.STATE_CODE is also being dropped.
Domain ADMINISTRATION.PERSONNEL.STATUS_CODE is also being dropped.
Domain ADMINISTRATION.RECRUITING.ID is also being dropped.
SQL>
```

Be careful when you drop a catalog using the CASCADE keyword—you may drop elements in other catalogs. For example, if a view in another catalog depends upon a table in the catalog you are dropping, SQL drops the view. However, if a cascading delete drops a domain on which tables in other catalogs depend, SQL does not drop those tables. Instead, SQL modifies the table definitions by explicitly declaring the data type and other characteristics in place of the domain definition.

You cannot drop the system catalog RDB$CATALOG.

9
Defining Database Protection

Oracle Rdb provides a security mechanism to protect your database from unauthorized users. Unauthorized users can harm your database or your business through accidental or intentional acts. Without sufficient security protections on your database, any system user might gain access to confidential information, or alter the contents of the database in an unacceptable manner. This chapter describes how to use access privileges and protection to secure your database.

Oracle Rdb provides the following two methods to grant or deny privileges to particular users:

- SQL GRANT and REVOKE statements and access privilege sets to control privileges for database operations such as creating tables, updating, inserting, and deleting records, and reading records

  Section 9.2 discusses access privilege sets and Section 9.3 describes how to use the GRANT and REVOKE statements to control access to database definition and manipulation operations.

- RMU privileges to control privileges for database maintenance operations performed with RMU commands

  Section 9.5 discusses how to use the RMU privileges.

9.1 Planning for Database Security

In planning for database security, it is useful to think in terms of the **reference monitor concept**. This concept depicts a computer system (or database system) in terms of subjects, objects, an authorization database, a security audit journal, and a reference monitor mechanism.

A **subject** is an active entity that gains access to information on behalf of people. In Oracle Rdb, a subject is an attachment to the database. Thus, an Oracle Rdb subject could be either an interactive user or an application program.

An **object** is a passive repository of information to be protected. In Oracle Rdb, objects include databases, catalogs, schemas, tables, views, or columns.

You define Oracle Rdb security requirements by determining which subjects (acting on behalf of users) can have what kinds of access to which objects (that contain information).

An audit journal maintains a record of access attempts, successful or not, as required by the authorization database. (For information about the audit journal for Oracle Rdb, see the *Oracle Rdb7 Guide to Database Maintenance*). The reference monitor mechanism enforces security rules by authorizing the creation of subjects, granting subjects access to objects according to the requirements of the database, and recording events as necessary in the audit journal.

To make proper use of the Oracle Rdb reference monitor mechanism, you must first determine what degree of protection your database objects require. Some objects require more protection than others. For instance, you might want to prevent unauthorized read access to a table or column containing salary information, but merely prevent unauthorized changes to a table that simply contains information about departments and their managers.

You should grant to users only those privileges necessary to perform an operation. For example, if a user needs to query the DEPARTMENTS table in the database but not update it, and the same user needs to update the EMPLOYEES table, that user needs the SELECT and UPDATE privileges for the database, the SELECT privilege for both tables, and the UPDATE privilege for the EMPLOYEES table.

9.2 Understanding Privilege Checking for SQL Statements

You use the SQL GRANT and REVOKE statements to control data definition and data manipulation operations on databases. The SQL GRANT and REVOKE statements use access privilege sets to define which users can access the database object and what operations they can perform.

An **access privilege set** is associated with each database object (such as a database, catalog, schema, table, view, or column). Each entry in an access privilege set consists of an identifier and a list of privileges assigned to that identifier.

Access privilege set is a generic term that relates to two styles of privileges (see Figure 9–1):

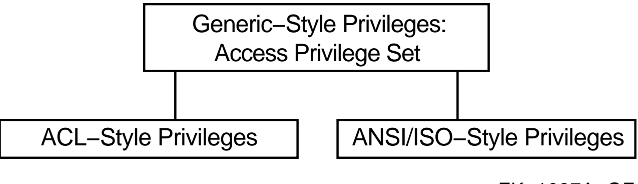
- Access control list- (ACL) style privileges

ACL-style privileges use access control lists made up of access control entries (ACEs). In ACL-style privileges, the order of the ACEs is critical to the privileges granted to any one user. When you first create an ACL, it is usually easier to build a command file so you can edit your ACEs and put them in the most useful order.

- American National Standards Institute/International Standards Organization- (ANSI/ISO) style privileges

  ANSI/ISO-style privileges use sets made up of ACEs or user access sets. The order of these user access sets within the access privilege set is not important.

Figure 9–1 Relationship Between Generic-Style Privileges and ACL- and ANSI/ISO-Style Privileges



Section 9.2.2 describes the ACL-style and ANSI/ISO-style privileges in more detail.

By default, when you create a new database, Oracle Rdb grants all privileges to the user identifier of the creator of the database, and no privileges to the public on the database, on tables, and on views. (Although a column is an Oracle Rdb object, it neither needs nor receives a default access privilege set.) You must explicitly issue the GRANT statement to give privileges to other users to let them access your databases.

You use the SHOW PROTECTION statement to display the privileges for a database. The following example shows the privileges Oracle Rdb grants when you create a database:

```
SQL> CREATE DATABASE ALIAS test1 FILENAME test1;
SQL> SHOW PROTECTION ON DATABASE test1;
Protection on Alias test1
  (IDENTIFIER=[grp2,heleng],ACCESS=SELECT+INSERT+UPDATE+DELETE+SHOW+CREATE+
    ALTER+DROP+DBCTRL+OPERATOR+DBADM+SECURITY+DISTRIBTRAN)
  (IDENTIFIER=[*,*],ACCESS=NONE)
```

In the previous example, the identifier [grp2,heleng] is the user identifier of the creator of the database. The identifier [*,*] is the identifier for the public. Because the CREATE DATABASE statement does not specify PROTECTION IS ANSI clause, Oracle Rdb creates the database using the ACL style.

The access privilege sets apply specifically to data definition and data manipulation operations. The access privilege sets are independent of the security defined by RMU privileges or the repository or the operating system. However, privileged users on Digital UNIX and on OpenVMS can override the access privilege sets. Figure 9–2 shows Digital UNIX users and some OpenVMS privileges that can override the access privilege sets.

Figure 9–2 Privileges to Access Oracle Rdb Databases

For more information on how operating system privileges and privileged accounts override Oracle Rdb security, see Section 9.4.1.

The access privilege sets maintained by the repository apply only to the copies of the Oracle Rdb definitions stored in the repository. You can use the repository protection mechanism to secure copies of the shareable field and record definitions in the repository from unauthorized access. However, you *cannot* use the repository interface to change protection for Oracle Rdb database objects. For more information, refer to Section 9.7, Chapter 10, and the Oracle CDD/Repository documentation.

9.2.1 Introducing Access Control Entries (ACEs)

An **Access Control Entry (ACE)** consists of a user identifier and the Oracle Rdb access privileges assigned to the identification code.

To create ACEs for a database and its objects, you must have the DBCTRL privilege for those objects. When you create a database, Oracle Rdb automatically creates an ACE that grants you DBCTRL privileges for that database. When you create a table or view, Oracle Rdb automatically grants you the DBCTRL privilege for that table or view. The table or view ACE is independent of the privileges granted by the database ACE.

The person who creates a database, table, view, or column is its **owner**, and all users (represented by [*,*] for ACL-style and PUBLIC for ANSI/ISO-style) constitute the **public**.

For a particular user, Oracle Rdb allows a data manipulation access privilege to a table only if that privilege is granted for *both* the database and the table. For example, a user has INSERT privilege for the EMPLOYEES table only if that user has INSERT privilege for both the mf_personnel database and the EMPLOYEES table.

Thus, the database ACE for each user or group of users should grant all the data manipulation privileges they might need for any table. Then, you can revoke privileges as necessary from users at the table level. However, to restrict users from modifying certain columns in tables, you must *deny* them table-level privileges and *grant* them column-level privileges.

You probably want to allow more access than the default protection provides for the database, for some or all tables, and for some or all columns within some or all tables. If you wish to grant users the privilege to define indexes or views for a table, you must grant them CREATE privilege for that particular table. You do not need to grant users CREATE privilege for the database itself.

See Table 9–1 in Section 9.2.3 for additional information about specific operations that can be controlled by database or table privileges.

9.2.2 Introducing ACL-Style and ANSI/ISO-Style Privileges

Both ACL-style and ANSI/ISO-style privileges use ACEs to specify a user identifier and a list of privileges assigned to that user identifier.

The main differences between ACL-style and ANSI/ISO-style privileges are as follows:

- Different types of identifiers are supported

  ACL-Style identifiers support three types of identifiers:

  - User identifiers

    A user identifier is a code that uniquely identifies each user on the system.

    Digital UNIX

    On Digital UNIX, the user identifier consists of the name of the group and the user identifier (UID), represented in alphanumeric form. Users can belong to more than one group on Digital UNIX. Therefore, if you add a new entry and there is already an entry for that user, but a different group, Oracle Rdb adds the new entry. For example, you could have the following user identifiers represented in an ACE:

    ```
    [grp1,ingrid]
    [doc,ingrid]
    ```
    ♦

    OpenVMS VAX OpenVMS Alpha

    On OpenVMS, the user identifier consists of the user identification code (UIC). It can be represented in numeric or alphanumeric format. ♦

    When you specify a user identifier, you can specify the group name and the user name, omit the group name, or use wildcards. For example, the following are valid user identifiers:

    ```
    heleng
    [doc,heleng]
    [*,heleng]
    [doc,*]
    ```

  OpenVMS VAX OpenVMS Alpha

  - General rights identifiers

    General identifiers are those defined in the OpenVMS system rights database. For example, all application programmers may be assigned the general identifier PROGRAMMERS. When you use the PROGRAMMERS identifier in an ACE, all application programmers receive the same privileges. ♦

OpenVMS VAX OpenVMS Alpha

– System-defined identifiers

  OpenVMS defines the identifiers based on the type of login a process executes. For example, users who log in interactively are granted the INTERACTIVE identifier. ♦

ANSI/ISO-style privileges support only those identifiers that translate to a user identifier. They do not support wildcards, except for the keyword PUBLIC, which allows access to anyone.

The following are valid ANSI/ISO-style identifiers:

```
heleng
[doc,heleng]
```

- The privilege mask is determined differently

  With ACL-style privileges, the access control entries (ACEs) are in a specific order within the ACL and only the privileges from the first ACE that matches any particular user apply. For example:

```
[rhonda]=UPDATE+INSERT
[ingrid]=CREATE+ALTER+DROP+DBCTRL+UPDATE+INSERT+DELETE
[*,*]=SELECT
```

  Becaue only the first set of privileges apply, the effective privileges for [rhonda] are UPDATE and INSERT.

  With ANSI/ISO-style privileges, to determine the applicable set of privileges (the privilege mask), you combine all of the privileges that apply to a user and all of the privileges that apply to the public. For example:

```
[rhonda]=UPDATE+INSERT
[ingrid]=CREATE+ALTER+DROP+DBCTRL+UPDATE+INSERT+DELETE
PUBLIC=SELECT
```

  In the preceding example, the effective privileges for [rhonda] are UPDATE, INSERT, and SELECT. The privileges specified for PUBLIC or [*,*] apply to everyone.

- GRANT OPTION is available only with ANSI/ISO-style

  ANSI/ISO-style allows you to give specific users GRANT OPTION on specific commands for specific objects. If a user has the SELECT privilege WITH GRANT OPTION on the database, the user can grant SELECT privilege on the database to anyone else. Granting and revoking of privileges is tracked in ANSI/ISO-style protection, allowing you to perform a cascading revoke of privileges back to the originator of the privilege.

- Combining more than one identifier is allowed only with ACL-style

ACL-style allows you to specify more than one identifier, by separating them with plus signs (+). However, the six system-defined identifiers are mutually exclusive. You can combine them with other identifiers (UICs and general identifiers).

The following is a multiple identifier that grants a set of privileges to user jones when jones uses Oracle Rdb applications interactively:

```
[grp2,jones]+INTERACTIVE
```

The advantages of ACL-style protection are:

- You can use general identifiers and system identifiers, as well as user identifiers.
- You can use wildcards in identifiers.
- You can combine identifiers.
- You can use the POSITION and AFTER keywords to control the placement of ACEs within an ACL. You can use the keywords to place group or wildcard identifiers *after* more restrictive identifiers have granted extra privileges to a particular user.

The advantages of ANSI/ISO-style protection are:

- Compatibility with the ANSI/ISO standard makes your application more portable.
- You have greater flexibility in granting certain privileges and in allowing others to also grant privileges by using the WITH GRANT OPTION. The WITH GRANT OPTION lets you see who granted what privilege to what users and lets you perform a cascading revoke of privileges.
- The order of ACEs in the access privilege set is not a concern.

9.2.3 Privileges Required for Data Manipulation and Data Definition

Table 9–1 lists data manipulation and data definition operations that users can perform on a database and specifies which privileges the user needs to perform each operation. In all cases, the user needs at least the SELECT privilege in the access privilege set for the database. Without the SELECT privilege in the database access privilege set, the user cannot attach to the database, and receives a “No privilege” error message. In most cases, the user also needs privileges in one or more access privilege sets for tables, views, or columns to which the database refers.

The Oracle Rdb role-oriented database privileges, DBADM and SECURITY, can bypass the SELECT privilege, as can OpenVMS privileges SYSPRV, BYPASS, READALL, OPER, and SECURITY. On Digital UNIX, the users `dbsmgr` and `root` can bypass the SELECT privilege.

OpenVMS VAX OpenVMS Alpha

On OpenVMS, you must have the operating system READ privilege on entire directory path to the database, in addition to database privileges, to attach to the database. ♦

Table 9–1 Privileges Required for DML and DDL Operations

User Privileges Required for Data Manipulation		
To Perform the Following Operation:	**In the Database ACL:**	**In the Table, View, Function, or Module ACL:**
Attach to a database	SELECT	Not applicable
Attach to a database and restrict access	DBADM + SELECT	Not applicable
Connect to a session	SELECT	Not applicable
Connect to a session and restrict access	DBADM + SELECT	Not applicable
Display definitions	SELECT	SELECT on all tables and views being accessed
Execute a module (stored function or procedure)	SELECT	EXECUTE on the module
Execute an external routine (function or procedure)	SELECT	EXECUTE on the routine
Select rows	SELECT	SELECT on all tables and views being accessed
Insert rows	SELECT + INSERT	INSERT on table or view in which rows are stored

(continued on next page)

Table 9–1 (Cont.) Privileges Required for DML and DDL Operations

User Privileges Required for Data Manipulation		
To Perform the Following Operation:	**In the Database ACL:**	**In the Table, View, Function, or Module ACL:**
Update rows	SELECT + UPDATE	UPDATE on view in which rows are updated or on column to be updated. If UPDATE statement contains a WHERE clause that refers to columns in a table or right-hand side of assignment refers to columns in a table or uses ACL-style privileges, SELECT on table + UPDATE on view in which rows are updated or on column to be updated.
Delete rows	SELECT + DELETE	DELETE on table from which rows are deleted. If DELETE statement contains a WHERE clause that refers to columns in a table or uses ACL-style privileges, SELECT + DELETE on table.
Open cursors	SELECT +[1]	SELECT +[1]
Start a distributed transaction	SELECT + DISTRIBTRAN	

User Privileges Required for Data Definition		
To Perform the Following Operation:	**In the Database ACL:**	**In the Table, View, Function, or Module ACL:**
Alter database characteristics	DBADM	Not applicable
Drop database by path name	DBADM	Not applicable
Drop database by file name	DBADM	Not applicable
Modify privileges for database (ACL style)	DBCTRL + SELECT	Not applicable

[1]When an application program opens a cursor, Oracle Rdb checks that the user has all privileges necessary for the request. That is, if the application program inserts, updates, or deletes rows within the cursor, in the same module as the OPEN statement, Oracle Rdb checks that the user has the appropriate privileges necessary to perform those actions on the tables regardless of the actual run-time characteristics of the program. For example, if a module contains an SQL statement to delete a row from a cursor's result table, but the run-time characteristic of the program is to not execute the DELETE statement, the user must still have DELETE privilege on the table.

(continued on next page)

Table 9–1 (Cont.) Privileges Required for DML and DDL Operations

	User Privileges Required for Data Definition	
To Perform the Following Operation:	**In the Database ACL:**	**In the Table, View, Function, or Module ACL:**
Modify privileges for database (ANSI/ISO style)	SELECT + (GRANT option for privilege or DBCTRL)	Not applicable
Modify privileges for a table, view, or column (ACL style)	SELECT	DBCTRL on the table or view whose ACL is being modified
Modify privileges for a table, view, or column (ANSI/ISO style)	SELECT	GRANT option for privilege or DBCTRL on table for which privilege is granted
Grant users power to grant or revoke privileges (ACL style)	DBCTRL + SELECT	DBCTRL on table for which privilege is granted
Grant users power to grant or revoke privileges (ANSI/ISO style)	SELECT + (GRANT option for privilege or DBCTRL)	GRANT option for privilege or DBCTRL on table for which privilege is granted
Create a catalog	SELECT + CREATE	Not applicable
Drop a catalog	SELECT + DROP	Not applicable
Create a collating sequence	SELECT + CREATE	Not applicable
Drop a collating sequence	SELECT + DROP	Not applicable
Add a column	SELECT	ALTER + CREATE
Alter a column	SELECT	ALTER
Drop a column	SELECT	ALTER + DROP
Create a constraint	SELECT + CREATE	ACL style: (SELECT + CREATE) on tables to which the constraint refers ANSI/ISO style: (SELECT + REFERENCES) on tables or columns to be referenced

(continued on next page)

Table 9–1 (Cont.) Privileges Required for DML and DDL Operations

To Perform the Following Operation:	In the Database ACL:	In the Table, View, Function, or Module ACL:
	User Privileges Required for Data Definition	
Add a constraint	SELECT	ACL style: ALTER + CREATE + (SELECT + CREATE) on table containing constraint and on tables to which the constraint refers ANSI/ISO style: ALTER + REFERENCES + (SELECT + REFERENCES) on constraint tables or columns
Drop a constraint	SELECT	ALTER on table containing constraint and on all tables to which constraint refers
Drop a constraint (DROP CONSTRAINT)	SELECT	DROP on all tables to which constraint refers
Create a domain	SELECT + CREATE	Not applicable
Alter a domain	SELECT + CREATE	Not applicable
Drop a domain	SELECT + DROP	Not applicable
Create an external routine (function or procedure)	SELECT + CREATE	Not applicable
Drop an external routine (function or procedure)	SELECT	DROP on routine being deleted
Create an index	SELECT	CREATE on table associated with index
Alter an index	SELECT	ALTER on table associated with index
Disable an index	SELECT	DROP on table associated with the index
Drop an index	SELECT	DROP on table associated with the index
Create a module (stored procedure or function)	SELECT + CREATE	Not applicable
Drop a module (stored procedure or function)	SELECT	DROP on module being deleted
Drop a stored procedure or stored function	SELECT	ALTER on module containing the procedure or function
Create an outline		CREATE on tables referred to in the outline
Drop an outline		DROP on tables referred to in the outline
Create a schema	SELECT + CREATE	Not applicable

(continued on next page)

Table 9–1 (Cont.) Privileges Required for DML and DDL Operations

User Privileges Required for Data Definition		
To Perform the Following Operation:	**In the Database ACL:**	**In the Table, View, Function, or Module ACL:**
Drop a schema	SELECT + DROP	Not applicable
Add a storage area	DBADM	Not applicable
Alter a storage area	DBADM	Not applicable
Drop a storage area	DBADM	Not applicable
Create a storage map	SELECT	CREATE on table to which storage map refers
Alter a storage map	SELECT	ALTER on table to which storage map refers
Drop a storage map	SELECT	DROP on table to which storage map refers
Create a table	SELECT + CREATE	Not applicable[2]
Alter a table	SELECT + ALTER	ALTER[2]
Drop a table	SELECT	DROP on table being deleted and all tables to which any constraint in the table refers
Truncate a table	SELECT + DROP + CREATE	DROP on table being deleted and all tables to which any constraint in the table refers, and privileges to create the related metadata objects. [3]
Create a trigger	SELECT	SELECT + CREATE on table for which trigger is created + (privileges for actions taken by the trigger + DBCTRL on tables affected by the trigger)
Create a view	SELECT	SELECT + CREATE on all tables to which view refers
Drop a view	SELECT	DROP on view being deleted

User Privileges Required for Database Maintenance		
To Perform the Following Operation:	**In the Database ACL:**	**In the Table or View ACL:**
Export a database	DBADM + SELECT	SELECT on all tables
Integrate, updating a database	SELECT + (ALTER or CREATE or DROP)	SELECT + (ALTER or CREATE or DROP)

[2]See also the entries for constraints and columns.

[3] See the entries for adding a constraint and creating an index, storage map, and trigger.

(continued on next page)

Table 9–1 (Cont.) Privileges Required for DML and DDL Operations

	User Privileges Required for Database Maintenance	
To Perform the Following Operation:	**In the Database ACL:**	**In the Table or View ACL:**
Integrate, updating a repository	SELECT	SELECT

For information about how to override the database privileges, see Section 9.4.1.

For more information about database privileges, see the the GRANT and REVOKE statements in the *Oracle Rdb7 SQL Reference Manual*.

9.2.4 Building Access Control Lists

ACL-style protection consists of an access control list (ACL) made up of access control entries (ACEs). The position of an ACE in the list is significant in ACL-style protection; it is not relevant in ANSI/ISO-style protection.

When you create a database, Oracle Rdb automatically creates a default ACL. When you create a table, Oracle Rdb automatically creates a default ACL. These ACLs contain the following entries:

- The owner's, which grants all access privileges. These privileges include the DBCTRL privilege, which lets you modify access privilege sets.
- The entry with the identifier [*,*], which denies privileges to all other users. If you, as owner, want to allow selective access to a database, you must change this entry as part of the process of defining protection.

To see the ACL for a database or database object, use the SHOW PROTECTION statement. The SHOW PROTECTION statement displays the ACL in its correct order so you can see where to place new entries.

Example 9–1 displays the ACEs for a database owned by the identifier [grp2,jones], who created the database and the tables.

Example 9–1 Issuing SHOW PROTECTION Statements

```
SQL> SHOW PROTECTION ON DATABASE RDB$DBHANDLE
Protection on Alias RDB$DBHANDLE
  (IDENTIFIER=[grp2,jones],ACCESS=SELECT+INSERT+UPDATE+DELETE+SHOW+CREATE+
    ALTER+DROP+DBCTRL+OPERATOR+DBADM+SECURITY+DISTRIBTRAN)
  (IDENTIFIER=[*,*],ACCESS=NONE)
SQL> --
SQL> SHOW PROTECTION ON TABLE SALARY_HISTORY
Protection on Table SALARY_HISTORY
  (IDENTIFIER=[grp2,jones],ACCESS=SELECT+INSERT+UPDATE+DELETE+SHOW+CREATE+
    ALTER+DROP+DBCTRL+REFERENCES)
  (IDENTIFIER=[*,*],ACCESS=NONE)
SQL> --
SQL> SHOW PROTECTION ON COLUMN EMPLOYEES.EMPLOYEE_ID
Protection on Column EMPLOYEES.EMPLOYEE_ID
SQL>
```

When a user tries to perform an operation on a database, table, view, or column, Oracle Rdb reads the associated ACL from top to bottom, comparing the identifiers currently held by the user with each entry. As soon as Oracle Rdb finds the first match, it grants the privileges listed in that entry. For this reason, both the ACEs themselves and their order in the list are important.

To create an ACL, you can enter the individual GRANT statements using interactive SQL. Usually, it is easier to use a text editor to build an SQL script that defines protection for the whole database.

The script method is useful when you want to keep a permanent record of changes to the ACLs. You can use interactive SQL when you want to create a prototype of new entries or make temporary changes to an existing ACL.

You start building the script by creating the ACL for the database. Then you add the table ACLs. The following list shows the steps necessary to create a database or table ACL:

1. Make a list of database users, the functions they perform, and the minimum privileges they need to carry out these functions. You can use comments to make your restrictions clear.
2. Arrange the entries in the order you want them in the ACL.
3. Edit the entries to create the GRANT statements.

The following discussion shows the first step in creating a command file to add ACEs to the database. The examples include the comments, list of privileges, and identifiers for the ACL entry. The second step is discussed in Section 9.2.5; The third step, adding the initial GRANT portion of the statement and the

POSITION or AFTER clauses (or both clauses) is discussed in Section 9.3 and Section 9.3.1.

Assume that you are the owner and your user identifier is [grp2,jones]. Protection for the owner is defined by default to have all privileges and is placed in position 1 of the ACL.

User [grp2,clark] will help create and perform minor changes on databases. Therefore, the user must have privileges to use CREATE, ALTER, and DROP to make changes to tables. To perform data definition, the user also must have SELECT access to system tables. However, the user should not be able to change data in the database. Deny her access to update statements and to DBCTRL and DBADM privileges:

```
-- Assistant -- needs to use data definition statements.
--
     SELECT, CREATE, ALTER, DROP
     [grp2,clark]
```

OpenVMS VAX OpenVMS Alpha

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. They also need to use the two-phase commit protocol to process distributed transactions. Grant them all the privileges except those associated with database maintenance:

```
-- Programmers -- need to perform data definition, including creating
-- temporary tables, and data manipulation to test application programs.
--
     SELECT, INSERT, UPDATE, DELETE, CREATE, ALTER, DROP,
     DISTRIBTRAN, REFERENCES
     PROGRAMMERS                                   ♦
```

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

```
-- Clerks -- need to be able only to read data.
--
     SELECT
     [clercl,*]
```

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

```
--
-- Secretary -- needs to use all data manipulation statements.
-- No access to data definition or maintenance.
--
     SELECT, INSERT, UPDATE, DELETE
     [clercl,ford]
```

9.2.5 Putting the Access Control List in Order

The order in which the entries appear on the ACL determines the protection actually applied to the database.

When a user tries to perform an Oracle Rdb operation on a database or table, Oracle Rdb reads the ACL for the database object from top to bottom, comparing the identifiers held by the user with the identifiers listed in each entry. When Oracle Rdb finds the first match, it grants the privileges listed in that entry and stops the search.

All user identifiers that do not match a previous entry “fall through” to the entry [*,*], if it exists. If there is no entry with the user identifier [*,*], unmatched user identifiers are denied all access to the database or table.

Remember the following general guidelines for ordering ACEs:

- The more powerful the privilege, the higher on the list that ACE should go.
- The less restrictive the user identification code, the lower on the list that ACE should go.

Using the sample from Section 9.2.4, you might put the entries in the following order:

```
--
-- Owner -- already defined, in position 1 of the ACL, with all privileges.
--
     [grp2,jones]
--
-- Assistant -- needs to use data definition statements.
--
     [grp2,clark]
--
-- Secretary -- needs to be able to read, write, and delete data.
-- No access to data definition or maintenance.
--
     [clercl,ford]
```

```
--
-- Programmers -- need to perform data definition, including creating
-- temporary tables, and data manipulation 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.
--
     [clercl,*]
--
```

Because Oracle Rdb 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 user identifier, [*,*], first in the list, all users match it, and Oracle Rdb grants or denies all the access privileges specified there to all users.

User [grp2,jones] would also match any of the following user identifiers from an ACL:

[*,jones]
[grp2,*]
[*,*]

Similarly, if you place the general entry [clercl,*] before the specific entry [clercl,ford], Oracle Rdb matches user [clercl,ford] with [clercl,*] and denies the access privileges INSERT, UPDATE, and DELETE, which user [clercl,ford] needs.

The following guidelines may help when you order ACEs:

- An entry that identifies a certain user (for example [grp1,jblown]) should occur before an entry that identifies a group to which that user belongs (for example, [grp1,*], PROGRAMMERS, or [*,*]).
- An entry that identifies a group of users (for example, PROGRAMMERS or [grp1,*]) should occur before an entry for the public ([*,*]).
- Entries that deny privileges to users or groups who log into the system a certain way should occur before more privileged entries that do not specify login characteristics. For example, an entry that contains the identifier NETWORK and a null privilege list should occur before an entry that contains the identifier [*,*] and the SELECT privilege.

See the *Oracle Rdb7 SQL Reference Manual* for more information on the GRANT and REVOKE statements. For both statements, read the text about including or omitting an AFTER or POSITION clause to better understand how you control order or selection of entries when granting and revoking privileges.

9.3 Granting and Revoking Privileges

You use the GRANT and REVOKE statements to control data definition and data manipulation operations for a database and database objects.

To deny a privilege to users, simply do not grant it, or revoke the privilege from existing entries if applicable. There are no negative forms of SQL privileges. NOSELECT, for example, is invalid.

You must attach to the database to display the database protection or to issue a GRANT or REVOKE statement.

To attach to a database, you must already have SELECT privilege for that database. Therefore, the database SELECT privilege is implicitly required for all data definition and data manipulation statements.

Use the GRANT statement to:

- Create a new entry in an access privilege set

  If [grp1,*] is not an identifier in an ACE for the CURRENT_INFO table, the following statement inserts a new entry with this identifier:

  ```
  SQL> GRANT SELECT ON TABLE CURRENT_INFO TO [grp1,*];
  ```

- Add one or more privileges to an existing entry

  For example, if [grp1,rhonda] is already an identifier entered in the access privilege set for a database with the default alias of RDB$DBHANDLE, the following statement adds UPDATE and DELETE to the user's list of privileges:

  ```
  SQL> GRANT UPDATE,DELETE ON DATABASE ALIAS RDB$DBHANDLE TO [grp1,rhonda];
  ```

Use the REVOKE statement to:

- Delete an access privilege set entry

  For example, the following statement deletes the entry for the user identifier [grp1,rhonda] for a database with the alias PERS.

  ```
  SQL> REVOKE ENTRY ON DATABASE ALIAS PERS FROM [grp1,rhonda];
  ```

- Remove one or more privileges from an existing entry

  For example, the following statement removes the DBADM privilege from the first entry with the general identifier MANAGERS for the database with the default alias RDB$DBHANDLE:

  ```
  SQL> REVOKE DBADM ON DATABASE ALIAS RDB$DBHANDLE FROM MANAGERS;
  ```

If you want to create an entry with an empty privilege list (to deny all privileges to a specific user or group of users), you must first use a GRANT statement followed by a REVOKE statement. GRANT is the only statement that creates an ACE, however the statement does not allow a null privilege list. To include an entry with a null privilege list, use the GRANT statement to create an entry with the user's or group's identifier and then enter a REVOKE statement to remove all privileges. Example 9–2 creates an entry at the beginning of the ACL for the EMPLOYEES table and denies all privileges to users whose group is grp1.

Example 9–2 Denying Privileges to a Group of Users

```
SQL> GRANT ALL PRIVILEGES ON TABLE EMPLOYEES TO [grp1,*];
SQL>
SQL> REVOKE ALL PRIVILEGES ON TABLE EMPLOYEES FROM [grp1,*];
```

Remember, users are denied access to an entire database if their identifiers do not match an ACE specified for the database. If users do have privileges for the database, they cannot access specific tables or views if their identifiers do not match one of those specified in the table or view ACEs. Therefore, you need to worry about explicitly denying access to users only if one of the following conditions applies:

- An ACE grants privileges to all members of a user's group or to the public ([*,*])
- You want to deny database access to certain login modes (OpenVMS only)

You can grant users privileges for a view and deny users the same privileges for the tables on which the view is based. To give certain users the right to see only a subset of columns or rows in a table, create a view that contains the subset and grant the users privileges only to the view.

After you commit changes to an access privilege set by a GRANT or REVOKE statement, the changes may not immediately affect users already attached to the database. The SHOW PRIVILEGES and SHOW PROTECTION statements displays any access privilege set changes you make even before you commit the changes; however, the changes may not affect active users until the next time they attach to the database. (Remember that this is true for your own process when you change privileges for the identifier that applies to you.)

In addition, when you change access privilege sets (particularly the one for the database), affected users may be prevented from using tables or may inappropriately retain access to tables until you commit the changes. Changes to definitions, including access privilege sets, often cause a transaction share mode to be upgraded to exclusive. For

information on row and table locking that occurs during transactions, see the *Oracle Rdb7 Guide to SQL Programming*.

For more information on the GRANT and REVOKE statements, including complete syntax, see the *Oracle Rdb7 SQL Reference Manual*.

9.3.1 Defining Protection for Databases

To define protection for a database, you can execute a command procedure, such as mf_personnel_acls.sql, from interactive SQL. If errors occur, roll back the transaction, edit the procedure to correct the problem, and execute the procedure again. When you achieve error-free execution of the procedure, use the SHOW PROTECTION statement to display the ACLs you modified. After verifying that the ACEs are correct and in the order you intended, commit the transaction.

Example 9–3 illustrates a section from a command procedure that modifies ACL-style default protection for the mf_personnel database. For each user group identified in the ACL, entries are ordered from most to least privileged. Note that this is a fairly restrictive set of ACL entries.

Example 9–3 Defining Protection on a Database

```
-- mf_personnel_acls.sql
--
ATTACH 'ALIAS PERS filename mf_personnel';
--
-- ************************************************
-- * Changes to ACL for database
-- ************************************************
--
-- Owner [grp2,jones]--leave privileges (ALL) and position (1) as is.
--
-- Assistant--needs to use data definition statements
--
GRANT SELECT, CREATE, ALTER, DROP
   ON DATABASE ALIAS PERS
   TO [grp2,clark]
AFTER [grp2,jones];
--
-- Secretary--needs to use all data manipulation statements
--
GRANT SELECT, INSERT, UPDATE, DELETE
   ON DATABASE ALIAS PERS
   TO [clercl,ford]
AFTER [CLERCL,SMITH];
```

(continued on next page)

Example 9–3 (Cont.) Defining Protection on a Database

```
--
-- Programmers -- need to perform data definition, including creating
-- temporary tables, and data manipulation to test application programs.
--
GRANT SELECT, INSERT, UPDATE, DELETE, CREATE, ALTER, DROP,
   DISTRIBTRAN, REFERENCES
   ON DATABASE ALIAS PERS
   TO PROGRAMMERS
AFTER [clercl,ford];
--
-- Remainder of users in CLERCL group--need to read data
--
GRANT SELECT
   ON DATABASE ALIAS PERS
   TO [clercl,*]
AFTER PROGRAMMERS;
--
```

Note that the order of the ACEs is not relevant for ANSI/ISO-style privileges.

9.3.2 Defining Protection for Tables

The ACL shown in Example 9–3 grants database access to all the users who need it. You might want to put additional restrictions on certain tables in the database.

For example, the SALARY_HISTORY table contains sensitive information. Only the department secretary should have the privileges to run the programs that read, write, and modify the SALARY_HISTORY table.

Example 9–4 contains a series of entries for defining table-level protection.

Example 9–4 Defining Protection on a Table

```
-- ************************************************
-- * Changes to ACL for table EMPLOYEES
-- ************************************************
--
-- Owner [grp2,jones]--leave privileges (ALL) and position (1) as is.
```

(continued on next page)

Example 9–4 (Cont.) Defining Protection on a Table

```
--
-- Assistant--needs to use data definition statements
--
GRANT SELECT, CREATE, ALTER, DROP
   ON TABLE PERS.EMPLOYEES
   TO [grp2,clark]
AFTER [grp2,jones];
--
-- Manager--needs to use all data manipulation statements
--
GRANT SELECT, INSERT, UPDATE, DELETE
   ON TABLE PERS.EMPLOYEES
   TO [clercl,smith]
AFTER [grp2,clark];
--
-- Secretary--needs to use all data manipulation statements
--
GRANT SELECT, INSERT, UPDATE, DELETE
   ON TABLE PERS.EMPLOYEES
   TO [clercl,ford]
AFTER [clercl,smith];
--
-- Programmers -- need to use all data manipulation statements
-- and to create and drop indexes.
--
GRANT SELECT, INSERT, UPDATE, DELETE, CREATE, ALTER, DROP
   ON TABLE PERS.EMPLOYEES
   TO PROGRAMMERS
AFTER [clercl,ford];
--
-- Remainder of users in CLERCL group--need to read data
--
GRANT SELECT
   ON TABLE PERS.EMPLOYEES
   TO [clercl,*]
AFTER PROGRAMMERS;
--
-- Secretary has access to SALARY_HISTORY
--
GRANT SELECT, INSERT, UPDATE, DELETE
   ON TABLE PERS.SALARY_HISTORY
   TO [clercl,ford];
```

9.3.3 Defining Protection for Columns

Only certain privileges apply to column-specific protection. These are:

- UPDATE
- REFERENCES

You need the REFERENCES privilege to define constraints that affect a particular column. You need the UPDATE privilege to update data in a column. A user with the UPDATE privilege on a table automatically receives the UPDATE privilege on all columns in that table. To update a column you must have UPDATE privilege either for the column or for the table. You can restrict UPDATE privileges by defining them only on columns users should be able to update, and then removing the UPDATE privilege from the table entry.

For example, because current salary is a sensitive piece of information, you might want to restrict the ability to update this data. Example 9–5 shows how to prevent user [clercl,ford] from updating any column in SALARY_HISTORY except SALARY_START and SALARY_END. User [clercl,ford] *cannot* update SALARY_AMOUNT.

Example 9–5 Defining Column Protection

```
SQL> GRANT UPDATE ON COLUMN SALARY_HISTORY.SALARY_START TO [clercl,ford];
SQL> GRANT UPDATE ON COLUMN SALARY_HISTORY.SALARY_END TO [clercl,ford];
SQL> --
SQL> REVOKE UPDATE ON TABLE SALARY_HISTORY FROM [clercl,ford];
SQL> --
SQL> COMMIT;
SQL> --
SQL> SHOW PROTECTION ON TABLE SALARY_HISTORY;
Protection on Table SALARY_HISTORY
  (IDENTIFIER=[grp2,jones],ACCESS=SELECT+INSERT+UPDATE+DELETE+SHOW+CREATE+
    ALTER+DROP+DBCTRL+REFERENCES)
  (IDENTIFIER=[*,*],ACCESS=NONE)
SQL> --
SQL> SHOW PROTECTION ON COLUMN SALARY_HISTORY.SALARY_START;
Protection on Column SALARY_HISTORY.SALARY_START
  (IDENTIFIER=[clercl,ford],ACCESS=UPDATE)
  (IDENTIFIER=[*,*],ACCESS=NONE)
```

The following example shows an unsuccessful attempt by [clercl,ford] to update the SALARY_AMOUNT column:

```
SQL> UPDATE SALARY_HISTORY
cont>   SET SALARY_AMOUNT = 35000
cont>   WHERE EMPLOYEE_ID = '00164';
%RDB-E-NO_PRIV, privilege denied by database facility
```

9.3.4 Restricting Access to Tables by Using Views

The discussion of views in Chapter 3 mentions security as one of the advantages to creating these “virtual” tables. You can use a view to restrict access to specific columns of one or more tables or views. You can also apply precise database access privileges to those columns in the view definition to maintain the required level of security for your database.

You can define a view based on:

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

Oracle Rdb allows you to specify access privileges for every table. However, if you grant a user SELECT access to a table, you make every column in the row available for retrieval by that user. You cannot restrict access to specific columns in that row with table-level protection.

If your intention is to allow a user to access only two columns in each of two tables, you must secure the columns in the tables by denying certain access privileges at the table level for that group of users. Next, you define a view that includes only four columns, two from each table. Then, you define protection for the view that allows certain users SELECT access to the four columns from the two tables. You must revoke the SELECT privilege from that user’s access privilege set for the base table to make the security provided by the view effective.

Views provide column-level protection for your database by making a subset of a table’s columns, of rows, or of both columns and rows available to authorized users.

If you grant restricted access to any user to the data in a table, you should not give the user the CREATE privilege on the table or the GRANT option in association with other privileges. In the first instance, a user may define his or her own views to access a table’s data and defeat the original restrictions. In the second instance, giving a user the GRANT option in association with other privileges gives that user the power to give those privileges to other users.

When you grant or deny access privileges for a particular view, Oracle Rdb evaluates the ACEs for that view, but does not evaluate the ACEs from the underlying tables or views. Example 9–6 shows a view definition that contains only some of the columns from the EMPLOYEES and SALARY_HISTORY TABLES. It does not contain sensitive information such as BIRTHDAY and SALARY_AMOUNT.

Example 9–6 Creating a View to Restrict Access to the Table

```
SQL> CREATE VIEW EMPLOYEE_INFO
cont>  AS SELECT
cont>        E.LAST_NAME, E.FIRST_NAME,
cont>        E.EMPLOYEE_ID, SH.SALARY_START
cont>      FROM  SALARY_HISTORY SH, EMPLOYEES E
cont>      WHERE SH.EMPLOYEE_ID = E.EMPLOYEE_ID
cont>            AND
cont>            SH.SALARY_END IS NULL;
```

Now you can restrict access privileges to the underlying tables and grant them for the subset of columns and rows defined in the view.

Example 9–7 shows how you can restrict unauthorized access to the SALARY_HISTORY table while specifying SELECT access for the EMPLOYEE_INFO view.

Example 9–7 Restricting Access with View Definitions

```
SQL> REVOKE SELECT, INSERT, UPDATE, DELETE, CREATE, DROP, ALTER
cont>       ON TABLE SALARY_HISTORY
cont>       FROM PROGRAMMER;
SQL>
SQL> GRANT SELECT
cont>      ON TABLE EMPLOYEE_INFO
cont>      TO PROGRAMMER;
```

You can provide other views based on the same table to allow other groups of users only the access privileges they require. Unauthorized users encounter an error message similar to the following:

```
%RDB-E-NO_PRIV, privilege denied by database facility
```

View definitions let you control access to an entire table, allowing accessing by one or more groups responsible for the data in that table, while maintaining security for all the data in the database.

9.3.5 Restricting Access to a Subset of Rows

You can grant users access to a specific set of rows in a table by tracking which users entered the data and controlling access to the data. Take the following steps to control access:

1. Add a column to the table to hold the user name of the person entering the data.
2. Create a trigger that uses the CURRENT_USER built-in function to track the user name of the person entering the data.
3. Create a trigger that prevents users from modifying the column that holds the user name.
4. Create a view that allows users to access only those rows that they inserted.
5. Set the protection on the EMPLOYEES table so that unauthorized users cannot access the table.

The following series of examples shows how you restrict access to employee records in the EMPLOYEES table, allowing only the user who entered the data to access the data.

You must modify the EMPLOYEES table to add a column that stores the user name of the person entering the data. Then, you create a trigger that uses the built-in function CURRENT_USER as the default value and executes after a user inserts a new row into the EMPLOYEES table. Example 9–8 shows how to modify the EMPLOYEES table and how to create the trigger.

Example 9–8 Adding a Column and a Trigger to Track Users

```
SQL> ALTER TABLE EMPLOYEES
cont>      ADD COLUMN USER_ID CHAR(31);
SQL>
SQL> -- Create the trigger.
SQL> CREATE TRIGGER TAG_EMPLOYEES
cont>        AFTER INSERT ON EMPLOYEES
cont>              (UPDATE EMPLOYEES E
cont>               SET USER_ID = CURRENT_USER
cont>               WHERE EMPLOYEES.DBKEY = E.DBKEY)
cont>        FOR EACH ROW;
```

To protect the USER_ID column from being modified, create another trigger, as shown in Example 9–9.

Example 9–9 Preventing Modification of a Column with a Trigger

```
SQL> CREATE TRIGGER DONT_CHANGE_USERID
cont>       AFTER UPDATE ON EMPLOYEES
cont>             REFERENCING OLD AS OLD NEW AS NEW
cont>       WHEN (NEW.USER_ID <> OLD.USER_ID)
cont>            (ERROR)
cont>       FOR EACH ROW;
```

If users attempt to modify the USER_ID column, they receive an error, as the following example shows:

```
SQL> UPDATE EMPLOYEES SET USER_ID = 'BROWN' WHERE USER_ID = 'GREMBOWSKI';
%RDB-E-TRIG_INV_UPD, invalid update; encountered error condition defined for
trigger
-RDMS-E-TRIG_ERROR, trigger DONT_CHANGE_USERID forced an error
-RDB-F-ON_DB, on database DISK1:[SAMPLE]MF_PERSONNEL.RDB;1
```

To allow users to access only those rows they inserted, create a view, as shown in Example 9–10.

Example 9–10 Creating a View That Restricts Access to Certain Records

```
SQL> CREATE VIEW SELECTIVE_EMPLOYEES_UPDATE
cont>    AS SELECT * FROM EMPLOYEES
cont>       WHERE USER_ID = CURRENT_USER
cont>    WITH CHECK OPTION CONSTRAINT MUST_HAVE_USER;
```

When a user selects data from this view, Oracle Rdb returns only records that contain the user name of the current process, thus preserving the privacy of the other rows. If the user name of the current user is GREMBOWSKI, SQL returns only those rows that contain GREMBOWSKI in the USER_ID column, as the following example shows:

```
SQL> SELECT EMPLOYEE_ID, LAST_NAME, USER_ID
cont>       FROM SELECTIVE_EMPLOYEES_UPDATE;
 EMPLOYEE_ID   LAST_NAME        USER_ID
 33334         DAY              GREMBOWSKI
1 row selected
SQL>
```

When a user attempts to update a row, the CHECK OPTION clause enforces the WHERE clause restriction on the entered data. The current user cannot update rows that do not contain the user name of the current user. In addition, the trigger DONT_CHANGE_USERID makes sure that the user does not change the USER_ID column to any value other than his or her current user name.

The following example shows that because the row with EMPLOYEE_ID 00164 does not contain the user name of the current user, SQL updates no rows.

```
SQL> UPDATE SELECTIVE_EMPLOYEES_UPDATE
cont>        SET STATUS_CODE = '2'
cont>        WHERE EMPLOYEE_ID = '00164';
0 rows updated
```

For the protection to be effective, you must revoke privileges on the EMPLOYEES table. Example 9–11 revokes all privileges for the user group CLERCL.

Example 9–11 Revoking Protection on the Underlying Table

```
SQL> REVOKE ALL ON TABLE EMPLOYEES
cont>       FROM [CLERCL,*];
```

9.3.6 Using Views to Maintain Role-Oriented Access

In many organizations, users can change roles from month to month, making the maintenance of ACES in the database difficult. In this type of environment, it is more convenient to assign roles to users, and use these granted roles for the access checking.

The same techniques shown in Section 9.3.5 can also be used to translate the user name through a table that maps the user name to a role in the organization. The database administrator needs to maintain only this table.

For example, a new user, JONES, is granted the role of SUPERVISOR while the manager is on leave. The database administrator adds the following entry to the ACCESS_ROLES table:

```
SQL> INSERT INTO ACCESS_ROLES
cont>           (USER_ID, ACCESS_START, ACCESS_END, GRANTED_ROLE)
cont>     VALUES ('JONES', DATE '1996-5-1, DATE '1996-8-1', 'SUPERVISOR');
```

This entry defines JONES as having the SUPERVISOR role from May 1, 1996 until August 1, 1996, the date of the manager's return.

Then, you create a view to check for role-oriented privileges, as shown in Example 9–12. Note that you must also add the column GRANTED_ROLE to the EMPLOYEES table and insert the appropriate role into all rows in the column.

Example 9–12 Creating a View to Check for Role-Oriented Privileges

```
SQL> CREATE VIEW SELECTIVE_EMPLOYEES_UPDATE
cont>       AS SELECT *  FROM EMPLOYEES
cont>             WHERE EXISTS (SELECT GRANTED_ROLE FROM ACCESS_ROLES AR
cont>                   WHERE EMPLOYEES.GRANTED_ROLE = AR.GRANTED_ROLE
cont>                         AND USER_ID = CURRENT_USER
cont>                         AND CURRENT_DATE
cont>                             BETWEEN ACCESS_START AND ACCESS_END)
cont>       WITH CHECK OPTION CONSTRAINT MUST_HAVE_USER_ROLE;
```

This view contains a subquery that finds all roles that the current user can perform currently. For example, any EMPLOYEES row that matches any of these roles can be viewed and updated by JONES. JONES may be able to play several roles, and can choose to use any of them when updating the EMPLOYEES row by providing the role in the INSERT statement. Note also that the CHECK OPTION constraint forces the selected role to be one previously granted to the user JONES by a more privileged user.

9.3.7 Defining Default Protection

Existing applications might depend upon some type of public access to newly created tables and views in a database. You can override the default privileges for [*,*] by using the OpenVMS rights identifier [DEFAULT] within the database's access privilege set. Use the [DEFAULT] identifier to store the default privileges for [*,*]. The [DEFAULT] ACE indicates the protection given to [*,*] for newly created named tables and views in that database, as demonstrated in Example 9–13.

Example 9–13 Defining Default Protection

```
SQL> CREATE DATABASE ALIAS TEST1 FILENAME TEST1;
SQL> SHOW PROTECTION ON DATABASE TEST1;
Protection on Alias TEST1
  (IDENTIFIER=[grp2,jones],ACCESS=SELECT+INSERT+UPDATE+DELETE+SHOW+CREATE+
    ALTER+DROP+DBCTRL+OPERATOR+DBADM+SECURITY+DISTRIBTRAN)
  (IDENTIFIER=[*,*],ACCESS=NONE)
```

(continued on next page)

Example 9–13 (Cont.) Defining Default Protection

```
SQL> --
SQL> GRANT SELECT, INSERT
cont> ON DATABASE ALIAS TEST1
cont> TO [DEFAULT];
SQL> --
SQL> SHOW PROTECTION ON DATABASE TEST1;
Protection on Alias TEST1
  (IDENTIFIER=DEFAULT,ACCESS=SELECT+INSERT)
  (IDENTIFIER=[grp2,jones],ACCESS=SELECT+INSERT+UPDATE+DELETE+SHOW+CREATE+
    ALTER+DROP+DBCTRL+OPERATOR+DBADM+SECURITY+DISTRIBTRAN)
  (IDENTIFIER=[*,*],ACCESS=NONE)
SQL> COMMIT;
SQL> DISCONNECT DEFAULT;
SQL> --
SQL> ATTACH 'ALIAS TEST1 FILENAME TEST1';
SQL> CREATE TABLE TEST1.TABLE1
cont>   (COLUMN_ONE CHAR(5),
cont>    COLUMN_TWO CHAR(5));
SQL> --
SQL> SHOW PROTECTION ON TABLE TEST1.TABLE1;
Protection on Table TABLE1
  (IDENTIFIER=[grp2,jones],ACCESS=SELECT+INSERT+UPDATE+DELETE+SHOW+CREATE+
    ALTER+DROP+DBCTRL+REFERENCES)
  (IDENTIFIER=[*,*],ACCESS=SELECT+INSERT)
SQL> COMMIT;
SQL> DISCONNECT DEFAULT;
```

The [DEFAULT] ACE is stored in the same format as other ACEs in the database. It is not recognized until the current database attach is disconnected. When you attach to the database again and create a table or view, the [DEFAULT] ACE assigns privileges to [*,*] for that table or view. Changing the [DEFAULT] ACE does not change the [*,*] ACE for objects already created. Creating a [DEFAULT] ACE for objects other than the database is allowed, but only has meaning when it is placed in the database's access privilege set. ♦

9.4 Verifying Protection for a Database

You can use the SHOW PROTECTION statement to verify the protection for databases and tables. You must issue a separate SHOW PROTECTION statement for each object. Before issuing any statements, be sure to attach to the database.

Example 9–14 illustrates verifying the ACLs for the mf_personnel database and the SALARY_HISTORY table, and identifies each of the following steps with numbered callouts.

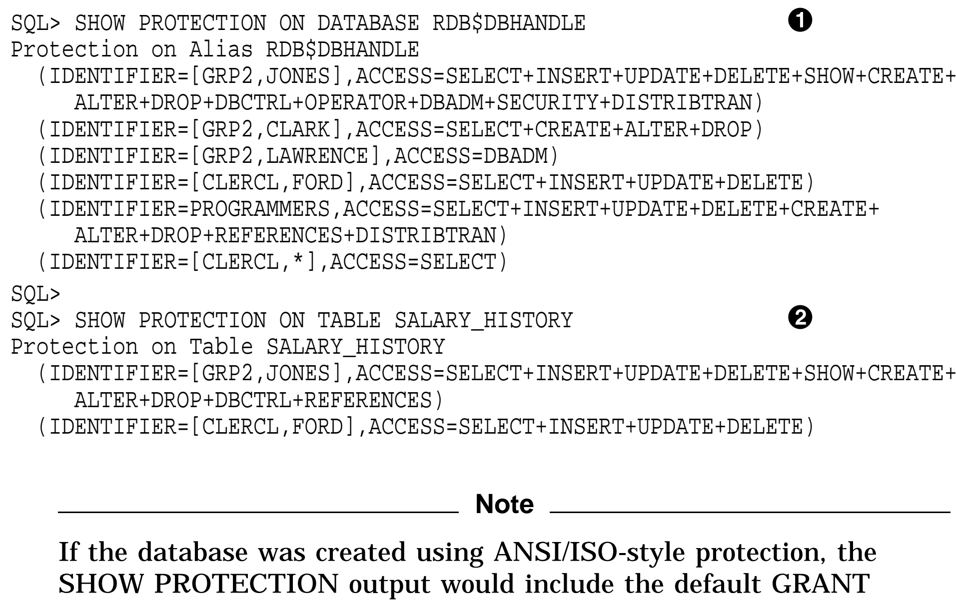
❶ Verifies the database ACL.

To display all the ACEs for a database for which you have SELECT privilege, include the DATABASE keyword in the SHOW PROTECTION statement. Identify the database using the alias for the database rather than a database file or path name. If you do not specify an alias, the default alias is RDB$DBHANDLE.

❷ Verifies a table ACL.

To display a table ACL, you must include the name of the table in the statement.

Example 9–14 Verifying ACLs



```
SQL> SHOW PROTECTION ON DATABASE RDB$DBHANDLE                        ❶
Protection on Alias RDB$DBHANDLE
  (IDENTIFIER=[GRP2,JONES],ACCESS=SELECT+INSERT+UPDATE+DELETE+SHOW+CREATE+
     ALTER+DROP+DBCTRL+OPERATOR+DBADM+SECURITY+DISTRIBTRAN)
  (IDENTIFIER=[GRP2,CLARK],ACCESS=SELECT+CREATE+ALTER+DROP)
  (IDENTIFIER=[GRP2,LAWRENCE],ACCESS=DBADM)
  (IDENTIFIER=[CLERCL,FORD],ACCESS=SELECT+INSERT+UPDATE+DELETE)
  (IDENTIFIER=PROGRAMMERS,ACCESS=SELECT+INSERT+UPDATE+DELETE+CREATE+
     ALTER+DROP+REFERENCES+DISTRIBTRAN)
  (IDENTIFIER=[CLERCL,*],ACCESS=SELECT)
SQL>
SQL> SHOW PROTECTION ON TABLE SALARY_HISTORY                         ❷
Protection on Table SALARY_HISTORY
  (IDENTIFIER=[GRP2,JONES],ACCESS=SELECT+INSERT+UPDATE+DELETE+SHOW+CREATE+
     ALTER+DROP+DBCTRL+REFERENCES)
  (IDENTIFIER=[CLERCL,FORD],ACCESS=SELECT+INSERT+UPDATE+DELETE)
```

Note

If the database was created using ANSI/ISO-style protection, the SHOW PROTECTION output would include the default GRANT OPTION privileges.

To see only your own access privileges, use the SHOW PRIVILEGES statement as Example 9–15 illustrates.

Example 9–15 Issuing the SHOW PRIVILEGES Statement

```
SQL> SHOW PRIVILEGES ON DATABASE RDB$DBHANDLE
Privileges on Alias RDB$DBHANDLE
  (IDENTIFIER=[GRP2,JONES],ACCESS=SELECT+INSERT+UPDATE+DELETE+SHOW+CREATE
    +ALTER+DROP+DBCTRL+OPERATOR+DBADM+REFERENCES+SECURITY+DISTRIBTRAN)
```

The SHOW PRIVILEGES statement displays the access privilege set and identifier of the user who is executing the command. Oracle Rdb matches the user's identifier with the identifier specified in the access privilege set. If your protection is ACL style, remember that Oracle Rdb reads the list from top to bottom. Although your identifier might match many ACEs, Oracle Rdb grants you access privileges when it finds the first match between your identification code and an identifier in the ACE.

9.4.1 Privileges with Override Capability

There are two database privileges that are role-oriented: DBADM and SECURITY. These privileges can override the access privilege sets for certain named objects in order to perform certain database-wide operations. Also, users possessing certain operating system privileges have some override capability on Oracle Rdb object access privilege sets. Oracle Rdb role-oriented privileges are database-wide (limited to the database in which they are granted), whereas operating system privileges are system-wide (span all databases on the system).

If you have one of these role-oriented privileges, you are implicitly granted certain other Oracle Rdb privileges. An *implicit* privilege is a privilege granted as a result of an override. You operate as if you actually hold the privilege, but the privilege is not explicitly granted and stored in an ACE for the object.

The DBADM database privilege allows you to perform any data definition or data manipulation operation on any named object, including the database, regardless of the access privilege set for the object. This is the most powerful privilege in Oracle Rdb because it can override most privilege checks performed by Oracle Rdb. If you possess the DBADM database privilege, you implicitly receive ALL privileges for all objects, except SECURITY database privileges.

If you possess the SECURITY database privilege, you implicitly receive SELECT, INSERT, UPDATE, and DELETE database privileges, and receive implicit DBCTRL (or GRANT) privilege for all objects. (The GRANT privilege refers to the option in ANSI/ISO-style protection of allowing subjects the ability to grant or revoke a particular privilege. For instance, the process with the

user identifier of [grp1,jones] could have the SELECT privilege with GRANT OPTION.)

On Digital UNIX, the users `root` and `dbsmgr` can perform any operation on any named object, including the database, regardless of the access privilege set of the object. ♦

Table 9–2 shows which privileges can be overridden by the Oracle Rdb or OpenVMS privileges.

Table 9–2 Privilege Override Capability

	Database Privileges		OpenVMS Privileges				
Privilege	**DBADM**	**SECURITY**	**SYSPRV**	**BYPASS**	**READALL**	**OPER**	**SECURITY**
ALTER	Y	N	Y	Y	N	N	N
CREATE	Y	N	Y	Y	N	N	N
DBADM	N/A	N	Y	N	N	N	N
DBCTRL	Y	Y	Y	N	N	N	Y
DELETE (database)	Y	Y	Y	Y	N	Y	Y
DELETE (table)	Y	N	Y	Y	N	N	N
DISTRIBTRAN (database only)	Y	N	Y	Y	N	N	N
DROP	Y	N	Y	Y	N	N	N
EXECUTE	Y	Y	N	N	N	N	Y
INSERT (database)	Y	Y	Y	Y	N	Y	Y
INSERT (table)	Y	N	Y	Y	N	N	N
REFERENCES	Y	N	Y	Y	N	N	N
SECURITY	N	N/A	N	N	N	N	Y
SELECT (database)	Y	Y	Y	Y	Y	Y	Y
SELECT (table)	Y	N	Y	Y	Y	N	N
SHOW	Y	N	Y	Y	Y	N	N
UPDATE (database)	Y	Y	Y	Y	N	N	Y
UPDATE (table)	Y	N	Y	Y	N	Y	N

♦

9.5 Understanding Privilege Checking for Oracle RMU Commands

You can control privileges for database maintenance operations with RMU privileges. RMU privileges provide the following features:

- You can easily determine the set of privileges that a user needs to issue RMU commands
- Most RMU commands do not require operating system privileges
- You can grant users access to only the specific RMU commands that they normally need for each database.
- You can perform security auditing for most RMU commands. See the *Oracle RMU Reference Manual* for more information.

Oracle Rdb creates an access control list (ACL) by default on the root file of each database. To be able to use a particular RMU command for the database, you must be granted the appropriate RMU privilege for that command in the database's root file ACL. For some RMU commands on OpenVMS, you must have one or more OpenVMS privileges as well as the appropriate RMU privilege to be able to use the command.

Note that the root file ACL created by default on each Oracle Rdb database controls only your RMU access to the database (by specifying privileges that allow a user or group of users access to specific RMU commands). Root file ACLs do not control your access to the database with SQL statements.

9.5.1 Using Oracle RMU Privileges

When you create a database, Oracle Rdb creates a root file ACL that gives RMU privileges to the creator of the database. Example 9–16, an OpenVMS example, shows the default root file ACL for the creator of the mf_personnel database, a user with a user identifier of [sql,user].

Example 9–16 Displaying Oracle RMU Privileges

```
$ RMU/SHOW PRIVILEGE mf_personnel
Object type: file,  Object name: SQL_USER1:[USER]MF_PERSONNEL.RDB;1,
on 20-JUL-1996 11:12:22.71

      (IDENTIFIER=[SQL,USER],ACCESS=READ+WRITE+CONTROL+RMU$ALTER+RMU$ANALYZE+
      RMU$BACKUP+RMU$CONVERT+RMU$COPY+RMU$DUMP+RMU$LOAD+RMU$MOVE+RMU$OPEN+
      RMU$RESTORE+RMU$SECURITY+RMU$SHOW+RMU$UNLOAD+RMU$VERIFY)
```

Note

Note that the names of the privileges are slightly different on each platform. On OpenVMS, the privileges begin with RMU$; on Digital UNIX, they begin with RMU_.

By default, Oracle Rdb grants the creator of the database all RMU privileges in the root file ACL and grants no privileges to other users. If a user has the RMU$SECURITY or RMU_SECURITY privilege, the user can grant privileges to other users. You use the RMU Set Privilege command to create root file ACLs and to add ACEs to or delete ACEs from a root file ACL.

Example 9–17 assumes that the user with a user identifier of [SQL,USER] created the mf_personnel database. The creator grants RMU privileges to database users. The RMU privileges granted to each type of user depend on the type of RMU access the user needs to the database.

Example 9–17 Setting Privileges for Oracle RMU Commands

```
$! Use the RMU Set Privilege command and the After qualifier to grant
$! the RMU$ANALYZE, RMU$OPEN, RMU$SHOW, and RMU$VERIFY privileges
$! to the user [SQL,USER2], who serves as the database administrator.
$!
$ RMU/SET PRIVILEGE/ACL=(IDENTIFIER=[SQL,USER2],ACCESS=RMU$ANALYZE -
_$ +RMU$OPEN+RMU$VERIFY+RMU$SHOW) -
_$ /AFTER=(IDENTIFIER=[SQL,USER]) mf_personnel.rdb
%RMU-I-MODIFIED, SQL_USER:[USER]MF_PERSONNEL.RDB;1 modified
$!
$! Next, grant the RMU$SECURITY privilege to the user [SQL,USER3].
$! This gives the user the ability to grant other users the appropriate
$! privileges they need for accessing the database with RMU commands.
$! Because both the database creator and user USER3 have the RMU$SECURITY
$! privilege, both can modify the root file ACL for the database:
$!
$ RMU/SET PRIVILEGE/ACL=(IDENTIFIER=[SQL,USER3],ACCESS=RMU$SECURITY) -
_$ /AFTER=(IDENTIFIER=[SQL,USER2]) mf_personnel.rdb
%RMU-I-MODIFIED, SQL_USER:[USER]MF_PERSONNEL.RDB;1 modified
$!
$! Grant the RMU$LOAD and RMU$SHOW privileges to user [RDB,USER4]
$! who will write programs that load data into the database:
$!
```

(continued on next page)

Example 9–17 (Cont.) Setting Privileges for Oracle RMU Commands

```
$ RMU/SET PRIVILEGE/ACL=(IDENTIFIER=[RDB,USER4],ACCESS=RMU$LOAD -
_$ +RMU$SHOW) /AFTER=(IDENTIFIER=[SQL,USER3]) mf_personnel.rdb
%RMU-I-MODIFIED, SQL_USER:[USER]MF_PERSONNEL.RDB;1 modified
$!
$! Grant no privileges to all other users:
$ RMU/SET PRIVILEGE/ACL=(IDENTIFIER=[*,*],ACCESS=NONE) -
_$ /AFTER=(IDENTIFIER=[RDB,USER4]) mf_personnel.rdb
%RMU-I-MODIFIED, SQL_USER:[USER]MF_PERSONNEL.RDB;1 modified
$!
$! The RMU Show Privilege command displays the root file ACL for the
$! mf_personnel database:
$ RMU/SHOW PRIVILEGE mf_personnel.rdb
Object type: file,  Object name: SQL_USER:[USER]MF_PERSONNEL.RDB;1,
on 20-JUL-1996 15:52:17.03
            (IDENTIFIER=[SQL,USER],ACCESS=READ+WRITE+CONTROL+RMU$ALTER+
            RMU$ANALYZE+RMU$BACKUP+RMU$CONVERT+RMU$COPY+RMU$DUMP+RMU$LOAD+
            RMU$MOVE+RMU$OPEN+RMU$RESTORE+RMU$SECURITY+RMU$SHOW+RMU$UNLOAD+
            RMU$VERIFY)
            (IDENTIFIER=[SQL,USER2],ACCESS=RMU$ANALYZE+RMU$OPEN+RMU$VERIFY)
            (IDENTIFIER=[SQL,USER3],ACCESS=RMU$SECURITY)
            (IDENTIFIER=[RDB,USER4],ACCESS=RMU$LOAD+RMU$SHOW)
            (IDENTIFIER=[*,*],ACCESS=NONE)
```

For reference information on the RMU Set Privilege command, see the *Oracle RMU Reference Manual*.

OpenVMS VAX OpenVMS Alpha

Table 9–3 shows the privileges that you must have to use each RMU command. The following list describes the columns listed in Table 9–3:

- The Required RMU Privileges column shows the RMU privileges you must have to use each RMU command. When more than one RMU privilege appears in the Required RMU Privileges column, you pass the RMU privilege check for the specified RMU command if you have *any* of the listed RMU privileges.
- If the RMU command requires a user to have one or more OpenVMS privileges in addition to the appropriate RMU privileges, the OpenVMS privileges are shown in the Required OpenVMS Privileges column of Table 9–3. When more than one OpenVMS privilege is listed in the Required OpenVMS Privileges column, you must have *all* of the listed OpenVMS privileges to pass the OpenVMS privilege check for the RMU command.

The OpenVMS privileges listed in the Required OpenVMS Privileges column are the privileges that are required if RMU has been installed only with the OpenVMS SYSPRV privilege. If the RMU image is also installed with one of the other privileges that appears in the Required OpenVMS Privileges column of the table, all users are automatically treated as if they held that privilege. For example, if RMU is installed with the SYSPRV and the WORLD privilege, you can use the RMU Show System command even if you do not hold the OpenVMS WORLD privilege listed as required in the Required OpenVMS Privileges column.

- The OpenVMS Override Privileges column shows the OpenVMS privileges that allow a user who does not have the appropriate required RMU and OpenVMS privileges to use the command anyway. When more than one OpenVMS privilege is listed in the OpenVMS Override Privileges column, you can use the specified RMU command if you have any of the listed privileges. If the OpenVMS Override Privileges column is blank, you must have the required RMU privilege (and required OpenVMS privilege, if any) to use the RMU command.

Table 9–3 Privileges Required for Oracle RMU Commands

Oracle RMU Command	Required Oracle RMU Privileges	Required OpenVMS Privileges	OpenVMS Override Privileges
Alter	RMU$ALTER[1]		SYSPRV, BYPASS
Analyze Areas	RMU$ANALYZE		SYSPRV, BYPASS
Analyze Cardinality	RMU$ANALYZE		SYSPRV, BYPASS
Analyze Indexes	RMU$ANALYZE		SYSPRV, BYPASS
Analyze Placement	RMU$ANALYZE		SYSPRV, BYPASS
Backup	RMU$BACKUP		SYSPRV, BYPASS
Backup After_Journal	RMU$BACKUP		SYSPRV, BYPASS

[1]You must have the OpenVMS SYSPRV or BYPASS privilege if you use an RMU/ALTER command to change a file name.

(continued on next page)

Table 9–3 (Cont.) Privileges Required for Oracle RMU Commands

Oracle RMU Command	Required Oracle RMU Privileges	Required OpenVMS Privileges	OpenVMS Override Privileges
Backup Plan	RMU$BACKUP		SYSPRV, BYPASS
Checkpoint	RMU$BACKUP, RMU$OPEN		WORLD
Close	RMU$OPEN		WORLD
Collect Optimizer_Statistics	RMU$ANALYZE		SYSPRV, BYPASS
Convert	RMU$CONVERT, RMU$RESTORE		SYSPRV, BYPASS
Copy_Database	RMU$COPY		SYSPRV, BYPASS
Delete Optimizer_Statistics	RMU$ANALYZE		SYSPRV, BYPASS
Dump After_journal	RMU$DUMP		SYSPRV, BYPASS
Dump Areas	RMU$DUMP		SYSPRV, BYPASS
Dump Backup_File	RMU$DUMP, RMU$BACKUP, RMU$RESTORE	READ[2]	BYPASS
Dump Export		READ[3]	BYPASS
Dump Header	RMU$DUMP, RMU$BACKUP, RMU$OPEN		SYSPRV, BYPASS
Dump Lareas	RMU$DUMP		SYSPRV, BYPASS
Dump Recovery_Journal		READ[4]	BYPASS
Dump Snapshots	RMU$DUMP		SYSPRV, BYPASS

[2]You must have OpenVMS READ access for the .rbf file.

[3]You must have OpenVMS READ access for the .rbr or .unl file.

[4]You must have OpenVMS READ access for the .ruj file.

(continued on next page)

Table 9–3 (Cont.) Privileges Required for Oracle RMU Commands

Oracle RMU Command	Required Oracle RMU Privileges	Required OpenVMS Privileges	OpenVMS Override Privileges
Dump Users	RMU$DUMP, RMU$BACKUP, RMU$OPEN		WORLD
Extract	RMU$UNLOAD		SYSPRV, BYPASS
Insert Optimizer Statistics	RMU$ANALYZE		SYSPRV, BYPASS
Load	RMU$LOAD[5]		SYSPRV, BYPASS
Load Audit	RMU$SECURITY		SECURITY, BYPASS
Load Plan	RMU$LOAD		SYSPRV, BYPASS
Monitor Reopen_Log		WORLD, CMKRNL, DETACH, PSWAPM, ALTPRI, SYSGBL, SYSNAM, SYSPRV, BYPASS	SETPRV
Monitor Start		WORLD, CMKRNL, DETACH, PSWAPM, ALTPRI, PRMMBX, SYSGBL, SYSNAM, SYSPRV, BYPASS	SETPRV

[5]You must have OpenVMS WORLD access *in addition to* the RMU$BACKUP, RMU$OPEN, or RMU$SHOW privilege for all databases on your node if you do not specify a database file name.

(continued on next page)

Table 9–3 (Cont.) Privileges Required for Oracle RMU Commands

Oracle RMU Command	Required Oracle RMU Privileges	Required OpenVMS Privileges	OpenVMS Override Privileges
Monitor Stop		WORLD, CMKRNL, DETACH, PSWAPM, ALTPRI, PRMMBX, SYSGBL, SYSNAM, SYSPRV, BYPASS	SETPRV
Move_Area	RMU$MOVE		SYSPRV, BYPASS
Open	RMU$OPEN		WORLD
Optimize After_Journal	RMU$BACKUP, RMU$RESTORE		SYSPRV, BYPASS
Recover	RMU$RESTORE		SYSPRV, BYPASS
Repair	RMU$ALTER		SYSPRV, BYPASS
Resolve	RMU$RESTORE		SYSPRV, BYPASS
Restore	RMU$RESTORE		SYSPRV, BYPASS
Restore Only_Root	RMU$RESTORE		SYSPRV, BYPASS
Server After_Journal Reopen_ Output	RMU$OPEN		WORLD
Server After_Journal Start	RMU$OPEN		WORLD
Server After_Journal Stop	RMU$OPEN		WORLD
Server Backup_Journal Resume	RMU$OPEN		WORLD
Server Backup_Journal Suspend	RMU$OPEN		WORLD

(continued on next page)

Table 9–3 (Cont.) Privileges Required for Oracle RMU Commands

Oracle RMU Command	Required Oracle RMU Privileges	Required OpenVMS Privileges	OpenVMS Override Privileges
Set After_Journal	RMU$ALTER, RMU$BACKUP, RMU$RESTORE		SYSPRV, BYPASS
Set Audit	RMU$SECURITY		SECURITY, BYPASS
Set Corrupt_Pages	RMU$ALTER, RMU$BACKUP, RMU$RESTORE		SYSPRV, BYPASS
Set Privilege	RMU$SECURITY		SECURITY, BYPASS
Show After_Journal	RMU$BACKUP, RMU$RESTORE, RMU$VERIFY		SYSPRV, BYPASS
Show Audit	RMU$SECURITY		SECURITY, BYPASS
Show Corrupt_Pages	RMU$BACKUP, RMU$RESTORE, RMU$VERIFY		SYSPRV, BYPASS
Show Locks		WORLD	
Show Optimizer_Statistics	RMU$ANALYZE, RMU$SHOW		SYSPRV, BYPASS
Show Privilege	RMU$SECURITY		SECURITY, BYPASS
Show Statistics[8]	RMU$SHOW		SYSPRV, BYPASS, WORLD
Show System		WORLD	
Show Users[5]	RMU$SHOW, RMU$BACKUP, RMU$OPEN		WORLD

[5]You must have OpenVMS WORLD access *in addition to* the RMU$BACKUP, RMU$OPEN, or RMU$SHOW privilege for all databases on your node if you do not specify a database file name.

[8]You must have the OpenVMS WORLD privilege if you use this command to display statistics about other users (as opposed to database statistics). You must have both the OpenVMS WORLD and BYPASS privileges if you use this command to update fields in the Database Dashboard.

(continued on next page)

Table 9–3 (Cont.) Privileges Required for Oracle RMU Commands

Oracle RMU Command	Required Oracle RMU Privileges	Required OpenVMS Privileges	OpenVMS Override Privileges
Show Version			
Unload	RMU$UNLOAD[6]		SYSPRV, BYPASS
Verify	RMU$VERIFY[7]		SYSPRV, BYPASS

[6]The appropriate Oracle Rdb privileges for accessing the database tables involved are also required.
[7]You must also have the SQL DBADM privilege.

♦

Table 9–4 shows the privileges that you must have to use each RMU command on Digital UNIX.

Table 9–4 Privileges Required for RMU Commands on Digital UNIX

RMU Command	Required RMU Privileges	Override Account
Alter	RMU_ALTER	dbsmgr, superuser
Analyze	RMU_ANALYZE	dbsmgr, superuser
Analyze Cardinality	RMU_ANALYZE	dbsmgr, superuser
Analyze Indexes	RMU_ANALYZE	dbsmgr, superuser
Analyze Areas	RMU_ANALYZE	dbsmgr, superuser
Analyze Placement	RMU_ANALYZE	dbsmgr, superuser
Backup	RMU_BACKUP	dbsmgr, superuser
Backup After_Journal	RMU_BACKUP	dbsmgr, superuser
Checkpoint	RMU_BACKUP	dbsmgr, superuser
Close	RMU_OPEN	dbsmgr, superuser
Collect Optimizer Statistics	RMU_ANALYZE	dbsmgr, superuser
Convert	RMU_CONVERT, RMU_RESTORE	dbsmgr, superuser
Copy_Database	RMU_COPY	dbsmgr, superuser

(continued on next page)

Table 9–4 (Cont.) Privileges Required for RMU Commands on Digital UNIX

RMU Command	Required RMU Privileges	Override Account
Delete Optimizer Statistics	RMU_ANALYZE	dbsmgr, superuser
Dump After_Journal	RMU_DUMP	dbsmgr, superuser
Dump Areas	RMU_DUMP	dbsmgr, superuser
Dump Backup_File	RMU_BACKUP, RMU_DUMP, RMU_RESTORE[1]	dbsmgr, superuser
Dump Export	See footnote.[2]	dbsmgr, superuser
Dump Header	RMU_BACKUP, RMU_DUMP, RMU_OPEN	dbsmgr, superuser
Dump Lareas	RMU_DUMP	dbsmgr, superuser
Dump Recovery_Journal	See footnote.[3]	dbsmgr, superuser
Dump Snapshots	RMU_DUMP	dbsmgr, superuser
Dump Users	RMU_BACKUP, RMU_DUMP, RMU_OPEN	dbsmgr, superuser
Extract	RMU_UNLOAD	dbsmgr, superuser
Help	None	None
Insert Optimizer Statistics	RMU_ANALYZE	dbsmgr, superuser
Load	RMU_LOAD	dbsmgr, superuser
Load Plan	RMU_LOAD	dbsmgr, superuser
Monitor Reopen_Log	Not applicable	dbsmgr, superuser
Monitor Start	Not applicable	dbsmgr, superuser
Monitor Stop	Not applicable	dbsmgr, superuser
Move_Area	RMU_MOVE	dbsmgr, superuser
Open	RMU_OPEN	dbsmgr, superuser
Optimize After_Journal	RMU_BACKUP, RMU_RESTORE	dbsmgr, superuser
Recover	RMU_RESTORE	dbsmgr, superuser
Recover Resolve	RMU_RESTORE	dbsmgr, superuser
Repair	RMU_ALTER	dbsmgr, superuser

[1] You must have Digital UNIX read access to the .rbf file.

[2] You must have Digital UNIX read access to the .rbr file.

[3] You must have Digital UNIX read access to the .ruj file.

(continued on next page)

Table 9–4 (Cont.) Privileges Required for RMU Commands on Digital UNIX

RMU Command	Required RMU Privileges	Override Account
Restore	RMU_RESTORE	dbsmgr, superuser
Restore Only_Root	RMU_RESTORE	dbsmgr, superuser
Server After_Journal Reopen_Log	RMU_OPEN	dbsmgr, superuser
Server After_Journal Start	RMU_OPEN	dbsmgr, superuser
Server After_Journal Stop	RMU_OPEN	dbsmgr, superuser
Server Backup_Journal Resume	RMU_OPEN	dbsmgr, superuser
Server Backup_Journal Suspend	RMU_OPEN	dbsmgr, superuser
Set After_Journal	RMU_ALTER, RMU_BACKUP, RMU_RESTORE	dbsmgr, superuser
Set Corrupt_Pages	RMU_ALTER, RMU_BACKUP, RMU_RESTORE	dbsmgr, superuser
Set Privilege	RMU_SECURITY	dbsmgr, superuser
Show After_Journal	RMU_BACKUP, RMU_RESTORE, RMU_VERIFY	dbsmgr, superuser
Show Corrupt_Pages	RMU_BACKUP, RMU_RESTORE, RMU_VERIFY,	dbsmgr, superuser
Show Locks	Not applicable	dbsmgr, superuser
Show Optimizer Statistics	RMU_ANALYZE, RMU_SHOW	dbsmgr, superuser
Show Privilege	RMU_SECURITY	dbsmgr, superuser
Show Statistics	RMU_SHOW	dbsmgr, superuser
Show System	Not applicable	dbsmgr, superuser
Show Users	RMU_BACKUP, RMU_OPEN, RMU_SHOW	dbsmgr, superuser
Show Version	None	None
Unload	RMU_UNLOAD	dbsmgr, superuser
Verify	RMU_VERIFY	dbsmgr, superuser

♦

9.5.2 Using Oracle RMU Privileges with Databases Created with Version 4.1 or Earlier

When you use RMU Convert or SQL IMPORT to convert a database created with Version 4.1 or earlier, an ACL is placed on the root file of the database. In creating this initial root file ACL for the database, Oracle Rdb considers information from the Oracle Rdb internal database ACL and from any previously existing root file ACL on the original database. If these ACLs exist, Oracle Rdb examines each entry within the ACL and attempts to translate the entry into one with the appropriate set of RMU privileges within the new root file ACL.

When examining the internal database ACL, Oracle Rdb follows two general rules:

1. Identifiers with the DBADM privilege receive a root file ACE granting all RMU privileges except RMU$SECURITY privilege.
2. Identifiers with the SECURITY privilege receive a root file ACE granting the RMU$SECURITY privilege.

This feature provides a measure of backward compatibility for RMU command access based on the methods used to check security for RMU commands in previous versions of Oracle Rdb.

It is unlikely that the resulting root file ACL will meet all the needs of your database users for executing RMU commands. After the database has been converted to V4.2 or higher, you should modify the root file ACL to grant access to users for RMU commands that they need to perform. The RMU Set Privilege command allows you to edit the root file ACL to grant or revoke RMU privileges as required for each database.

See the *Oracle RMU Reference Manual* for a description of the RMU Set Privilege command and the RMU Show Privilege command.

9.6 Restricting Database Creation

OpenVMS VAX OpenVMS Alpha

You might wish to prevent users from creating their own databases, but still allow those users the right to use data manipulation statements on already existing databases. There is no Oracle Rdb privilege limiting the right to create a database. However, you can define the logical name RDBVMS$CREATE_DB and a rights identifier of the same name to restrict the creation of databases.

Caution

When you define this logical name, other installed products and third-party products will not be able to use Oracle Rdb to create Oracle Rdb

databases. Therefore, you must deassign this logical name whenever users of such products need to create an Oracle Rdb database.

To restrict the creation of Oracle Rdb databases, you must first define the SYSTEM/EXECUTIVE logical name RDBVMS$CREATE_DB. You can use any name for the equivalence name, although you may want to use a meaningful name.

Use the rights identifier RDBVMS$CREATE_DB to control which users are allowed to create databases. When a user enters a CREATE DATABASE statement, Oracle Rdb checks for the RDBVMS$CREATE_DB logical name in the LMN$SYSTEM_TABLE. If you have not defined the logical name, Oracle Rdb allows all users to create a database. If you have defined the RDBVMS$CREATE_DB logical name, Oracle Rdb checks whether or not the user (or current process) holds the RDBVMS$CREATE_DB identifier. If this identifier is in the rights list belonging to the user or process, that user or process is allowed to create a database. Otherwise, the user receives a "privilege violation" error message.

Use the OpenVMS AUTHORIZE utility to add the RDBVMS$CREATE_DB rights identifier. Then grant this identifier to those users who *should* have the ability to create databases.

Some restrictions on the use of the RDBVMS$CREATE_DB identifier are:

- After you define the RDBVMS$CREATE_DB logical name, the creation of all Oracle Rdb databases is controlled by this mechanism.
- Users with OpenVMS SYSPRV or BYPASS privilege can bypass the enforcement of this logical.
- Use of the RDBVMS$CREATE_DB logical name and identifier does not prevent users from re-creating databases using the operating system backup or copy commands. Therefore, use standard operating system protections on any directories containing Oracle Rdb databases.

♦

9.7 Securing Shareable Oracle Rdb Definitions in the Repository

OpenVMS VAX OpenVMS Alpha

Repository commands that set protection for the repository shareable entities are separate from the SQL statements that set protection for an Oracle Rdb database.

In the default repository protection scheme for entities (columns and rows), *owner* is granted all privileges, including DBCTRL, and *world* is granted the SHOW privilege only.

For information about protecting shareable repository definitions, see the Oracle CDD/Repository documentation. ♦

10
Using Oracle Rdb with Oracle CDD/Repository

This chapter provides information about designing, defining, and maintaining an Oracle Rdb database using Oracle CDD/Repository. It presents the following material:

- An overview of repository use in the design of an Oracle Rdb database
- Scenarios for using and tracking shareable Oracle Rdb definitions

For more detailed information on repository usage, refer to the Oracle CDD/Repository documentation.

10.1 Overview of the Repository

A **repository** is a software product that allows you to create, analyze, and administer metadata. The repository provides the ability to define, track, and control Oracle Rdb metadata using one centralized mechanism.

An **element**, used in reference to the repository, is a piece of information that represents real-world objects, such as files, data definitions, and databases. A repository element definition contains various properties, or individual characteristics. It can also be related to other repository elements. The most commonly used elements are fields (domains), records (tables), and databases.

A **field definition** is the smallest unit of metadata that can be created and accessed. Field definitions typically include information about the data type, size, and other optional properties. The repository keeps track of definition usage at the field level. Therefore, you can easily show which repository elements make use of a particular field definition. When a field definition changes, you can identify which elements are affected by the change and which elements need to be redefined to access the changed field. This ability to track elements is known as **pieces tracking**. For more information on pieces tracking, see the repository documentation.

As an Oracle Rdb user, you gain access to repository metadata through the Common Dictionary Operator (CDO) utility. You can access CDO through a menu-driven editor and the DCL command REPOSITORY OPERATOR. Using CDO, you can create, modify, delete, or inquire about metadata elements to which you have the appropriate access privileges.

In the repository and Oracle Rdb environments, you can use either CDO or SQL to define fields (domains) and records (tables). This chapter contains examples using both CDO and SQL. However, working from CDO has certain advantages because you can define, share, track, and control records and fields independently of a particular database. To be able to share records and fields across applications and databases, you must define them using CDO.

10.1.1 Repository Naming Conventions

Every repository definition has a full name that uniquely identifies it. Each definition consist of three parts: the repository anchor, the repository path, and a version number. The following SQL statement attaches to the Oracle Rdb database DEPT1 using its full path name:

```
ATTACH 'PATHNAME SYS$COMMON:[REPOSITORY]PERS.DEPT1';
                 ^---------------------^^--^ ^---^
                      anchor            dir db-name
```

An **anchor** is the name of the operating system directory where the repository is stored. SYS$COMMON:[REPOSITORY] is an example of an anchor, and PERS is a directory in the repository that contains the database definition DEPT1.

A **path** is similar to a file specification. It consists of a list of repository directory names separated by periods or slashes and terminates with the name of an element. **Repository directories** are similar to operating system directories; you use them to organize repository definitions. The path name specifies the path to the desired repository elements.

You can represent each element in DEPT1 using a full path name. For example, you can represent SALARY_HISTORY as:

```
SYS$COMMON:[REPOSITORY]PERS.DEPT1.SALARY_HISTORY
```

The path name consists of the repository directory name, the database name, and the record (table) name.

A **version** is similar to an OpenVMS file version. The version number is always preceded by a semicolon (;). The repository allows you to create multiple versions of a repository element.

10.1.2 Using CDO

CDO contains a variety of commands that are useful for Oracle Rdb users. CDO provides commands that:

- Define metadata. These commands perform the same tasks as the SQL CREATE statements and are described in Section 10.3.
- Track metadata. The commands listed in Table 10–1 allow you to track which databases use various repository definitions. These commands also allow you to analyze the impact of possible changes to definitions.
- Set protection on metadata. These CDO commands parallel the use of SQL statements that set protection on data.
- Delete or modify metadata.

Table 10–1 Summary of CDO Pieces Tracking Commands

CDO Command	Tracking Function
SHOW NOTICES	Displays any notices attached to the specified definition. The notice indicates that an inconsistency exists between the specified definition and a related definition, or that a new version exists. Supporting software products can read the message and generate a warning to the user.
SHOW UNUSED	Displays element definitions that are not used by any other definition. This helps you decide when it is safe to purge or delete repository definitions.
SHOW USES	Displays all the element definitions that use the specified definition. This helps you consider the impact of changing the definition by creating a new version.
SHOW USED_BY	Displays the definitions that are used by the specified definition.
SHOW WHAT_IF	Displays a list of the repository definitions that would be flagged with a message after a definition is changed with the CHANGE command. This helps you to consider the impact of changing the original definition.
SHOW FIELD	Displays information about a repository element. Use the FROM DATABASE clause of the SHOW FIELD command when a field (domain) does not have a directory entry. From CDO, you can use the ENTER command to give a field a CDO directory name.

(continued on next page)

Table 10–1 (Cont.) Summary of CDO Pieces Tracking Commands

CDO Command	Tracking Function
SHOW RECORD	Displays information about a repository element. Use the FROM DATABASE clause of the SHOW FIELD command when a record (table) does not have a directory entry. From CDO, you can use the ENTER command to give a record a CDO directory name.

10.1.3 Criteria for Using the Repository with Oracle Rdb Databases

You should consider implementing database definitions by first defining them in the repository if you have a requirement to:

- Track definitions
- Share definitions

Not all database designs benefit equally from a centrally placed repository. You do not need to create data definitions in the repository if:

- Data definitions are local to one application and are likely to remain so.
- Requirements for controlling definitions are met by the existing protection placed on metadata.

Figure 10–1 illustrates the repository in a centralized database design. Figure 10–1 shows a prototype database, where two department-wide production databases, as well as user applications, share the same field and record definitions through the repository. Because the metadata definitions reside in the repository and not in any particular database, they can be maintained independently and shared by databases with different uses.

Figure 10–1 Centralized Design with the Repository

DEPT3

Prototype database

CDD/Repository

User applications

DEPT1

Definitions

DEPT2

Production database

Production database

NU–2078A–RA

You can use the repository to establish corporate and department standard fields, such as ID_NUMBER and STANDARD_DATE. Using CDO, you can define a shareable field (domain) and base other fields on it, as the following example shows:

```
CDO> DEFINE FIELD ID_NUMBER
cont>   DESCRIPTION IS 'Corporate Standard 12-01-93'
cont>   DATATYPE IS TEXT SIZE IS 5.
CDO> DEFINE FIELD EMPLOYEE_ID
cont>   AUDIT IS 'copied from Corporate Standard 12-01-93'
cont>   BASED ON ID_NUMBER.
CDO> DEFINE FIELD STANDARD_DATE
cont>   DESCRIPTION IS 'Corporate Standard 12-01-93'
cont>   DATATYPE IS DATE.
```

After defining shareable fields (domains) and records (tables) using CDO, you can share repository definitions across multiple Oracle Rdb databases by issuing SQL statements to define columns and tables from the repository using path names. This process is described in Section 10.3.

10.2 Deciding Whether to Require the Repository

If your application uses definitions from many Oracle Rdb databases, you should decide whether to require the use of the repository. Consider the following trade-offs:

- Field and record definitions in the repository are stored in a format that can be interpreted by other software products.

- You cannot rename a repository definition as you include it in a database.
- You can use CDO or SQL to change shareable definitions created with CDO.
- Whenever you change database definitions without a corresponding change to the repository definitions, inconsistencies can arise between the repository and database copies of the definitions. Oracle Rdb users who attach to a database using the PATHNAME clause receive a message that repository elements have changed, as the following example shows:

```
SQL> ATTACH 'PATHNAME DEPT1';
%SQL-I-DIC_DB_CHG1, A repository definition used by database
SYS$COMMON:[REPOSITORY]PERS.DEPT1;1 has changed
-SQL-I-DIC_DB_CHG2, Use the INTEGRATE statement to resolve any differences
between the repository and the database
%CDD-I-MESS, element has messages
```

If you decide to require the use of your repository in your database design, use the DICTIONARY IS REQUIRED clause in the CREATE DATABASE statement.

If your database requires all metadata updates to be maintained in the repository, a user who attaches to that database using the FILENAME clause and attempts to manipulate database definitions receives the following messages:

```
SQL> CREATE DOMAIN EMPLOYEE_NAME IS CHAR (10);
%RDB-E-NO_META_UPDATE, metadata update failed
-RDMS-F-CDDISREQD, CDD required for metadata updates is not being maintained
```

Storing database definitions in the repository provides a central source of shareable field (domain) and record (table) definitions. To avoid data definition inconsistencies, attach to the database using the repository path name. In this way, the database definitions are always available to other products that use the repository. For more information about whether a product uses the repository, see the documentation for that product.

10.3 Creating New Repository Definitions

To use the repository with SQL, take the following steps:

1. Use CDO to define the shareable fields (domains) and records (tables) for an Oracle Rdb database.
2. Use SQL to create the database.
3. Use SQL to copy the definitions of the domains and tables from the repository to the database.

4. Use SQL to create local Oracle Rdb views and indexes.
5. Use SQL to include or integrate any changed definitions into the repository.
6. Share the definitions as needed among other Oracle Rdb databases.

The following series of examples illustrate how you might use a central repository to create shareable database definitions for multiple Oracle Rdb databases. These examples assume:

- A repository whose anchor is located at SYS$COMMON:[REPOSITORY].
- A repository directory, called PERS, that contains the database definitions.

If the directory PERS does not exist, the following example shows how to create it and set default to it:

```
$ REPOSITORY  OPERATOR
Welcome to CDO V2.3
The CDD/Repository V5.3 User Interface
Type HELP for help
CDO> SET DEF SYS$COMMON:[REPOSITORY]
CDO> SHOW DEF
SYS$COMMON:[REPOSITORY]
CDO> DEFINE DIRECTORY PERS.
```

If the directory PERS already exists, the following example demonstrates how to check your default repository directory and invoke CDO:

```
$ ! To begin, show your default repository directory and invoke CDO.
$ SHOW LOGICAL CDD$DEFAULT
 "CDD$DEFAULT" = "SYS$COMMON[REPOSITORY]PERS" (LNM$PROCESS_TABLE)

$ REPOSITORY OPERATOR
Welcome to CDO V2.3
The CDD/Repository V5.3 User Interface
Type HELP for help
CDO> !
CDO> SHOW DEFAULT
CDD$DEFAULT
    = SYS$COMMON:[REPOSITORY]PERS
```

Example 10–1 shows how to define shareable field definitions using CDO.

Example 10–1 Defining Shareable Fields

```
CDO> ! Define a shareable field (domain), ID_NUMBER, and base another
CDO> ! field, EMPLOYEE_ID, on it:
CDO> !
CDO> DEFINE FIELD ID_NUMBER
cont>   DESCRIPTION IS 'Corporate Standard 12-01-93'
cont>   DATATYPE IS TEXT SIZE IS 5.
CDO> DEFINE FIELD EMPLOYEE_ID
cont>   AUDIT IS 'copied from Corporate Standard 12-01-93'
cont>   BASED ON ID_NUMBER.
CDO> !
CDO> ! Define a field, STANDARD_DATE, and base other fields (domains) on this
CDO> ! field as a corporate standard:
CDO> !
CDO> DEFINE FIELD STANDARD_DATE
cont>   DESCRIPTION IS 'Corporate Standard 12-02-93'
cont>   DATATYPE IS DATE.
CDO> DEFINE FIELD SALARY_START
cont>   DESCRIPTION IS 'copied from Corporate Standard 12-02-93'
cont>   BASED ON STANDARD_DATE.
CDO> DEFINE FIELD SALARY_END
cont>   DESCRIPTION IS 'copied from Corporate Standard 12-02-93'
cont>   BASED ON STANDARD_DATE.
CDO> DEFINE FIELD SALARY_AMOUNT
cont>   DATATYPE IS SIGNED QUADWORD SCALE -2.
CDO> DEFINE FIELD BIRTHDAY
cont>   DESCRIPTION 'copied from Corporate Standard 12-02-93'
cont>   BASED ON STANDARD_DATE.
CDO> !
CDO> ! Define other fields (domains) for the EMPLOYEES record (table):
CDO> !
CDO> DEFINE FIELD LAST_NAME
cont>   DATATYPE IS TEXT SIZE IS 14.
CDO> DEFINE FIELD FIRST_NAME
cont>   DATATYPE IS TEXT SIZE IS 10.
CDO> DEFINE FIELD MIDDLE_INITIAL
cont>   DATATYPE IS TEXT SIZE IS 1.
CDO> DEFINE FIELD ADDRESS_DATA_1
cont>   DATATYPE IS TEXT SIZE IS 25.
CDO> DEFINE FIELD ADDRESS_DATA_2
cont>   DATATYPE IS TEXT SIZE IS 25.
CDO> DEFINE FIELD CITY
cont>   DATATYPE IS TEXT SIZE IS 20.
CDO> DEFINE FIELD STATE
cont>   DATATYPE IS TEXT SIZE IS 2.
CDO> DEFINE FIELD POSTAL_CODE
cont>   DATATYPE IS TEXT SIZE IS 5.
```

(continued on next page)

Example 10–1 (Cont.) Defining Shareable Fields

```
CDO> DEFINE FIELD SEX
cont>   DATATYPE IS TEXT SIZE IS 1.
CDO> DEFINE FIELD STATUS_CODE
cont>   DATATYPE IS TEXT SIZE IS 1.
```

Example 10–2 shows how to use the CDO DIRECTORY command to verify that all of the fields are defined.

Example 10–2 Checking Field Definitions

```
CDO> DIRECTORY

Directory SYS$COMMON:[REPOSITORY]PERS

ADDRESS_DATA_1(1)                               FIELD
ADDRESS_DATA_2(1)                               FIELD
BIRTHDAY(1)                                     FIELD
CITY(1)                                         FIELD
EMPLOYEE_ID(1)                                  FIELD
FIRST_NAME(1)                                   FIELD
ID_NUMBER(1)                                    FIELD
LAST_NAME(1)                                    FIELD
MIDDLE_INITIAL(1)                               FIELD
POSTAL_CODE(1)                                  FIELD
SALARY_AMOUNT(1)                                FIELD
SALARY_END(1)                                   FIELD
SALARY_START(1)                                 FIELD
SEX(1)                                          FIELD
STANDARD_DATE(1)                                FIELD
STATE(1)                                        FIELD
STATUS_CODE(1)                                  FIELD
CDO> !
CDO> ! If you require more information on an element, use the
CDO> ! SHOW FIELD field-name /ALL/FULL command:
CDO> !
CDO> SHOW FIELD CITY/ALL/FULL
Definition of field CITY
|   acl
(IDENTIFIER=[SQL,DBA],ACCESS=READ+WRITE+MODIFY+ERASE+SHOW+DEFINE+CHANGE+DELETE
    +CONTROL+OPERATOR+ADMINISTRATOR)
(IDENTIFIER=[*,*],ACCESS=READ+WRITE+MODIFY+ERASE+SHOW+OPERATOR+ADMINISTRATOR)
|   Created time            28-SEP-1995 09:55:23.66
|   Modified time           28-SEP-1995 09:55:23.66
|   Owner                   [SQL,DBA]
|   Status                  Available
|   Freeze time             28-SEP-1995 09:55:23.66
```

(continued on next page)

Example 10–2 (Cont.) Checking Field Definitions

```
│   Controlled              No
│   Allow concurrent        No
│   Datatype                text size is 20 characters
│   │   History entered by SANDERSON ([SQL,DBA])
│   │    using CDO V2.3
│   │    to CREATE definition on 28-SEP-1995 09:55:23.63
CDO> !
CDO> ! The SHOW USES command lists the parents of STANDARD_DATE
CDO> ! or the elements that use the STANDARD_DATE field.
CDO> !
CDO> DEFINE FIELD START_DATE BASED ON STANDARD_DATE.
CDO> SHOW USES STANDARD_DATE
Owners of SYS$COMMON:[REPOSITORY]PERS.STANDARD_DATE(1)
│   SYS$COMMON:[REPOSITORY]PERS.SALARY_START(1)  (Type : FIELD)
│   │   via CDD$DATA_ELEMENT_BASED_ON
│   SYS$COMMON:[REPOSITORY]PERS.SALARY_END(1)  (Type : FIELD)
│   │   via CDD$DATA_ELEMENT_BASED_ON
│   SYS$COMMON:[REPOSITORY]PERS.BIRTHDAY(1)  (Type : FIELD)
│   │   via CDD$DATA_ELEMENT_BASED_ON
│   SYS$COMMON:[REPOSITORY]PERS.START_DATE(1)  (Type : FIELD)
│   │   via CDD$DATA_ELEMENT_BASED_ON
```

Example 10–3 shows how to create record definitions for Oracle Rdb tables.

Example 10–3 Defining Records

```
CDO> !  Define a record for the Oracle Rdb table EMPLOYEES.
CDO> !
CDO> DEFINE RECORD EMPLOYEES DESCRIPTION IS
cont>   /* CORPORATE EMPLOYEE INFORMATION */.
cont>   EMPLOYEE_ID.
cont>   LAST_NAME.
cont>   FIRST_NAME.
cont>   MIDDLE_INITIAL.
cont>   ADDRESS_DATA_1.
cont>   ADDRESS_DATA_2.
cont>   CITY.
cont>   STATE.
cont>   POSTAL_CODE.
cont>   BIRTHDAY.
cont>   SEX.
cont>   STATUS_CODE.
cont> END EMPLOYEES RECORD.
```

(continued on next page)

Example 10–3 (Cont.) Defining Records

```
CDO> !
CDO> ! Define a record for the Oracle Rdb table SALARY_HISTORY.
CDO> !
CDO> DEFINE RECORD SALARY_HISTORY
cont>   DESCRIPTION IS /* INFO ABOUT EACH JOB HELD */.
cont>   EMPLOYEE_ID.
cont>   SALARY_AMOUNT.
cont>   SALARY_START.
cont>   SALARY_END.
cont> END SALARY_HISTORY RECORD.
CDO> !
CDO> ! Display the definition for the EMPLOYEES record (table).
CDO> !
CDO> SHOW RECORD EMPLOYEES
Definition of record EMPLOYEES
|     Description             'CORPORATE EMPLOYEE INFORMATION'
|     Contains field          EMPLOYEE_ID
|     Contains field          LAST_NAME
|     Contains field          FIRST_NAME
|     Contains field          MIDDLE_INITIAL
|     Contains field          ADDRESS_DATA_1
|     Contains field          ADDRESS_DATA_2
|     Contains field          CITY
|     Contains field          STATE
|     Contains field          POSTAL_CODE
|     Contains field          BIRTHDAY
|     Contains field          SEX
|     Contains field          STATUS_CODE
CDO> !
CDO> ! Display the definition for the SALARY_HISTORY record (table).
CDO> !
CDO> SHOW RECORD SALARY_HISTORY
Definition of record SALARY_HISTORY
|     Description             'INFO ABOUT EACH JOB HELD'
|     Contains field          EMPLOYEE_ID
|     Contains field          SALARY_AMOUNT
|     Contains field          SALARY_START
|     Contains field          SALARY_END
```

Next, you can use these definitions in the repository to create domains and tables in Oracle Rdb databases using SQL. The EMPLOYEES and SALARY_HISTORY record (table) definitions you create using CDO can be shared among many Oracle Rdb databases.

Example 10–4 defines two departmental databases, DEPT1 and DEPT2, that share the EMPLOYEES and SALARY_HISTORY record (table) definitions created using CDO.

The DEPT1 database is defined using a relative path name. The DEPT2 database is defined using the full path name, including the anchor where the repository is stored.

Example 10–4 Using CDO Definitions to Create an Oracle Rdb Database with SQL

```
SQL> -- Create the database definition for DEPT1 using a relative path name.
SQL> --
SQL> CREATE DATABASE FILENAME DEPT1
cont>              PATHNAME DEPT1
cont>              DICTIONARY IS REQUIRED;
SQL> DISCONNECT DEFAULT;
SQL> --
SQL> -- Create the database definition for DEPT2 using a full path name.
SQL> --
SQL> CREATE DATABASE FILENAME DEPT2
cont>              PATHNAME SYS$COMMON:[REPOSITORY]PERS.DEPT2;
SQL> DISCONNECT DEFAULT;
SQL> --
SQL> -- Attach to the DEPT1 database using a PATHNAME.
SQL> --
SQL> ATTACH 'PATHNAME SYS$COMMON:[REPOSITORY]PERS.DEPT1';
SQL> --
SQL> -- Create the EMPLOYEES table using the repository definitions.
SQL> --
SQL> CREATE TABLE
cont>       FROM SYS$COMMON:[REPOSITORY]PERS.EMPLOYEES;
SQL> --
SQL> -- Create the SALARY_HISTORY table using the repository definitions.
SQL> --
SQL> CREATE TABLE
cont>       FROM SYS$COMMON:[REPOSITORY]PERS.SALARY_HISTORY;
SQL> --
SQL> -- Display the two tables just created.
SQL> --
SQL> SHOW TABLES;
User tables in database with pathname SYS$COMMON:[REPOSITORY]PERS.DEPT1;1
     EMPLOYEES
     SALARY_HISTORY
SQL> --
SQL> -- Display the EMPLOYEES table definition.
SQL> --
SQL> SHOW TABLE (COLUMNS) EMPLOYEES;
Information for table EMPLOYEES

CDD Pathname:   SYS$COMMON:[REPOSITORY]PERS.EMPLOYEES;1
```

(continued on next page)

Example 10–4 (Cont.) Using CDO Definitions to Create an Oracle Rdb Database with SQL

```
Comment on table EMPLOYEES:
CORPORATE EMPLOYEE INFORMATION

Columns for table EMPLOYEES:

Column Name                     Data Type        Domain
-----------                     ---------        ------
EMPLOYEE_ID                     CHAR(5)          EMPLOYEE_ID
LAST_NAME                       CHAR(14)         LAST_NAME
FIRST_NAME                      CHAR(10)         FIRST_NAME
MIDDLE_INITIAL                  CHAR(1)          MIDDLE_INITIAL
ADDRESS_DATA_1                  CHAR(25)         ADDRESS_DATA_1
ADDRESS_DATA_2                  CHAR(25)         ADDRESS_DATA_2
CITY                            CHAR(20)         CITY
STATE                           CHAR(2)          STATE
POSTAL_CODE                     CHAR(5)          POSTAL_CODE
BIRTHDAY                        DATE             BIRTHDAY
SEX                             CHAR(1)          SEX
STATUS_CODE                     CHAR(1)          STATUS_CODE

SQL> COMMIT;
SQL> DISCONNECT DEFAULT;
SQL> EXIT;
```

Should you make an error when using SQL to create a table or other database objects, issue a ROLLBACK statement. After you are satisfied with the database objects you created, give them the appropriate protection rights, and commit the transaction.

Use SQL to set protection on any data, and use CDO to set protection on any metadata. In the default CDO protection scheme, *owner* is granted all privileges, including CONTROL, while *world* is granted the SHOW privilege only.

After committing the transaction, you can track any modifications to the Oracle Rdb domains (fields) and tables (records) through the repository, using CDO pieces tracking commands, such as the SHOW USES command. See Table 10–1 for a brief description of each of the CDO pieces tracking commands.

10.4 Defining Record-Level Constraints in the Repository

You can define constraints in the repository. When you do, you should give a name to each constraint. If you do not specify a constraint name, CDO assigns a random name to your constraint, thus making it difficult to distinguish the constraint in subsequent error messages.

The default evaluation time for constraints in CDO is NOT DEFERRABLE. This means constraints are evaluated at statement time and not commit time.

Before you can define a foreign key constraint, the record and field referenced by the foreign key must be defined in the repository.

Example 10–5 shows how to define a record with constraints. In this example, the record (table) PARTS is defined with the following constraints:

- Primary key called PARTS_PMK
- Unique key called PARTS_UNQ
- Check constraint PART_CST
- Foreign key PART_FRK

This example assumes that OTHER_PARTS record and OTHER_PARTS_ID field have been previously defined in the repository. It begins with defining the fields and the record in the repository using the Common Dictionary Operator utility.

Example 10–5 Creating Record-Level Constraints

```
$ ! Define CDD$DEFAULT:
$ !
$ DEFINE CDD$DEFAULT SYS$COMMON:[REPOSITORY]TABLE_TEST
$ !
$ REPOSITORY OPERATOR
Welcome to CDO V2.3
The CDD/Repository V5.3 User Interface
Type HELP for help
CDO> !
CDO> ! Create the field definitions for the PARTS record:
CDO> !
CDO> DEFINE FIELD PART_NO DATATYPE IS SIGNED WORD.
CDO> DEFINE FIELD PART_ID DATATYPE IS SIGNED LONGWORD.
CDO> DEFINE FIELD PART_ID_USED_IN DATATYPE IS SIGNED LONGWORD.
CDO> DEFINE FIELD PART_QUANT DATATYPE IS SIGNED WORD.
```

(continued on next page)

Example 10–5 (Cont.) Creating Record-Level Constraints

```
CDO> !
CDO> ! Create the PARTS record definition by first defining the constraints
CDO> ! and then including the field definitions just created.  Note that
CDO> ! CDO creates the constraints as NOT DEFERRABLE.
CDO> !
CDO> DEFINE RECORD PARTS
cont>    CONSTRAINT PARTS_PMK PRIMARY KEY PART_ID
cont>    CONSTRAINT PARTS_UNQ UNIQUE PART_NO
cont>    CONSTRAINT PART_CST CHECK
cont>        (ANY P IN PARTS WITH (PART_ID IN
cont>        PARTS = PART_ID_USED_IN IN P))
cont>    CONSTRAINT PART_FRK
cont>        FOREIGN KEY PART_ID REFERENCES OTHER_PARTS OTHER_PART_ID.
cont>    PART_NO.
cont>    PART_ID.
cont>    PART_ID_USED_IN.
cont>    PART_QUANT.
cont> END.
CDO>  !
CDO>  ! Display the record PARTS.
CDO>  !
CDO>  SHOW RECORD PARTS/FULL
Definition of record PARTS
|   Contains field          PART_NO
|   |   Datatype                signed word
|   Contains field          PART_ID
|   |   Datatype                signed longword
|   Contains field          PART_ID_USED_IN
|   |   Datatype                signed longword
|   Contains field          PART_QUANT
|   |   Datatype                signed word
|   Constraint PARTS_PMK  primary key PART_ID NOT DEFERRABLE
|   Constraint PARTS_UNQ  unique PART_NO NOT DEFERRABLE
|   Constraint PART_CST (ANY (P IN PARTS WITH
|          (PART_ID IN PARTS EQ PART_ID_USED_IN IN P))) NOT DEFERRABLE
|   Constraint PART_FRK  foreign key PART_ID references OTHER_PARTS
|           OTHER_PART_ID NOT DEFERRABLE
CDO> EXIT
$ !
$ ! Invoke SQL:
$ SQL
SQL> -- Attach to the AUTO database.
SQL> --
SQL> ATTACH 'ALIAS AUTO PATHNAME AUTO';
```

(continued on next page)

Example 10–5 (Cont.) Creating Record-Level Constraints

```
SQL> --
SQL> -- Create a table called PARTS using the PARTS record (table)
SQL> -- just created in the repository:
SQL> --
SQL> CREATE TABLE FROM SYS$COMMON:[REPOSITORY]TABLE_TEST.PARTS
cont>        ALIAS AUTO;
SQL> --
SQL> -- Display information about the PARTS table.
SQL> --
SQL> SHOW TABLE AUTO.PARTS;
Information for table AUTO.PARTS

CDD Pathname: SYS$COMMON:[REPOSITORY]TABLE_TEST.PARTS;1

Columns for table AUTO.PARTS:
Column Name    Data Type  Domain
-----------    ---------  ------
PART_NO                         SMALLINT  AUTO.PART_NO
PART_ID                         INTEGER   AUTO.PART_ID
PART_ID_USED_IN                 INTEGER   AUTO.PART_ID_USED_IN
PART_QUANT                      SMALLINT  AUTO.PART_QUANT

Table constraints for AUTO.PARTS:
AUTO.PARTS_PMK
 Primary Key constraint
 Table constraint for AUTO.PARTS
 Evaluated on each VERB
 Source: primary key PART_ID

AUTO.PARTS_UNQ
 Unique constraint
 Table constraint for AUTO.PARTS
 Evaluated on each VERB
 Source: unique PART_NO

AUTO.PART_CST
 Check constraint
 Table constraint for AUTO.PARTS
 Evaluated on each VERB
 Source: (ANY (P IN PARTS WITH (PART_ID IN PARTS EQ PART_ID_USED_IN IN P)))

AUTO.PART_FRK
 Foreign Key constraint
 Table constraint for AUTO.PARTS
 Evaluated on each VERB
 Source: foreign key PART_ID references OTHER_PARTS OTHER_PART_ID
```

(continued on next page)

Example 10–5 (Cont.) Creating Record-Level Constraints

```
Constraints referencing table AUTO.PARTS:
No constraints found
   .
   .
   .
SQL> --
SQL> COMMIT;
SQL> DISCONNECT DEFAULT;
SQL> EXIT;
```

10.5 Modifying Repository Definitions Using CDO

You can modify metadata in the repository by using one of the CDO DEFINE commands to make a new version of a definition or with one of the CDO CHANGE commands to replace a definition.

10.5.1 Using the CDO DEFINE Commands

The CDO DEFINE RECORD and DEFINE FIELD commands allow you to create new versions of existing field and record definitions. Several versions of the same definition can be stored in the repository. When you specify a definition without a version number, the repository assigns the highest version number to the definition.

In Example 10–6, the CDO DEFINE FIELD command creates a new version of the SECOND_ORDER field.

Example 10–6 Using the CDO DEFINE FIELD Command

```
CDO> ! Note there is only one version of the field SECOND_ORDER.
CDO> !
CDO> DIRECTORY
 Directory SYS$COMMON:[REPOSITORY]TEST2
CUSTOMER_ORDERS(1)                          RECORD
FIFTH_DOM(1)                                FIELD
FIRST_ORDER(1)                              FIELD
FOURTH_ORDER(1)                             FIELD
SECOND_ORDER(1)                             FIELD
TEST2(1)                                    CDD$DATABASE
THIRD_ORDER(1)                              FIELD
```

(continued on next page)

Example 10–6 (Cont.) Using the CDO DEFINE FIELD Command

```
CDO> !
CDO> ! Display the field SECOND_ORDER.
CDO> !
CDO> SHOW FIELD SECOND_ORDER
Definition of field SECOND_ORDER
|   Datatype                 text size is 4 characters
CDO> !
CDO> ! Use the DEFINE FIELD command to create a new version of
CDO> ! the SECOND_ORDER field that has a word data type.
CDO> !
CDO> DEFINE FIELD SECOND_ORDER DATATYPE IS WORD.
CDO> !
CDO> ! Display the new version of the field SECOND_ORDER. Note
CDO> ! CDO displays the highest version of the field by default.
CDO> !
CDO> SHOW FIELD SECOND_ORDER
Definition of field SECOND_ORDER
|   Datatype                 signed word
CDO> !
CDO> ! Note the two versions of the field SECOND_ORDER.
CDO> !
CDO> DIRECTORY
 Directory SYS$COMMON:[REPOSITORY]TEST2
CUSTOMER_ORDERS(1)                          RECORD
FIFTH_DOM(1)                                FIELD
FIRST_ORDER(1)                              FIELD
FOURTH_ORDER(1)                             FIELD
SECOND_ORDER(2)                             FIELD
SECOND_ORDER(1)                             FIELD
TEST2(1)                                    CDD$DATABASE
THIRD_ORDER(1)                              FIELD
```

You can issue a CDO SHOW USED_BY command to see if the database definition uses an older version, as opposed to a newer version of a definition. When you store several versions of the same definition, you can specify and use any of these versions.

Example 10–7 shows how to use the CDO SHOW USED_BY command to check what version of the field SECOND_ORDER is used by the database TEST2. Note TEST2 still uses the older version (version 1) of the SECOND_ORDER field.

Example 10–7 Determining What Version of a Definition Is Used by a Database

```
CDO> SHOW USED_BY TEST2/FULL

SHOW USED_BY TEST2(1)/FULL
Members of SYS$COMMON:[REPOSITORY]TEST2.TEST2(1)
|   TEST2                           (Type : CDD$RDB_DATABASE)
|   |   via CDD$DATABASE_SCHEMA
   .
   .
   .
|   |   SYS$COMMON:[REPOSITORY]TEST2.FIRST_ORDER(1)  (Type : FIELD)
|   |   |   via CDD$RDB_DATA_ELEMENT
|   |   SYS$COMMON:[REPOSITORY]TEST2.SECOND_ORDER(1)  (Type : FIELD)
|   |   |   via CDD$RDB_DATA_ELEMENT
|   |   SYS$COMMON:[REPOSITORY]TEST2.THIRD_ORDER(1)  (Type : FIELD)
|   |   |   via CDD$RDB_DATA_ELEMENT
|   |   SYS$COMMON:[REPOSITORY]TEST2.FOURTH_ORDER(1)  (Type : FIELD)
|   |   |   via CDD$RDB_DATA_ELEMENT
|   |   SYS$COMMON:[REPOSITORY]TEST2.FIFTH_DOM(1)  (Type : FIELD)
|   |   |   via CDD$RDB_DATA_ELEMENT
   .
   .
   .
```

When you use either the DEFINE FIELD or DEFINE RECORD command to change an existing definition, other definitions that use the existing definition are *not* automatically updated to reflect the change. You see notices stating that changes have been made to the definition in the repository when you invoke the database that uses the existing definition. Also, if you issue the CDO SHOW NOTICES command on the database definition, you see a message that a new version of the definition exists, as Example 10–8 shows.

10.5.2 Using the CDO CHANGE Commands

Use the CDO CHANGE FIELD or CHANGE RECORD command to modify an existing definition, if you want all other definitions using that definition to automatically reflect the change.

You receive a notice that a change was made to the definition in the repository when you invoke the database that uses the modified definition. Also, if you issue the CDO SHOW NOTICES command on the database definition, you see a message that indicates that the database is possibly invalid, as Example 10–8 shows.

Note that the CHANGE command changes the original definition without creating a new version of the definition.

Example 10–8 shows how to use the CDO CHANGE FIELD command to change an existing field definition in the repository.

Example 10–8 Using the CDO CHANGE FIELD Command

```
CDO> ! Display field to be changed.
CDO> !
CDO> SHOW FIELD FIRST_ORDER
Definition of field FIRST_ORDER
|   Datatype                 text size is 4 characters
CDO> !
CDO> ! Use the CHANGE FIELD command to change the data type of FIRST_ORDER.
CDO> ! Note the database TEST2 now has a notice attached to it.
CDO> !
CDO> CHANGE FIELD FIRST_ORDER DATATYPE IS WORD.
%CDO-I-DBMBR, database SYS$COMMON:[REPOSITORY]TEST2.TEST2(1) may need to
be INTEGRATED
CDO> !
CDO> ! Display the new definition for the field FIRST_ORDER.
CDO> !
CDO> SHOW FIELD FIRST_ORDER
Definition of field FIRST_ORDER
|   Datatype                 signed word
CDO> !
CDO> ! Note there is only one version of the FIRST_ORDER field.
CDO> !
CDO> DIRECTORY
 Directory SYS$COMMON:[REPOSITORY]TEST2
CUSTOMER_ORDERS(2)                         RECORD
FIFTH_DOM(1)                               FIELD
FIRST_ORDER(1)                             FIELD
FOURTH_ORDER(1)                            FIELD
SECOND_ORDER(2)                            FIELD
SECOND_ORDER(1)                            FIELD
TEST2(1)                                   CDD$DATABASE
THIRD_ORDER(1)                             FIELD
CDO> !
CDO> ! Display the notices on the database TEST2 caused by the CHANGE FIELD
CDO> ! and DEFINE FIELD commands.
CDO> !
CDO> SHOW NOTICES TEST2.TEST2
SYS$COMMON:[REPOSITORY]TEST2.TEST2(1) uses an entity which has new versions,
triggered by CDD$DATA_ELEMENT SYS$COMMON:[REPOSITORY]TEST2.SECOND_ORDER(1)
SYS$COMMON:[REPOSITORY]TEST2.TEST2(1) is possibly invalid,
triggered by CDD$DATA_ELEMENT SYS$COMMON:[REPOSITORY]TEST2.FIRST_ORDER(1)
CDO> EXIT
$ !
```

(continued on next page)

Example 10–8 (Cont.) Using the CDO CHANGE FIELD Command

```
$ ! Invoke SQL.
$ !
$ SQL
SQL> -- When you attach to the database, SQL displays any notices.
SQL> --
SQL> ATTACH 'ALIAS TEST2 PATHNAME SYS$COMMON:[REPOSITORY]TEST2.TEST2';
%SQL-I-DIC_DB_CHG1, A dictionary definition used by database
SYS$COMMON:[REPOSITORY]TEST2.TEST2;1 has changed
-SQL-I-DIC_DB_CHG2, Use the INTEGRATE statement to resolve any differences
between the repository and the database
%CDD-I-MESS, entity has messages
SQL> DISCONNECT DEFAULT;
SQL> EXIT
```

Because Example 10–6 and Example 10–8 modified the definitions in the repository, the database files and the repository no longer match. Use the SQL INTEGRATE statement to alter the database definitions so they match those in the repository.

```
$ SQL
SQL> INTEGRATE DATABASE PATHNAME SYS$COMMON:[REPOSITORY]TEST2.TEST2
cont>    ALTER FILES;
SQL> COMMIT;
```

Section 10.7 describes the INTEGRATE statement in more detail.

10.6 Modifying Repository Definitions and Database Files

There might be times when the definitions in the database files no longer match the definitions in the repository. Section 10.7 explains whether or not the repository is updated when you use SQL to create data definitions.

When the data definitions no longer match, you can use one source to update the other:

- You can update the repository using the definitions in the database files. See Section 10.7 for more information.
- You can update the database files using the definitions in the repository. See Section 10.10 for more information.

With either of these options, you can update the entire database or only specific domains or tables. Section 10.9 and Section 10.11 describe how to integrate domains and tables.

10.7 Understanding How the Repository Is Updated

There are four possible combinations of the CREATE DATABASE statement and the ATTACH statement that affect how the repository will be updated using the database files as the source. Table 10–2 provides an overview of these combinations and the action you must take to update the repository.

Table 10–2 How CREATE DATABASE and ATTACH Statements Affect Repository Updates

	CREATE DATABASE		ATTACH					
Combination Number	DICTIONARY IS REQUIRED	PATHNAME CLAUSE	FILENAME CLAUSE	PATHNAME CLAUSE	INTEGRATE	Repository Updated	Database Updated	Error Occurred
1	Y	Y	N	Y	N	Y	Y	N
2	Y	Y	Y	N	N	N	N	Y
3	N	Y	Y	N	Y[1]	N	Y	N
4	N	N	Y	N	Y[2]	N	Y	N

[1]INTEGRATE . . . ALTER DICTIONARY
[2]INTEGRATE . . . CREATE PATHNAME

How to read Table 10–2:

- Each combination number specifies one of the four possible combinations of the CREATE DATABASE and ATTACH statements.
- The letter Y (Yes) under the CREATE DATABASE or ATTACH statement indicates that a specific clause or key words were specified prior to a change.
- The letter Y under INTEGRATE indicates that the INTEGRATE statement must be issued to make the repository and the database file consistent if a change is made.
- The letter Y under Repository Updated or Database Updated indicates that any changes made during an attach are captured as updates.
- The letter Y under Error Occurred indicates that no changes made during an attach updated the repository or the database.

The following sections explain each of these combinations in more detail.

10.7.1 Automatically Updating the Repository and the Database File Using SQL

You create a database using the DICTIONARY IS REQUIRED and the PATHNAME clauses of the CREATE DATABASE statement. You subsequently attach to the database using the PATHNAME clause of the ATTACH statement.

SQL *automatically* updates the repository and the database file with any changes you make during that attachment to the database. You do not need to use the INTEGRATE statement.

Combination 1 in Table 10–2 summarizes this condition.

10.7.2 Receiving an Error on Updating the Repository

You create a database using the DICTIONARY IS REQUIRED and the PATHNAME clauses of the CREATE DATABASE statement. You subsequently attach to the database using the FILENAME option of the ATTACH statement.

SQL produces an error message in response to subsequent data definition statements because the database definition specifies the DICTIONARY IS REQUIRED clause.

If your database specifies the DICTIONARY IS REQUIRED clause, you should attach to the database using the PATHNAME clause because FILENAME specifies that only the database file will be updated. When you specify the PATHNAME clause, SQL updates both the repository and the database file.

Combination 2 in Table 10–2 summarizes this condition.

10.7.3 Storing Initial Definitions in the Repository but Updating Only the Database File

You create a database using the DICTIONARY IS NOT REQUIRED (the default) and PATHNAME clauses of the CREATE DATABASE statement. You subsequently attach to the database using the FILENAME clause of the ATTACH statement.

SQL stores the initial database definitions in the repository because you specified the PATHNAME clause in the CREATE DATABASE statement. However, because you attached to the database using the FILENAME clause and had previously specified the DICTIONARY IS NOT REQUIRED clause, any changes you make to the database definitions during that attachment are entered only in the database file, not in the repository.

Combination 3 in Table 10–2 summarizes this condition.

To update the repository definitions that no longer match those in the database file, use the INTEGRATE statement with the ALTER DICTIONARY clause. This statement alters the repository definitions so they are the same as those in the database file. Note that altering definitions in the repository might affect other repository elements that refer to these definitions.

Example 10–9 shows how to update the repository using the database files as the source by issuing the INTEGRATE statement with the ALTER DICTIONARY clause. The example starts with the definitions in the repository matching the definitions in the database file. There is a table in the database and a record in the repository, both called CUSTOMER_ORDERS. The CUSTOMER_ORDERS table has four columns based on four domains of the same name: FIRST_ORDER, SECOND_ORDER, THIRD_ORDER, and FOURTH_ORDER.

This example adds to the database file a domain called FIFTH_DOM, and bases a local column called FIFTH_ORDER on it. At this point, the database file and the repository definitions no longer match. The INTEGRATE . . . ALTER DICTIONARY statement resolves this situation by modifying the repository using the database file definitions as the source.

Example 10–9 Modifying Repository Definitions Using the INTEGRATE Statement with the ALTER DICTIONARY Clause

```
SQL> -- Create the database using the PATHNAME clause.
SQL> --
SQL> CREATE DATABASE FILENAME TEST1
cont>              PATHNAME SYS$COMMON:[REPOSITORY]TEST1;
SQL> --
SQL> -- Create domains for the TEST1 database.
SQL> --
SQL> CREATE DOMAIN FIRST_ORDER CHAR(4);
SQL> CREATE DOMAIN SECOND_ORDER CHAR(4);
SQL> CREATE DOMAIN THIRD_ORDER CHAR(4);
SQL> CREATE DOMAIN FOURTH_ORDER CHAR(4);
SQL> CREATE TABLE CUSTOMER_ORDERS
cont>     (FIRST_ORDER FIRST_ORDER,
cont>      SECOND_ORDER SECOND_ORDER,
cont>      THIRD_ORDER THIRD_ORDER,
cont>      FOURTH_ORDER FOURTH_ORDER);
SQL> COMMIT;
SQL> DISCONNECT DEFAULT;
```

(continued on next page)

Example 10–9 (Cont.) Modifying Repository Definitions Using the INTEGRATE Statement with the ALTER DICTIONARY Clause

```
SQL> --
SQL> -- Attach to the database with the FILENAME clause so the
SQL> -- repository is not updated.
SQL> --
SQL> ATTACH  'ALIAS TEST1 FILENAME TEST1';
SQL> --
SQL> -- Create a new domain called FIFTH_DOM.
SQL> CREATE DOMAIN TEST1.FIFTH_DOM CHAR(4);
SQL> --
SQL> -- Add a new column, FIFTH_ORDER, to the CUSTOMER_ORDERS table
SQL> -- and base it on the domain FIFTH_DOM.
SQL> ALTER TABLE TEST1.CUSTOMER_ORDERS
cont>      ADD FIFTH_ORDER TEST1.FIFTH_DOM;
SQL> --
SQL> -- Check the CUSTOMER_ORDERS table to verify that the column FIFTH_ORDER
SQL> -- was created.
SQL> --
SQL> SHOW TABLE (COLUMNS) TEST1.CUSTOMER_ORDERS;

Information on table TEST1.CUSTOMER_ORDERS

Column Name                     Data Type        Domain
-----------                     ---------        ------
FIRST_ORDER                     CHAR(4)          TEST1.FIRST_ORDER
SECOND_ORDER                    CHAR(4)          TEST1.SECOND_ORDER
THIRD_ORDER                     CHAR(4)          TEST1.THIRD_ORDER
FOURTH_ORDER                    CHAR(4)          TEST1.FOURTH_ORDER
FIFTH_ORDER                     CHAR(4)          TEST1.FIFTH_DOM

SQL> COMMIT;
SQL> EXIT
$ !
$ ! Invoke CDO:
$ !
$ REPOSITORY
Welcome to CDO V2.3
The CDD/Repository V5.3 User Interface
Type HELP for help
CDO> !
CDO> ! Note that only the database definition for TEST1 appears in the
CDO> ! repository directory.
CDO> !
  DIRECTORY
Directory SYS$COMMON:[REPOSITORY]
TEST1(1)                                  CDD$DATABASE
CDO> !
```

(continued on next page)

Example 10–9 (Cont.) Modifying Repository Definitions Using the INTEGRATE Statement with the ALTER DICTIONARY Clause

```
CDO> ! Check the record CUSTOMER_ORDERS.  The field FIFTH_ORDER is not part of
CDO> ! the record CUSTOMER_ORDERS.  This means that the definitions in the
CDO> ! database file do not match the definitions in the repository.
CDO> !
CDO> !
CDO> SHOW RECORD CUSTOMER_ORDERS FROM DATABASE TEST1
Definition of the record CUSTOMER_ORDERS
|   Contains field              FIRST_ORDER
|   Contains field              SECOND_ORDER
|   Contains field              THIRD_ORDER
|   Contains field              FOURTH_ORDER
CDO> EXIT
$ !
$ ! Invoke SQL again:
$ !
$ SQL
SQL> -- To make the definitions in the repository match those in the database
SQL> -- file, use the INTEGRATE statement with the ALTER DICTIONARY clause.
SQL> -- Note that the INTEGRATE statement implicitly attaches to the
SQL> -- database.
SQL> --
SQL> INTEGRATE DATABASE PATHNAME TEST1 ALTER DICTIONARY;
SQL> COMMIT;
SQL> EXIT
$ !
$ ! Invoke CDO again:
$ !
$ REPOSITORY
Welcome to CDO V2.3
The CDD/Repository V5.3 User Interface
Type HELP for help
CDO> !
CDO> ! Use the SHOW RECORD command to verify that the field FIFTH_ORDER is now
CDO> ! part of the record CUSTOMER_ORDERS.  Now, the definitions in both the
CDO> ! repository and the database file are the same.
CDO> !
CDO> SHOW RECORD CUSTOMER_ORDERS FROM DATABASE TEST1
Definition of record CUSTOMER_ORDERS
|   Contains field           FIRST_ORDER
|   Contains field           SECOND_ORDER
|   Contains field           THIRD_ORDER
|   Contains field           FOURTH_ORDER
|   Contains field           FIFTH_ORDER
```

(continued on next page)

Example 10–9 (Cont.) Modifying Repository Definitions Using the INTEGRATE Statement with the ALTER DICTIONARY Clause

```
CDO> !
CDO> ! Use the ENTER command to make the record (table) CUSTOMER_ORDERS and
CDO> ! its fields (domains) appear in the repository.  The ENTER command
CDO> ! assigns a repository directory name to an element.
CDO> !
CDO> ENTER FIELD FIRST_ORDER FROM DATABASE TEST1
CDO> !
CDO> ! Verify that a repository path name was assigned to the field
CDO> ! FIRST_ORDER.
CDO> !
CDO> DIRECTORY
 Directory SYS$COMMON:[REPOSITORY]
FIRST_ORDER(1)                           FIELD
TEST1(1)                                 CDD$DATABASE
CDO> ENTER FIELD SECOND_ORDER FROM DATABASE TEST1
   .
   .
   .
CDO> ENTER FIELD FIFTH_DOM FROM DATABASE TEST1
CDO> !
CDO> ! Now all the domains and tables in TEST1 have been assigned a
CDO> ! repository directory name.
CDO> DIRECTORY
Directory SYS$COMMON:[REPOSITORY]
CUSTOMER_ORDERS(1)                       RECORD
FIFTH_DOM(1)                             FIELD
FIRST_ORDER(1)                           FIELD
FOURTH_ORDER(1)                          FIELD
SECOND_ORDER(1)                          FIELD
TEST1(1)                                 CDD$DATABASE
THIRD_ORDER(1)                           FIELD
```

10.7.4 Not Storing Initial Definitions in the Repository and Updating Only the Database File

You create a database using the DICTIONARY IS NOT REQUIRED clause (the default), and you do not specify the PATHNAME clause of the CREATE DATABASE statement. You subsequently attach to the database using the FILENAME clause of the ATTACH statement.

Because you did not specify the PATHNAME clause, SQL did not store the original database definitions in the repository. When you specified the FILENAME clause, SQL automatically entered the changes made during that attach into the database file only.

Combination 4 in Table 10–2 summarizes this condition.

To update the repository, you must first store the existing database file definitions in the repository. For information on how to do this, see Section 10.8.

10.8 Updating Repository Definitions Using SQL

If you create a database using the DICTIONARY IS NOT REQUIRED clause and do not specify the PATHNAME clause, SQL does not store the original database definitions in the repository.

To store existing database file definitions in the repository for the first time, use the INTEGRATE statement with the CREATE PATHNAME clause. This statement builds repository definitions using the database file as the source.

Example 10–10 shows how to store existing database system file definitions in the repository for the first time. This example first creates a database only in a database file, not in the repository. Next, the INTEGRATE statement with the CREATE PATHNAME clause updates the repository with the data definitions from the database system file.

Example 10–10 Storing Existing Database File Definitions in the Repository

```
SQL> -- Create a database without requiring the repository (the default)
SQL> -- or specifying a path name.
SQL> --
SQL> CREATE DATABASE ALIAS DOGS;
SQL> --
SQL> -- Now create a table for the breed of dog, poodles.  The
SQL> -- columns in the table are types of poodles.
SQL> --
SQL> CREATE TABLE DOGS.POODLES
cont>       ( STANDARD  CHAR(10),
cont>         MINIATURE CHAR(10),
cont>         TOY       CHAR(10) );
SQL> --
SQL> COMMIT;
SQL> EXIT
```

(continued on next page)

Example 10–10 (Cont.) Storing Existing Database File Definitions in the Repository

```
$ !
$ ! Invoke CDO:
$ !
$ REPOSITORY
Welcome to CDO V2.3
The CDD/Repository V5.3 User Interface
Type HELP for help
CDO> !
CDO> ! Use the DIRECTORY command to see that the database definition DOGS is
CDO> ! not in the repository.
CDO> !
CDO> DIRECTORY
Directory SYS$COMMON:[REPOSITORY]
%CDO-E-NOTFOUND, entity  not found in dictionary
CDO> !
CDO> EXIT
$ !
$ ! Invoke SQL again.
$ !
$ SQL
SQL> -- Use the INTEGRATE statement with the CREATE PATHNAME clause to
SQL> -- update the repository using the DOGS database file.
SQL> --
SQL> INTEGRATE DATABASE FILENAME SQL_USER:[PRODUCTION.ANIMALS]DOGS
cont>    CREATE PATHNAME SYS$COMMON:[REPOSITORY]DOGS;
SQL> COMMIT;
SQL> EXIT
$ !
$ ! Invoke CDO again:
$ !
$ REPOSITORY
Welcome to CDO V2.3
The CDD/Repository V5.3 User Interface
Type HELP for help
CDO> !
CDO> ! Use the DIRECTORY command to check if the database definition DOGS
CDO> ! has been integrated into the repository.
CDO> !
CDO> DIRECTORY
 Directory SYS$COMMON:[REPOSITORY]
DOGS(1)                                  CDD$DATABASE
CDO> !
CDO> ! You can also use the SHOW USED_BY command to see if the
CDO> ! record (table) POODLES and the fields (columns) STANDARD,
CDO> ! MINIATURE, and TOY are part of the database definition DOGS.
CDO> !
```

(continued on next page)

Example 10–10 (Cont.) Storing Existing Database File Definitions in the Repository

```
CDO> SHOW USED_BY/FULL DOGS
Members of SYS$COMMON:[REPOSITORY]DOGS(1)
|   DOGS                           (Type : CDD$RDB_DATABASE)
|   |   via CDD$DATABASE_SCHEMA
   .
   .
   .
|   SYS$COMMON:[REPOSITORY]CDD$RDB_SYSTEM_METADATA.RDB$CDD_NAME;1(Type : FIELD)
|   |   |   |   via CDD$DATA_AGGREGATE_CONTAINS
|   |   POODLES                        (Type : RECORD)
|   |   |   via CDD$RDB_DATA_AGGREGATE
|   |   |   STANDARD                       (Type : FIELD)
|   |   |   |   via CDD$DATA_AGGREGATE_CONTAINS
|   |   |   |   SQL$10CHR                      (Type : FIELD)
|   |   |   |   |   via CDD$DATA_ELEMENT_BASED_ON
|   |   |   MINIATURE                      (Type : FIELD)
|   |   |   |   via CDD$DATA_AGGREGATE_CONTAINS
|   |   |   |   SQL$10CHR                      (Type : FIELD)
|   |   |   |   |   via CDD$DATA_ELEMENT_BASED_ON
|   |   |   TOY                            (Type : FIELD)
|   |   |   |   via CDD$DATA_AGGREGATE_CONTAINS
|   |   |   |   SQL$10CHR                      (Type : FIELD)
|   |   |   |   |   via CDD$DATA_ELEMENT_BASED_ON
   .
   .
   .
CDO> EXIT
```

For more information on using the INTEGRATE statement to store existing database file definitions in the repository, see the *Oracle Rdb7 SQL Reference Manual*.

10.9 Integrating Domains and Tables Using Database Files

Instead of integrating all the database definitions, you can specify particular domains or tables to integrate using the following arguments to the INTEGRATE statement:

- DOMAIN
- TABLE

The INTEGRATE statement with the DOMAIN or TABLE argument makes the domain or table definition in a database and a repository match. You can specify that the definitions in the database be used to update the repository by using the ALTER DICTIONARY clause.

Example 10–11 illustrates how to integrate a domain, updating the repository from the database definition.

Example 10–11 Modifying a Domain Definition in the Repository Using the Definition from the Database

```
SQL> -- Attach to the database with the FILENAME clause so the
SQL> -- repository is not updated.
SQL> --
SQL> ATTACH 'FILENAME PERSONNEL';
SQL> --
SQL> -- Modify the definition of the POSTAL_CODE_DOM domain:
SQL> --
SQL> ALTER DOMAIN POSTAL_CODE_DOM IS CHAR(9);
SQL> COMMIT;
SQL> DISCONNECT ALL;
SQL> --
SQL> -- Attach to the database with the PATHNAME clause.
SQL> --
SQL> ATTACH 'PATHNAME CDD$DEFAULT.PERSONNEL';
SQL> --
SQL> -- Use the ALTER DICTIONARY clause to update the repository
SQL> -- using the database definition as the source.
SQL> --
SQL> INTEGRATE DOMAIN POSTAL_CODE_DOM ALTER DICTIONARY;
SQL> COMMIT;
```

10.10 Updating the Database File Using Repository Definitions

There are times when you need to update the database file using the repository definitions as the source. This integration goes in the opposite direction of the integrations described in the previous section.

If you create a table or domain definition with the CREATE TABLE FROM or the CREATE DOMAIN FROM statement, you can use the INTEGRATE . . . ALTER FILES statement to update the database system file using the repository definitions as the source. The INTEGRATE . . . ALTER FILES statement has no effect on definitions not created with the FROM clause.

Example 10–12 shows how to use the INTEGRATE statement with the ALTER FILES clause. In this example, you define fields (domains) in the repository. Then, using SQL, you create a table based on the repository definitions. Subsequently, you change the repository definitions so the definitions in the database file and the repository no longer match. The INTEGRATE statement resolves this situation by altering the database definitions using the repository definitions as the source.

Example 10–12 Updating the Database File Using the Repository Definitions

```
$ ! Invoke CDO to create new field and record definitions:
$ !
$ REPOSITORY
Welcome to CDO V2.3
The CDD/Repository V5.3 User Interface
Type HELP for help
CDO> !
CDO> ! Create two field (domain) definitions in the repository.
CDO> !
CDO> DEFINE FIELD PART_NUMBER DATATYPE IS WORD.
CDO> DEFINE FIELD PRICE DATATYPE IS WORD.
CDO> !
CDO> ! Define a record called INVENTORY using the two fields
CDO> ! previously defined:
CDO> !
CDO> DEFINE RECORD INVENTORY.
CDO>    PART_NUMBER.
CDO>    PRICE.
CDO> END RECORD INVENTORY.
CDO> !
CDO> EXIT
$ !
$ ! Invoke SQL:
$ !
$ SQL
SQL> -- Create the database ORDERS.
SQL> --
SQL> CREATE DATABASE ALIAS ORDERS PATHNAME ORDERS;
SQL> --
SQL> -- Create a table using the INVENTORY record (table)
SQL> -- just created in the repository.
SQL> --
SQL> CREATE TABLE FROM SYS$COMMON:[REPOSITORY]CATALOG.INVENTORY
cont>       ALIAS ORDERS;
SQL> --
SQL> -- Display information about the INVENTORY table.
SQL> --
SQL> SHOW TABLE (COLUMNS) ORDERS.INVENTORY
Information for table ORDERS.INVENTORY
```

(continued on next page)

Example 10–12 (Cont.) Updating the Database File Using the Repository Definitions

```
CDD Pathname:   SYS$COMMON:[REPOSITORY]CATALOG.INVENTORY;1

Columns for table ORDERS.INVENTORY:
Column Name                    Data Type        Domain
-----------                    ---------        ------
PART_NUMBER                    SMALLINT         ORDERS.PART_NUMBER
PRICE                          SMALLINT         ORDERS.PRICE

SQL> COMMIT;
SQL> EXIT
$ !
$ ! Invoke CDO again:
$ !
$ REPOSITORY
Welcome to CDO V2.3
The CDD/Repository V5.3 User Interface
Type HELP for help
CDO> !
CDO> ! Verify that the fields PART_NUMBER and PRICE are in the
CDO> ! record INVENTORY:
CDO> !
CDO> SHOW RECORD INVENTORY
Definition of record INVENTORY
|   Contains field          PART_NUMBER
|   Contains field          PRICE
CDO> !
CDO> ! Define the fields VENDOR_NAME and QUANTITY. Add them to the
CDO> ! record INVENTORY using the CDO CHANGE RECORD command. Now, the
CDO> ! definitions used by the database no longer match the definitions
CDO> ! in the repository, as the CDO message indicates.
CDO> !
CDO> DEFINE FIELD VENDOR_NAME DATATYPE IS TEXT 20.
CDO> DEFINE FIELD QUANTITY DATATYPE IS WORD.
CDO> !
CDO> CHANGE RECORD INVENTORY.
CDO>    DEFINE VENDOR_NAME.
CDO>    END.
CDO>    DEFINE QUANTITY.
CDO>    END.
CDO> END INVENTORY RECORD.
%CDO-I-DBMBR, database SQL_USER:[PRODUCTION]CATALOG.ORDERS(1) may need
to be INTEGRATED
```

(continued on next page)

Example 10–12 (Cont.) Updating the Database File Using the Repository Definitions

```
CDO> !
CDO> ! Use the SHOW RECORD command to see if the fields VENDOR_NAME
CDO> ! and QUANTITY are part of the INVENTORY record.
CDO> !
CDO> SHOW RECORD INVENTORY
Definition of record INVENTORY
|   Contains field          PART_NUMBER
|   Contains field          PRICE
|   Contains field          VENDOR_NAME
|   Contains field          QUANTITY
CDO> !
CDO> EXIT
$ !
$ ! Invoke SQL again:
$ !
$ SQL
SQL> -- Use the INTEGRATE  . . .  ALTER FILES statement to update the
SQL> -- definitions in the database file, using the repository definitions
SQL> -- as the source. Note the INTEGRATE statement implicitly attaches to
SQL> -- the database.
SQL> --
SQL> INTEGRATE DATABASE PATHNAME SYS$COMMON:[REPOSITORY]CATALOG.ORDERS
cont>   ALTER FILES;
SQL> --
SQL> -- Use the SHOW TABLE statement to see that SQL added the
SQL> -- VENDOR_NAME and QUANTITY domains to the database file.
SQL> --
SQL> SHOW TABLE (COLUMNS) INVENTORY
Information for table INVENTORY

CDD Pathname:   SYS$COMMON:[REPOSITORY]CATALOG.INVENTORY;1

Columns for table INVENTORY:
Column Name                     Data Type        Domain
-----------                     ---------        ------
PART_NUMBER                     SMALLINT         PART_NUMBER
PRICE                           SMALLINT         PRICE
VENDOR_NAME                     CHAR(20)         VENDOR_NAME
QUANTITY                        SMALLINT         QUANTITY

SQL> COMMIT;
SQL> EXIT
```

For more information on using the INTEGRATE statement to update database files, see the *Oracle Rdb7 SQL Reference Manual*.

10.11 Integrating Domains and Tables Using Repository Definitions

Instead of integrating all the database definitions, you can specify particular domains or tables to integrate using the following arguments to the INTEGRATE statement:

- DOMAIN
- TABLE

The INTEGRATE statement with the DOMAIN or TABLE argument makes the domain or table definition in a database and a repository match. You can specify that the definitions in the repository be used to update the database files by using the ALTER FILES clause.

Example 10–13 illustrates how to integrate a table, updating the repository from the database definition.

Example 10–13 Modifying a Table Definition in the Database Using the Definition from the Repository

```
CDO> !  Define a new field.
CDO> !
CDO> DEFINE FIELD ACCREDITATION_STATUS DATATYPE IS TEXT 3.
CDO> !
CDO> !  Add the field to an existing record.
CDO> !
CDO> CHANGE RECORD COLLEGES.
cont> DEFINE ACCREDITATION_STATUS.
cont> END.
cont> END COLLEGES RECORD.
%CDO-I-DBMBR, database SQL_USER1:[DAY.CDDTEST]PERSONNEL(1) may need to be
INTEGRATED
CDO> EXIT.
$ !
$ !  Invoke SQL:
$ !
$ SQL
SQL> -- Attach to the database with the PATHNAME clause.
SQL> --
SQL> ATTACH 'PATHNAME CDD$DEFAULT.PERSONNEL';
%SQL-I-DIC_DB_CHG1, A dictionary definition used by database
SQL_USER1:[DAY.CDD.TEST]PERSONNEL;1 has changed
-SQL-I-DIC_DB_CHG2, Use the INTEGRATE statement to resolve any differences
between the dictionary and the database
%CDD-I-MESS, entity has messages
```

(continued on next page)

Example 10–13 (Cont.) Modifying a Table Definition in the Database Using the Definition from the Repository

```
SQL> --
SQL> -- Use the ALTER FILES clause to update the database
SQL> -- using the repository definition as the source.
SQL> --
SQL> INTEGRATE TABLE COLLEGES ALTER FILES;
SQL> COMMIT;
```

10.12 Using SQL to Delete Definitions

This section describes how to use SQL to:

- Delete all links (associations) between a database and the repository. The repository definition can be used by other elements in the repository.
- Delete a link between a particular database definition and the repository.
- Delete the CDD$DATABASE element from the repository with its associated definitions.

10.12.1 Removing All Links Between a Database and the Repository

To remove the link between the repository and database but still maintain the definitions in both places, you can use the DICTIONARY IS NOT USED clause of the ALTER DATABASE statement. After you remove the links, you can integrate the database to a new repository. You might use this command to move the database to a new system and use the repository on that system.

Example 10–14 shows how to remove the link between the dept1 database and the repository.

Example 10–14 Removing Links to the Repository

```
SQL> -- Attach to the database and show the pathname.
SQL> ATTACH 'PATHNAME dept1';
SQL> SHOW DATABASE RDB$DBHANDLE
Default alias:
    data dictionary pathname is SYS$COMMON:[REPOSITORY]PERS.DEPT1;1
   .
   .
   .
SQL> DISCONNECT DEFAULT;
```

(continued on next page)

Example 10–14 (Cont.) Removing Links to the Repository

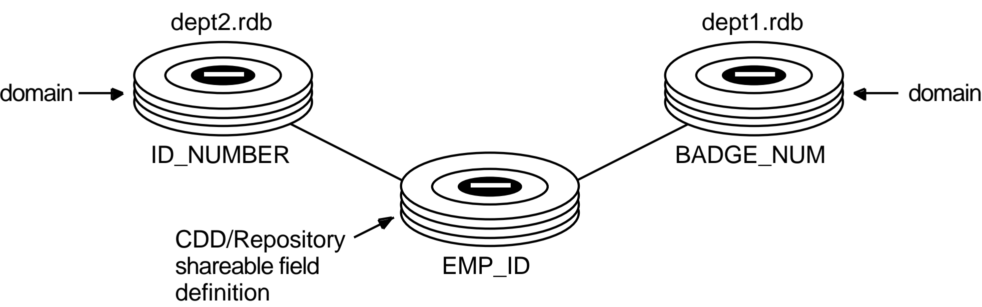
```
SQL> --
SQL> -- To remove the links, alter the database using the DICTIONARY IS NOT
SQL> -- USED clause.
SQL> --
SQL> ALTER DATABASE FILENAME DEPT1
cont>      DICTIONARY IS NOT USED;
```

To use the DICTIONARY IS NOT USED clause, you must use the FILENAME clause, not the PATHNAME clause, in the ALTER DATABASE statement.

10.12.2 Deleting Links with Database Definitions

You cannot use SQL to delete shared fields (domains) that reside in a repository. For example, Figure 10–2 shows two domains in two separate Oracle Rdb databases that are based on one shareable field definition (EMP_ ID) in the repository. If you try to delete EMP_ID, you get an error, as Example 10–15 shows.

Figure 10–2 Shareable Fields in the Repository



Example 10–15 Attempting to Drop a Domain Used by Another Database

```
SQL> ATTACH 'PATHNAME SYS$COMMON:[REPOSITORY]DEPT1';
SQL> --
SQL> -- You cannot delete EMP_ID from the repository, because EMP_ID is
SQL> -- used by DEPT2 in the record BADGE_REC.
SQL> --
SQL> DROP DOMAIN DEPT1.EMP_ID;
%RDB-E-NO_META_UPDATE, metadata update failed
-RDMS-F-RELEXI, field EMP_ID is used in relation BADGE_REC
-RDMS-F-FLDNOTDEL, field EMP_ID has not been deleted
```

Although you cannot use SQL to delete EMP_ID because EMP_ID is used by both ID_NUMBER and BADGE_NUM, you can delete the association between EMP_ID in the repository and the database that uses EMP_ID.

You delete the association between EMP_ID and BADGE_NUM by attaching to a database by path name and issuing a DROP DOMAIN statement, as Example 10–16 shows. The field, EMP_ID, remains in the repository for other Oracle Rdb databases that share this field.

Example 10–16 Using the DROP DOMAIN Statement to Delete a Link with a Database

```
SQL> DROP DOMAIN DEPT1.BADGE_NUM;
SQL> COMMIT;
SQL> DISCONNECT DEFAULT;
SQL> EXIT
$ !
$ ! Invoke CDO to check if EMP_ID remains in the repository:
$ !
$ REPOSITORY
Welcome to CDO V2.3
The CDD/Repository V5.3 User Interface
Type HELP for help
CDO> DIRECTORY
 Directory SYS$COMMON:[REPOSITORY]DEPT1
BADGE_NUM(1)                                  FIELD
EMPLOYEE_REC(1)                               RECORD
DEPT1(1)                                      CDD$DATABASE
EMP_ID(1)                                     FIELD
```

```
FIRST_NAME(1)                            FIELD
LAST_NAME(1)                             FIELD
CDO> EXIT
```

You can also use the ALTER TABLE statement with the DROP COLUMN clause to remove the association between the shareable field EMP_ID and the database DEPT1. The field EMP_ID remains in the repository so it can be used by other definitions. Example 10–17 uses ALTER TABLE with the DROP COLUMN clause.

Example 10–17 Using the ALTER TABLE Statement to Delete a Link with a Database

```
CDO> ! Display the record definition EMPLOYEE_REC.
CDO> !
CDO> SHOW RECORD EMPLOYEE_REC(1)
Definition of record EMPLOYEE_REC
|   Contains field          EMP_ID
|   Contains field          LAST_NAME
|   Contains field          FIRST_NAME
CDO> EXIT
$ !
$ ! Invoke SQL:
$ SQL
SQL> ATTACH 'PATHNAME SYS$COMMON:[REPOSITORY]DEPT1';
SQL> --
SQL> -- Display the table EMPLOYEE_REC from database DEPT1.
SQL> --
SQL> SHOW TABLES (COLUMNS) DEPT1.EMPLOYEE_REC;
Information for table DEPT1.EMPLOYEE_REC

CDD Pathname: SYS$COMMON:[REPOSITORY]EMPLOYEE_REC;1

Columns for table EMPLOYEE_REC:

Column Name    Data Type        Domain
-----------    ---------        ------
EMP_ID         CHAR(9)          EMP_ID
LAST_NAME      CHAR(20)         LAST_NAME
FIRST_NAME     CHAR(10)         FIRST_NAME

SQL> --
SQL> -- Use ALTER TABLE to delete EMP_ID from the table EMPLOYEE_REC.
SQL> --
SQL> ALTER TABLE EMPLOYEE_REC
cont>      DROP COLUMN EMP_ID;
SQL> --
```

(continued on next page)

Example 10–17 (Cont.) Using the ALTER TABLE Statement to Delete a Link with a Database

```
SQL> COMMIT;
SQL> DISCONNECT DEFAULT;
SQL> EXIT
$ !
$ ! Invoke CDO again:
$ REPOSITORY
Welcome to CDO V2.3
The CDD/Repository V5.3 User Interface
Type HELP for help
CDO> !
CDO> ! Display the record definition EMPLOYEE_REC(1) from the repository.
CDO> ! Notice that the field EMP_ID is still part of the EMPLOYEE_REC(1)
CDO> ! definition.
CDO> !
CDO> SHOW RECORD EMPLOYEE_REC(1)
Definition of record EMPLOYEE_REC
|   Contains field          EMP_ID
|   Contains field          LAST_NAME
|   Contains field          FIRST_NAME
CDO> !
CDO> ! Use the DIRECTORY command to verify that EMP_ID still exists as a
CDO> ! field definition in the repository.
CDO> !
CDO> DIRECTORY
Directory SYS$COMMON:[REPOSITORY]

BADGE_NUM(1)                              FIELD
BADGE_REC(1)                              RECORD
DEPT1(1)                                  CDD$DATABASE
EMPLOYEE_REC(1)                           RECORD
EMP_ID(1)                                 FIELD
FIRST_NAME(1)                             FIELD
LAST_NAME(1)                              FIELD
CDO> !
CDO> EXIT
```

10.12.3 Deleting Repository Definitions

To delete definitions from the repository using SQL, use the DROP PATHNAME statement. The DROP PATHNAME statement does not delete the physical database files, only the CDD$DATABASE element definition in the repository, as Example 10–18 shows.

Note that if records and fields associated with the CDD$DATABASE element were not assigned a repository directory name using the ENTER command, those definitions are also deleted from the repository with the CDD$DATABASE element. Deleting definitions in this way might affect other repository elements that refer to these definitions.

Example 10–18 Deleting Definitions from the Repository Using the DROP PATHNAME Statement

```
SQL> ATTACH 'PATHNAME SYS$COMMON:[REPOSITORY]DEPT1';
SQL> DROP PATHNAME SYS$COMMON:[REPOSITORY]DEPT1;
SQL> DISCONNECT DEFAULT;
SQL> EXIT;
$ !
$ ! Invoke CDO:
$ REPOSITORY
Welcome to CDO V2.3
The CDD/Repository V5.3 User Interface
Type HELP for help
CDO> !
CDO> DIRECTORY
!
! The CDD$DATABASE definition DEPT1 has been deleted, but the record
! and field definitions remain.
!
Directory SYS$COMMON:[REPOSITORY]

BADGE_NUM(1)                              FIELD
EMPLOYEE_REC(1)                           RECORD
EMP_ID(1)                                 FIELD
FIRST_NAME(1)                             FIELD
LAST_NAME(1)                              FIELD
```

10.13 Using CDO to Delete Repository Definitions

Example 10–19 shows how to use the CDO SHOW USES command to determine if EMP_ID is used by another definition in the repository. If EMP_ ID is used by other definitions, you get an error message if you try to delete EMP_ID. Before you can delete EMP_ID from the repository, you must first delete its owners: EMPLOYEE_REC, BADGE_NUM, and DEPT1.

Example 10–19 Determining Owners of a Repository Field Definition

```
CDO> SHOW USES EMP_ID
Owners of SYS$COMMON:[REPOSITORY]EMP_ID(1)
|   SYS$COMMON:[REPOSITORY]BADGE_NUM(1)  (Type : FIELD)
|   |   via CDD$DATA_ELEMENT_BASED_ON
|   SYS$COMMON:[REPOSITORY]EMPLOYEE_REC(1)  (Type : RECORD)
|   |   via CDD$DATA_AGGREGATE_CONTAINS
|   DEPT1                 (Type : CDD$RDB_DATABASE)
|   |   via CDD$RDB_DATA_ELEMENT
CDO> DELETE FIELD EMP_ID.
%CDD-E-INUSE, element is the member of a relationship; it cannot be deleted
```

You can also use the CDO DELETE GENERIC command to delete the CDD$DATABASE element and its associated definitions from the repository. Example 10–20 shows how to do this.

Example 10–20 Using the CDO DELETE GENERIC Command

```
$ REPOSITORY OPERATOR
Welcome to CDO V2.3
The CDD/Repository V5.3 User Interface
Type HELP for help
CDO> !
CDO> ! Display the CDD$DATABASE element DEPT1.
CDO> !
CDO> DIRECTORY
Directory SYS$COMMON:[REPOSITORY]

DEPT1(1)                                  CDD$DATABASE
EMPLOYEE_REC(1)                           RECORD
FIRST_NAME(1)                             FIELD
LAST_NAME(1)                              FIELD
CDO> !
CDO> ! Using the DELETE GENERIC command to delete DEPT1.
CDO> !
CDO> DELETE GENERIC CDD$DATABASE DEPT1.
CDO> !
CDO> ! Note that DEPT1 has been deleted from the repository.
CDO> !
CDO> DIRECTORY
Directory SYS$COMMON:[REPOSITORY]

EMPLOYEE_REC(1)                           RECORD
FIRST_NAME(1)                             FIELD
LAST_NAME(1)                              FIELD
```

10.14 Changing the Database File Name Using the Repository

If you change the file name of the physical database file or you move the physical database file to a different directory, you must also change the file name (CDD$FILE) as it is stored in the repository.

The repository stores the file name as a CDD$FILE element. The CDD$FILE element is owned by the CDD$DATABASE element with the same name as the database.

To change the file name of a database definition in the repository, you must do the following:

1. Create a new directory in the repository. A separate directory is needed to assign a new directory name to MCS_BINARY. The repository does not allow two elements with the same name in the same directory.
2. Use the CDO ENTER GENERIC command to assign a directory name to the MCS_BINARY element.
3. Use the CDO CHANGE GENERIC command specifying the new path name.

Example 10–21 shows how to change the file name of database DEPT3.

Example 10–21 Changing the Database File Name in the Repository

```
CDO> SHOW DATABASE DEPT3
Definition of database  DEPT3
|   database uses RDB database DEPT3
|   database in file DEPT3
|   |   fully qualified file USER1:[TEST]DEPT3.RDB;
CDO> !
CDO> SHOW GENERIC CDD$DATABASE DEPT3
Definition of DEPT3   (Type : CDD$DATABASE)
|   MCS_allowConcurrent          1
|   Contains CDD$DATABASE_SCHEMA
|   |   DEPT3                        (Type : CDD$RDB_DATABASE)
|   Contains CDD$DATABASE_FILE
|   |   DEPT3                        (Type : MCS_BINARY)
CDO> !
CDO> ! Create the new directory.
CDO> !
CDO> DEFINE DIRECTORY CDD$DEFAULT:TEST3.
CDO> SET DEFAULT CDD$DEFAULT:TEST3
```

(continued on next page)

Example 10–21 (Cont.) Changing the Database File Name in the Repository

```
CDO> !
CDO> ! Assign a repository directory name to the MCS_BINARY type, which
CDO> ! contains the CDD$DATABASE_FILE, the file location.
CDO> !
CDO> ENTER GENERIC MCS_BINARY DEPT3
cont> FROM GENERIC CDD$DATABASE CDD$DEFAULT:DEPT3
CDO> !
CDO> SHOW GENERIC MCS_BINARY DEPT3
Definition of DEPT3   (Type : MCS_BINARY)
|   MCS_storeType                 1
|   MCS_allowConcurrent           1
|   MCS_storedIn                  USER1:[TEST]DEPT3.RDB;
CDO> !
CDO> ! Change the location of the database specified in MCS_storedIn.
CDO> CHANGE GENERIC MCS_BINARY DEPT1
cont> MCS_STOREDIN "USER1:[PRODUCTION]DEPT3.RDB".
cont> END.
CDO> !
CDO>  SHOW GENERIC MCS_BINARY DEPT3
Definition of DEPT3   (Type : MCS_BINARY)
|   MCS_storeType                 1
|   MCS_allowConcurrent           1
|   MCS_storedIn                  USER1:[PRODUCTION]DEPT3.RDB
CDO> EXIT
```

Index

"
  See Quotation mark
;
  See Semicolon (;)
<
  See Less than (<) operator
>
  See Greater than (>) operator

A

Access control entry (ACE), 9–2, 9–3, 9–5, 9–6
Access control list (ACL), 9–2, 9–6
  See also Privilege; Protection
  adding privileges to entry, 9–19
  creating, 9–5 to 9–17
    privilege needed, 9–5, 9–14
  creating new entry, 9–19, 9–20
  deleting entry, 9–19
  modifying, 9–19
    effect on transaction share mode, 9–20
    effect on user, 9–20
    using command procedure, 9–21e
  organizing, 9–17 to 9–22
  removing privilege from entry, 9–19
Access privilege set, 9–2
  building, 9–14
  deleting entry, 9–19
ACE
  See Access control entry (ACE)
ACL
  See Access control list (ACL)
ACL-style privilege, 9–3
  advantages of, 9–8
  compared to ANSI/ISO-style, 9–6
Activity rates
  of transactions, 2–10
ADD CONSTRAINT clause, 8–4, 8–18e
ADD domain-constraint clause, 8–2
ADD JOURNAL clause, 7–25
ADD STORAGE AREA clause, 7–42, 7–44
  privilege required, 9–13
ADJUSTABLE LOCK GRANULARITY clause
  modifying, 7–29
After-image journal (.aij) file, 1–8, 3–12, 7–1, 7–19
  adding, 7–25
  creating, 7–25
  disabling for
    write-once storage area, 4–34
  enabling, 7–23
  extensible, 3–12, 7–25
  fixed-size, 3–12, 7–25
  modifying allocation, 7–26
  reserving, 3–7, 3–13, 7–19
  setting allocation, 7–24
  size of, 7–24
  write-once storage area and, 4–34
.aij file
  See After-image journal (.aij) file
Alias, 3–12, 7–87
  in multischema database, 5–9
  in REVOKE statement, 9–19
Allocation
  of journal file, 7–24
    modifying, 7–26

ALLOCATION clause, 3–20
ALTER STORAGE AREA clause, 7–41
for snapshot file, 7–38
hashed index and, 4–22
Allocation size
calculating for storage area, 4–52, 6–15
ALTER DATABASE statement, 7–2, 7–3, 7–44e
ADJUSTABLE LOCK GRANULARITY clause, 7–29
ALTER STORAGE AREA clause, 7–54
BUFFER SIZE clause, 7–33
CARRY OVER LOCKS clause, 7–29, 7–31
creating journaling file, 7–25
DICTIONARY IS NOT USED clause, 10–36
DICTIONARY IS REQUIRED clause, 7–40
enabling journaling, 7–23
EXTENT IS clause, 7–27
GLOBAL BUFFERS clause, 7–33
JOURNAL FAST COMMIT clause, 7–26
LOCKING IS PAGE LEVEL clause, 7–32
LOCKING IS ROW LEVEL clause, 7–32
LOCK TIMEOUT clause, 7–29, 7–31
METADATA CHANGES clause, 3–16, 7–18
modifying database characteristics, 7–19
modifying storage area, 7–2
NUMBER OF BUFFERS IS clause, 6–14, 7–33
NUMBER OF CLUSTER NODES IS clause, 7–19, 7–29
NUMBER OF RECOVERY BUFFERS clause, 7–34
NUMBER OF USERS IS clause, 7–19, 7–28
privilege required, 9–10, 9–13
RESERVE JOURNALS clause, 7–25
restriction, 7–19
restructuring database, 7–2
SNAPSHOT ALLOCATION IS clause, 7–38, 7–39
SNAPSHOT EXTENT IS clause, 7–38
SNAPSHOT IS clause
DISABLED, 7–36
ENABLED, 7–34
ENABLED DEFERRED, 7–36, 7–37
ENABLED IMMEDIATE, 7–38
ALTER DOMAIN statement, 8–1
dropping default value, 8–2
in multischema database, 8–3
privilege required, 9–12
ALTER INDEX statement, 7–3, 7–44e
disabling maintenance, 7–63
MAINTENANCE IS DISABLED clause, 7–63
multifile database and, 4–6, 4–8
privilege required, 9–12
Altering
See Modifying
ALTER STORAGE AREA clause, 7–40, 7–54
See also CREATE DATABASE statement
ALLOCATION IS clause, 7–41
EXTENT IS clause, 7–41
LOCKING IS PAGE LEVEL clause, 7–32
LOCKING IS ROW LEVEL clause, 7–32
privilege required, 9–13
ALTER STORAGE MAP statement, 7–2, 7–9, 7–44e, 7–65
See also CREATE STORAGE MAP statement
PARTITIONING IS (NOT) UPDATABLE clause, 4–12, 7–66
privilege required, 9–13
reloading existing data with, 7–9
REORGANIZE clause, 7–70
ALTER TABLE statement, 8–4, 8–6e, 8–8e
ALTER COLUMN clause, 8–7, 8–10, 8–12, 8–13
default value
dropping, 8–16
DROP COLUMN clause, 8–6, 8–7
in multischema database, 8–17
privilege required, 9–11, 9–12, 9–13
ANSI/ISO quoting, 5–9
ANSI/ISO-style privilege, 9–3
advantages of, 9–8
compared to ACL-style, 9–6
Archiving
event rows, 2–14
strategy, 2–14
transactions, 2–14

Area
  See Storage area
Arithmetic, date, 3–43, 3–90
Asynchronous storage area creation, 3–15
Attaching to database, 1–10, 3–5
  privilege required, 9–9, 9–19
ATTACH statement, 1–10e
  PATHNAME clause, 3–5
  privilege required, 9–9
Attributes
  defined, 1–4
Audit journal
  in database security, 9–2
Authorization identifier, 5–6

B

Backing up a database, 7–3
BASIC program
  load data with, 6–20
Batch_Update transaction type
  load operation and, 6–36
BIGINT data type, 3–23
B-tree index
  See Sorted index
Buffer
  global, 3–19, 3–20
    modifying, 7–33
  local, 3–19, 3–20
    modifying, 7–33
  recovery
    modifying, 7–34
Buffer size, 4–14, 4–53, 4–56
  loading data and, 6–3
  modifying, 7–33
  specifying, 6–3, 6–14
Built-in function, 3–32
Business requirements
  data collection for, 2–2
  in logical design, 2–1
BYPASS privilege, 9–34

C

Cache
  row-level, 3–19
Calculating date, 3–43, 3–90e
Calculating dbkeys, 6–5
CARRY OVER LOCKS clause, 7–29, 7–31
CASCADE keyword
  DROP STORAGE AREA statement, 7–54
  DROP TABLE statement, 7–56, 8–4
Cascading delete, 3–55
  DROP STORAGE AREA, 7–54
  DROP TABLE statement, 8–4
  in dropping catalog, 8–23
  in dropping schema, 8–22
CAST function, 3–90e
Catalog, 3–4
  creating, 5–3
    privilege required, 9–11
  definition of, 5–1
  deleting, 8–23
    privilege required, 9–11
  dropping
    in multischema database, 8–24
CDD$FILE
  modifying name of, 10–43
CDD/Repository
  See Repository
CDO (Common Dictionary Operator)
  See REPOSITORY OPERATOR command
Changing
  See Modifying
Character length
  in interactive SQL, 3–25
  setting, 3–25
Character set
  column and, 3–42
  CREATE DATABASE statement and, 3–11
  CREATE TABLE statement and, 3–42
  database and, 3–11
  default, 3–11
  domain and, 3–31
  multiple, 3–10

Character set (cont'd)
specifying CHARACTER length, 3–25
specifying OCTET length, 3–25
CHAR data type, 3–23, 3–30e
CHECK constraint, 3–34, 3–45, 3–47, 3–51
creating, 3–47e, 3–48e
CHECK OPTION clause
of CREATE VIEW statement, 3–84
Circular .aij file, 3–12
See also After-image journal (.aij) file
C language
load data with, 6–28
Closing database, 3–21
Clump page organization
in storage area with uniform page format, 4–14
Clustering rows, 4–7, 4–15, 4–26, 7–44, 7–77
by index values
to improve join, 4–24
duplicate child row, 4–56
for exact match retrieval, 4–21
load operation and, 4–25
page size and, 4–22
to reduce I/O, 7–61
using hashed index, 4–57 to 4–63
Cluster nodes
modifying number of, 7–19, 7–29
Collating sequence, 3–3, 3–28
creating
privilege required, 9–11
deleting
privilege required, 9–11
specifying, 3–33
Column
adding, 8–8
character set, 3–42
COMPUTED BY clause, 3–42
data type, 3–40
default value, 3–44
dropping, 8–16
modifying, 8–13
defined, 1–1, 3–1
defining protection for, 9–24e
deleting, 8–4, 8–6, 8–7
dividing among storage areas, 4–28
Column (cont'd)
dropping, 8–4, 8–6, 8–7
privilege required, 9–11
elements of, 3–38
ensuring unique values in, 3–65, 3–67
modifying, 8–10
COMPUTED BY clause, 8–7
definition of, 8–4
name of, 8–16
position of, 8–16
privilege required, 9–11
with constraint, 8–18
replacing, 8–7, 8–8
Column constraint, 3–45
See also Constraint
Command procedure (SQL)
advantages of using, 3–5
online example, 1–10
COMMENT ON statement, 3–26
COMMIT statement
with CREATE DATABASE statement, 3–10
Compressed index, 3–70, 3–71, 3–72
sorted duplicate, 3–62
system, 3–16, 7–19
Compression
storage areas and, 4–10, 4–29
COMPUTED BY clause, 3–42
deleting table and, 8–5
limitations, 8–7
Computing date, 3–90e
Concurrency
in index definition, 3–69
snapshot file and, 3–18
Concurrent access
in data definition, 7–10t
Configuration parameter
RDB_BIND_BUFFERS, 6–3, 6–14
RDB_BIND_LOCK_TIMEOUT, 7–31
RDB_BIND_SEGMENTED_STRING_BUFFER, 4–30
Constraint
ADD CONSTRAINT clause, 8–4, 8–18e
adding, 8–19
privilege required, 9–11
CHECK, 3–47

Constraint (cont'd)
column, 3–45
adding, 8–4
creating, 8–4
compared with index, 3–65
compared with trigger, 3–56
creating, 3–39, 3–45 to 3–48, 8–19
CHECK, 3–48e
privilege required, 9–11
defined, 3–1
defining in repository, 10–14
displaying, 3–48, 8–18e
domain, 3–28
adding, 8–2
creating, 3–34
dropping, 8–2
modifying, 8–2
DROP CONSTRAINT clause, 8–18e
dropping, 3–50, 8–4, 8–19
privilege required, 9–12
evaluating, 3–46
during load operation, 6–35
FOREIGN KEY, 3–45, 3–46, 3–47
loading data and, 6–2, 6–15, 6–47, 6–50
modifying, 3–48, 8–18
multischema database and, 5–11
naming, 3–45, 3–47e, 3–48e
NOT NULL, 3–46
NULL, 3–51
PRIMARY KEY, 3–46
referential integrity and, 3–53, 3–54
repository and, 3–37
table, 3–45
adding, 8–4
creating, 8–4
temporary table and, 3–74
trade-offs, 3–51
TRUNCATE TABLE and, 8–6
UNIQUE, 3–46, 3–51
violation of, 8–19
with CHECK clause, 3–45, 3–51
Copying a database, 7–9, 7–82
COUNT clause
ADJUSTABLE LOCK GRANULARITY clause and, 7–30
COUNT function
in computed column, 3–43
CPU usage
during load, 6–18
CREATE CATALOG statement, 5–3
privilege required, 9–11
CREATE COLLATING SEQUENCE statement, 3–33
privilege required, 9–11
CREATE DATABASE statement, 3–6, 4–28
character set, 3–11
compressing system indexes, 3–16
CREATE STORAGE AREA clause, 3–14
DEFAULT STORAGE AREA clause, 3–15
defining database protection, 3–9
DICTIONARY IS REQUIRED clause, 3–5, 10–23
error condition following rollback, 3–10
METADATA CHANGES clause, 3–16, 7–18
multischema option, 5–2
MULTITHREADED AREA ADDITIONS clause, 3–15
OPEN IS clause, 3–21
PATHNAME clause, 3–5
RESERVE STORAGE AREAS clause, 3–14
semicolon (;) in, 3–9
SNAPSHOT IS ENABLED clause, 3–17
SYSTEM INDEX COMPRESSION clause, 3–16
with subordinate element definition, 3–9
CREATE DOMAIN statement, 3–27
in multischema database, 5–10
privilege required, 9–12
using character set, 3–31
CREATE FUNCTION statement, 3–58
privilege required, 9–12
CREATE INDEX statement, 3–60, 4–34
multifile database and, 4–6, 4–8
NODE SIZE clause, 4–35
partitioning index across storage areas, 4–8e
PERCENT FILL clause, 4–38
privilege required, 9–12
RANKED keyword, 3–61
STORE clause, 4–1, 4–7
storing hashed index, 4–23e

CREATE INDEX statement (cont'd)
USAGE clause, 4–38
CREATE MODULE statement, 3–3
privilege required, 9–12
CREATE OUTLINE statement
privilege required, 9–12
CREATE PROCEDURE statement
privilege required, 9–12
Create Routine statement, 3–4
See also CREATE FUNCTION statement, CREATE PROCEDURE statement
CREATE SCHEMA statement, 5–3
privilege required, 9–12
CREATE STORAGE AREA clause, 3–14, 4–1, 4–26
for hashed index, 4–23e
CREATE STORAGE MAP statement, 4–1, 4–7, 7–72
clustering rows from different tables, 4–26e
partitioning, 4–4, 4–6
horizontal, 4–8e
vertical, 4–10, 4–28e
PARTITIONING IS (NOT) UPDATABLE clause, 4–11
PLACEMENT VIA INDEX clause, 7–80
hashed index, 4–21
privilege required, 9–13
specifying options, 4–10
CREATE TABLE statement, 3–38
in multischema database, 5–11
privilege required, 9–13
temporary, 3–72
using character set, 3–42
using repository, 3–37
CREATE TEMPORARY TABLE statement, 3–72
CREATE TRIGGER statement, 3–55
in multischema database, 5–14
privilege required, 9–13
CREATE VIEW statement, 3–83
in multischema database, 5–13
privilege required, 9–13
Creating constraint
column, 8–4
table, 8–4
Creating database
multifile, 3–14
restricting ability to, 9–46
snapshot file, 3–17
storage design, 4–1
Creating domain, 3–27 to 3–36
in multischema database, 5–10
Creating hashed index, 7–80
Creating repository definition
using SQL, 10–28e
Creating storage area, 3–14, 4–1, 7–42, 7–80
default, 3–8
system area, 3–8
Creating storage map, 3–14, 4–1, 7–80
Creating table, 3–36, 3–38, 3–42
from repository, 3–37
with repository, 3–36
Creating temporary table, 3–72
Creating trigger, 3–54
Creating view, 3–83
CURRENT_DATE function, 3–44
CURRENT_TIME function, 3–32, 3–44
CURRENT_TIMESTAMP function, 3–32, 3–44, 3–54, 3–55e
CURRENT_USER function, 3–32, 9–27
Cursor
opening
privilege required, 9–10

D

Data
calculating size of data row, 4–47, 4–49
loading, 6–1 to 6–61
from flat file, 6–38, 6–39
modifying definition, 6–60
strategy for, 6–1
troubleshooting, 6–16
with BASIC program, 6–20
with RMU Load command, 6–32, 6–45, 6–46, 6–51
constraint and, 6–47, 6–50
from table with fewer columns, 6–47, 6–49

Data
loading
with RMU Load command (cont'd)
from table with more columns, 6–47, 6–48
with SQL module language, 6–20
with SQL precompiled C program, 6–28
unloading
record definition format, 6–37
to flat file, 6–39
with RMU Unload command, 6–32, 6–45, 6–46
Data access
methods of, 1–9
Database
allocating resources for, 3–6
attaching to, 3–5
backing up
before restructuring, 7–3
character set, 3–11
copying, 7–9, 7–82
creating, 3–4, 3–6, 3–11
See also CREATE DATABASE statement
multifile, 3–14
using repository, 3–5
with subordinate element definition, 3–9
creating index, 3–60, 3–62e, 3–63e, 3–65e
creating table, 3–36, 3–38, 3–40e, 3–48e
in multischema database, 5–11
creating temporary table, 3–72
creating view, 3–83
declaring, 3–5
defining protection for, 3–9, 9–1 to 9–34
definition of, 3–3
deleting, 3–10, 7–87
privilege required, 9–10
dropping link with repository, 10–36
duplicating metadata, 7–84
exporting with no data, 7–84
files, 4–2
.aij, 1–8
.rbr, 7–84
.rda, 1–7, 3–7
.rdb, 1–7, 3–7
.ruj, 1–8
Database
files (cont'd)
.snp, 1–7
importing
using TRACE clause and, 7–84
with no data, 7–84
invoking, 3–5
limiting number of simultaneous users, 3–9
loading data
hashed index and, 4–23
modifying database definition, 6–60
strategy for, 6–1
troubleshooting, 6–16
with RMU Load command, 6–32, 6–45, 6–46, 6–51
constraint and, 6–47, 6–50
from table with fewer columns, 6–47, 6–49
from table with more columns, 6–47, 6–48
logical design entities, 1–4
logical organization of, 2–8
maintaining, 7–1 to 7–87
metadata update
online, 7–10
modifying, 7–1 to 7–87
See also ALTER DATABASE statement
modifying before load operation, 6–13
modifying characteristics, 7–3t, 7–19 to 7–75
modifying metadata
online, 7–10
moving, 7–85
using EXPORT and IMPORT statement, 7–86
multischema, 5–1
creating, 5–2
opening, 3–21
parameter
for load operation, 6–13
protection of
using view, 3–83
reorganizing, 7–75
requirements analysis of, 2–1
restructuring
to change column name or order, 8–16

Database (cont'd)
snapshot (.snp) file, 1–7, 3–17
unloading data
record definition format, 6–37
with RMU Unload command, 6–32, 6–45, 6–46
verifying protection for, 9–31
Database attach
privilege needed, 9–19
Database directory (.rdb) file, 1–7
Database file
updating using repository, 10–21, 10–31
Database file name
modifying in the repository, 10–43
Database handle
See Alias
Database key
access for retrieval, 1–9
calculating
in preparation for loading database, 6–5
setting scope, 3–21
sorting, 6–9
uncompressed dbkey, 4–36
using to read rows, 6–10
Database performance
snapshot file and, 7–36
using deferred snapshot file, 7–37
Database root (.rdb) file, 1–7, 3–7
Data clustering strategies
using hashed index, 4–57 to 4–63
Data collection
in requirements analysis, 2–2
Data definition
disabling, 7–18
overview, 3–1
Data definition language (DDL)
defined, 1–2
Data dictionary
See Repository
Data item (column), 1–4
Data manipulation language (DML)
defined, 1–2
Data placement strategies
using hashed index, 4–57 to 4–63
Data retrieval
methods of, 1–9
Data row, 4–41
Data type
BIGINT, 3–23
CHAR, 3–23, 3–30e
converting, 3–90e
DATE, 3–30e
DATE ANSI, 3–24, 3–43
DATE VMS, 3–24
domain, 3–30
INTEGER, 3–23, 3–30e
INTERVAL, 3–24
LIST OF BYTE VARYING, 3–23
format, 3–24
modifying, 8–12
modifying column, 8–10e
NCHAR, 3–31, 3–42
NCHAR VARYING, 3–31, 3–42
SMALLINT, 3–23
specifying, 3–23 to 3–31, 3–40
TIME, 3–24
TIMESTAMP, 3–24
VARCHAR, 3–23, 3–30e
Data value (row), 1–4
Date
storing current, 3–54, 3–55e
DATE ANSI data type, 3–24, 3–43
modifying, 8–12
Date arithmetic, 3–43, 3–90e
DATE data type, 3–24, 3–30e
date arithmetic, 3–43, 3–90e
RMU Load and, 6–39
using in computed column, 3–43
Date-time data type
modifying columns, 8–12
DATE VMS data type, 3–24
modifying, 8–12
DBADM privilege, 9–9, 9–33
DBCTRL privilege, 9–5, 9–14
Dbkey
See Database key
dbsmgr account
privilege and, 9–34

DDL
  See Data definition language
Declared local temporary table
  creating, 3–73, 3–79
DECLARE TRANSACTION statement
  privilege required for distributed transaction,
    9–10
DEC Multinational character set (MCS), 3–10
Default protection, 9–30
Default storage area, 3–8, 3–15
  table with data, 7–72
DEFAULT STORAGE AREA clause
  of CREATE DATABASE statement, 3–15
Default value
  for column, 3–44
    modifying, 8–13
  for domain, 3–32, 8–2
    deleting, 8–2
    modifying, 8–2
Deferrable constraint, 3–46
Deferred snapshot file, 7–37
DEFERRED snapshot option, 3–17
Defer_Index_Updates qualifier
  RMU Load command, 6–55
Delete
  cascading, 3–55
DELETE statement
  performance with index, 3–66
  privilege required, 9–10
Deleting catalog, 8–23
Deleting column, 8–4, 8–6, 8–7
Deleting constraint, 8–18, 8–19
  domain, 8–2
  table, 8–4
Deleting data
  quickly, 8–6
Deleting database
  See DROP DATABASE statement
Deleting domain, 8–3
  in multischema database, 8–3
Deleting index
  See DROP INDEX statement
Deleting privilege, 9–19
Deleting repository definition
  using CDO, 10–41
  using SQL, 10–36
Deleting schema, 8–22
Deleting storage area, 7–54
Deleting storage map, 7–75
Deleting table, 8–4, 8–5e
  See also DROP TABLE statement
  in multischema database, 8–17
  quickly, 8–6
Deleting trigger, 8–19
Deleting view, 8–21
  See also DROP VIEW statement
  in multischema database, 8–22
Delimited identifier, 3–22, 5–9
Designing database
  amount of effort, 1–6
  compromise, 4–4
  method, 1–3
  output of good design, 1–3
Detected asynchronous prefetch, 6–4
Dialect
  setting, 3–11, 3–24
DICTIONARY IS NOT USED clause
  ALTER DATABASE statement, 10–36
DICTIONARY IS REQUIRED clause, 3–5, 7–18
DICTIONARY OPERATOR
  See REPOSITORY OPERATOR command
Direct I/O
  load operation and, 6–18
Disk drive
  increasing number to serve a database, 4–1
  index distribution among, 4–8e
  snapshot distribution among, 4–1
  storage area distribution among, 4–1
  table distribution among, 4–3, 4–4, 4–8e,
    4–10, 4–28e
Disk space
  allocating, 3–19
  reducing, 3–70
Displaying constraint, 3–48

Displaying domain, 8–1
Distributed transaction
  privilege required, 9–10
DML
  See Data manipulation language
Domain
  based on repository, 3–27
  characteristics, 3–28
  character set, 3–31
  column based on, 3–40
  constraint, 3–28
    adding, 8–2
    creating, 3–34
    dropping, 8–2
    modifying, 8–2
  creating, 3–27 to 3–36
    in multischema database, 5–10
    privilege required, 9–12
    with date-time data type, 8–13
  data type, 3–30
  default value, 3–32, 8–2
    deleting, 8–2
    modifying, 8–2
  defined, 3–3
  dropping, 8–3, 8–8e
    in multischema database, 8–3, 8–23
    privilege required, 9–12
  integrating with repository, 10–21, 10–30,
    10–35
  modifying, 8–1
    in multischema database, 8–3
    privilege required, 9–12
    with constraint, 8–18
    with date-time data type, 8–13
  multischema database and, 5–11
  naming, 3–28
    in multischema database, 5–10
Domain constraint, 3–28
  adding, 8–2
  creating, 3–34
  dropping, 8–2
  modifying, 8–2
DROP ALL CONSTRAINTS clause, 8–2
DROP CATALOG statement, 8–23
  privilege required, 9–11
DROP COLLATING SEQUENCE statement
  privilege required, 9–11
DROP COLUMN clause, 8–6
DROP CONSTRAINT clause, 8–4, 8–18e, 8–19
DROP CONSTRAINT statement, 8–4
  privilege required, 9–12
DROP DATABASE statement, 7–87
  error after issuing CREATE DATABASE,
    3–10
  privilege required, 9–10
DROP DEFAULT clause
  of ALTER DOMAIN statement, 8–2
  of ALTER TABLE statement, 8–16
DROP DOMAIN statement, 8–3, 8–8e
  in multischema database, 8–3
  privilege required, 9–12
DROP FUNCTION statement
  privilege required, 9–12
DROP INDEX statement, 7–63, 7–64
  in multischema database, 7–65
  privilege required, 9–12
DROP MODULE statement
  privilege required, 9–12
DROP OUTLINE statement
  privilege required, 9–12
DROP PATHNAME statement, 10–40e
  error after issuing CREATE DATABASE,
    3–10
Dropping link
  between repository and database, 10–36
DROP PROCEDURE statement
  privilege required, 9–12
DROP SCHEMA statement, 8–22
  privilege required, 9–12
DROP STORAGE AREA clause, 7–54
  privilege required, 9–13
  RESERVE STORAGE AREA and, 7–54
DROP STORAGE MAP statement, 7–75
  privilege required, 9–13
DROP TABLE statement, 8–4, 8–5e
  in multischema database, 8–17
  privilege required, 9–13
  temporary table, 3–74

DROP TRIGGER statement, 8–19
  restrictions, 8–19
DROP VIEW statement, 8–21
  in multischema database, 8–22
  privilege required, 9–13
Duplicate node record, 4–41
Duplicates compression, 3–70
Duplicate value, 3–67
  error generated, 3–66
  index and, 3–63
  performance and, 3–68
  SIZE IS index and, 3–72

E

EDIT STRING clause, 3–34
  for column, 3–39
    in view definition, 3–84
  in domain definition, 3–34
Embedded SQL
  creating database with, 3–5
EMPLOYEES table
  requirements for, 1–5
Entities
  database logical design, 1–4
Entity-relationship map, 2–3
Entry points
  of transactions, 2–10
E-R map
  See Entity-relationship map
Exact match retrieval
  index use and, 3–63
  using hashed indexes, 1–9
Exclusive transaction type
  load operation and, 6–36
Executor
  parallel load and, 6–52
Exporting database
  See EXPORT statement
EXPORT statement, 7–2
  alternatives to, 7–3t
  duplicating database metadata with, 7–84
  modifying storage area with, 7–47e
  moving between different versions of
    databases, 7–86
EXPORT statement (cont'd)
  moving database with, 7–86
  privilege required, 9–13
  reorganization with, 7–47, 7–75, 7–76, 7–79
  WITH EXTENSIONS option, 7–75
  with no data, 7–84
Extent
  of journal file, 7–24
EXTENT IS clause
  ALTER STORAGE AREA clause, 7–41
  for snapshot file, 7–38
  of ALTER DATABASE statement, 7–27
Extent value
  enabling, 7–27
  modifying, 7–27, 7–41
    storage area, 7–41
External function
  See also External routine
  trigger and, 3–58
External procedure
  See *External routine*
External routine, 3–4
  executing
    privilege required, 9–9
EXTRACT function, 3–90
Extracting data definition, 6–47, 6–48e
Extracting date field, 3–90

F

Fanout factor
  for adjustable lock granularity, 7–30
FAST COMMIT option, 6–3
  modifying, 7–26
Fields qualifier
  RMU Load command, 6–48
File space
  extending, 7–27, 7–41
    for storage area, 7–41
File type
  .aij, 1–8, 3–7, 3–12
  .rbr, 7–84
  .rda, 1–7, 3–7
  .rdb, 1–7, 3–7
  .rrd, 6–33

File type (cont'd)
.ruj, 1–8
.snp, 1–7, 3–8
.unl, 6–33
First normal form, 2–9
See Normalization
Fixed page overhead, 4–41, 4–42
Flat file
loading data from, 6–19 to 6–61
using RMU Load command, 6–32, 6–38
restriction, 6–39
using SQL module language and BASIC, 6–20
using SQL precompiled COBOL program, 6–24
using SQL precompiled C program, 6–28
using SQL program, 6–19 to 6–32
unloading data
using RMU Unload command
restriction, 6–39
Foreign key
finding, 2–2
referential integrity and, 3–54
FOREIGN KEY constraint, 3–45, 3–46, 3–47
Formatting clause, 3–34
for table, 3–39
in domain definition, 3–28
Fullness percentage
index and, 4–38
Fully normalized
See Third normal form
Function
built-in, 3–32
CAST, 3–90e
COUNT, 3–43
creating
privilege required, 9–12
CURRENT_DATE, 3–44
CURRENT_TIME, 3–44
CURRENT_TIMESTAMP, 3–44, 3–54
CURRENT_USER, 9–27
dropping
privilege required, 9–12
external, 3–4
trigger and, 3–58
Function (cont'd)
EXTRACT, 3–90
improving performance using index, 3–67

G

General identifier, 9–6
Global buffer, 3–19, 3–20
modifying, 7–33
GLOBAL BUFFERS clause
modifying, 7–33
Global temporary table
creating, 3–73, 3–75
GRANT OPTION clause, 9–7
GRANT statement, 9–2, 9–19
differences between ANSI/ISO and ACL style, 9–2, 9–6
effect on transaction share mode, 9–20
effect on user, 9–20
privilege required, 9–10
Greater than (>) operator
sorted index and, 3–61

H

Hash addressing
See Hashed index
Hash bucket, 4–41
Hashed index, 1–10, 3–61, 3–63, 4–41
algorithms for storing index keys, 3–64
calculating fixed and variable page overhead, 4–42
calculating page size, 4–41, 6–15
data rows, 4–47
example, 4–51
hash index structures, 4–43
calculating size of duplicate node record, 4–43
calculating size of hash bucket, 4–43
calculating size of hash index structures
example, 4–45
calculating size of system record, 4–43
calculating storage area allocation, 4–40
calculating storage area size, 4–52, 6–15
compressing, 3–70
creating, 7–80

Hashed index (cont'd)
database parameters and, 4–40
data clustering strategies, 4–57 to 4–63
data placement strategies, 4–57 to 4–63
estimating number of SPAM pages, 4–54
guidelines for designing, 4–21
horizontal partitioning, 7–61
load operation and, 4–23, 6–15
modifying, 7–60
overflow partition, 7–62
page format required for storing, 4–15
retrieval for exact matches, 1–9
using to place rows, 4–21
Hashed index structures, 4–41
HASHED ORDERED clause, 3–64
restrictions, 3–64
Hash insert value
during load, 6–18
Hidden delete
See Cascading delete
Horizontal partitioning, 1–8, 4–4, 7–61

I

I/O
See Input/output
Identifier, 3–22
delimited, 3–22, 5–9
general, 9–6
system-defined, 9–7
user, 9–6
IMMEDIATE snapshot option, 3–17, 7–38
IMPORT statement, 7–2, 7–47e
DEFAULT STORAGE AREA clause, 3–15
duplicating database metadata with, 7–84
METADATA CHANGES clause, 7–18
reorganization with, 7–47, 7–75, 7–76, 7–79
restricting access, 7–2
TRACE clause, 7–84
with no data, 7–84
Index
See also Hashed Index, Sorted Index
adding partition, 4–9, 7–62
algorithms for storing hashed index keys, 3–64
Index (cont'd)
based on multiple columns, 3–69
B-tree
See Sorted index
changes affect storage maps, 7–62
clustering, 6–2, 6–4
compressed, 3–70, 3–71, 3–72
compressing system, 3–16, 7–19
creating, 3–60 to 3–65e, 4–1
assigning to storage area, 4–7
concurrently, 3–69
guidelines, 4–6, 4–8
hashed, 4–8, 4–40
privilege required, 9–12
sorted, 4–34
deleting, 7–63, 7–64
in multischema database, 7–65
privilege required, 9–12
disabling, 7–63
privilege required, 9–12
distribution among storage areas, 4–3
dropping, 7–63, 7–64
in multischema database, 7–65
duplicate values and performance, 3–68
guidelines for creating, 3–67
improving performance for, 3–67
loading data and, 6–2, 6–15
modifying, 7–3, 7–58
hashed, 7–60
privilege required, 9–12
sorted, 7–59
multisegmented, 4–39
parallel load and, 6–55
partitioning across multiple storage areas, 4–8e, 4–26, 7–61
performance and, 3–66, 3–67
reorganizing database
definition sequence, 7–77
sorted, 3–61
duplicate compression, 3–62
load operation, 6–15
nonranked
calculating node size, 4–36
ranked
calculating node size, 4–35

Index (cont'd)
specifying storage area for, 4–7
storing
hashed, 4–23e
in CREATE INDEX statement, 4–24
trade-offs of, 3–66
unique, 3–65
Index key compression
See Run-length compression
Input/output
identifying bottlenecks, 2–12
increasing, 7–59
redistribution of loading, 7–86
reducing, 4–1, 4–3, 4–20, 4–21, 4–52
using clump storage, 4–14
using clustering, 4–24, 4–26, 4–60, 7–61
using global buffers, 7–33
using hashed index, 4–15, 4–55, 7–60
using hashed versus sorted index, 3–63
using index, 4–39
using partitioning, 4–7, 7–61
using PLACEMENT clause, 4–59
using shadowing, 4–61
using sorted index, 7–59, 7–60
tracing, 7–85
INSERT statement
performance with index, 3–66
privilege required, 9–9
using to calculate dbkeys, 6–5
using to load data, 6–6
INTEGER data type, 3–23, 3–30e
INTEGRATE statement
ALTER DICTIONARY clause, 10–23
ALTER FILES clause, 10–31e
CREATE PATHNAME clause, 10–28e
creating repository definitions, 10–22, 10–28
DOMAIN clause, 10–30, 10–35
privilege required, 9–13
TABLE clause, 10–30, 10–35
updating database file, 10–21, 10–31
updating repository, 10–22, 10–23, 10–24e
Interactive SQL
column header, 3–34
formatting
in view definition, 3–84
Interactive SQL (cont'd)
formatting clause
in domain definition, 3–34
Interchange (.rbr) file, 7–2
Internationalization
Oracle Rdb support for, 3–33
INTERVAL data type, 3–24

J

Join
improving performance with index, 3–67
JOURNAL ALLOCATION IS clause
modifying, 7–26
specifying, 7–24
JOURNAL EXTENT IS clause
specifying, 7–24
JOURNAL FAST COMMIT clause
modifying, 7–26
Journaling, 3–12
after-image journal file, 7–23
adding, 7–25
modifying allocation, 7–26
setting allocation, 7–24
disabling for
write-once storage area, 4–34
enabling, 7–23
FAST COMMIT option, 7–26
metadata update, 7–18, 7–19, 7–23, 7–25
write-once storage area and, 4–34
modifications not journaled, 7–19
reserving slots for, 3–7, 3–13
JOURNAL IS ENABLED clause, 7–23

K

Kanji character set
support for, 3–11
Key, 2–2
See also Foreign key; Primary key

L

Length of character
  setting, 3–25
Less than (<) operator
  sorted index and, 3–61
LIMIT OF clause, 4–8
LIST OF BYTE VARYING data type, 3–23
  adding to write-once storage area, 7–52
  format, 3–24
  moving to read/write storage area, 7–52
  moving to write-once storage area, 7–50
  performance issues, 4–30, 4–32
  storage of, 4–16, 4–30, 4–32
  storing in write-once storage area, 4–17
  storing on WORM optical device, 4–33
Loading data, 6–1 to 6–61
  clustering
    with a hashed index, 6–4
    with a sorted index, 6–2
  defining database, 6–13
  from flat file, 6–19 to 6–61
    using RMU Load command, 6–32, 6–38
      restriction, 6–39
    using SQL module language and BASIC, 6–20
    using SQL precompiled COBOL program, 6–24
    using SQL precompiled C program, 6–28
    using SQL program, 6–19 to 6–32
  index and, 6–2
  modifying database definition, 6–60
  null value, 6–40
  parallel, 6–1, 6–2, 6–17, 6–52 to 6–60
  PLACEMENT ONLY RETURNING DBKEY clause, 6–5
  setting database parameter, 6–13
  sorting, 6–2, 6–34
  troubleshooting, 6–16
  using repository definition, 6–24
  using RMU Load command, 6–46, 6–51
    constraint and, 6–47, 6–50
    from table with fewer columns, 6–47, 6–49
Loading data
  using RMU Load command (cont'd)
    from table with more columns, 6–47, 6–48
    improving performance, 6–34
  using RMU Unload command, 6–32, 6–45
Local buffer, 3–19, 3–20
  modifying, 7–33
Local temporary table
  creating, 3–73, 3–75
Lock
  disabling carry-over locks, 7–31
  enabling carry-over locks, 7–31
  specifying page-level locking, 7–32
  specifying row-level locking, 7–32
Lock conflict
  reducing, 7–36, 7–59
Locking
  effect of GRANT statement, 9–20
  effect of REVOKE statement, 9–20
  modifying characteristics, 7–29
  when restructuring database, 7–10
LOCKING IS PAGE LEVEL clause, 7–32
LOCKING IS ROW LEVEL clause, 7–32
LOCK TIMEOUT INTERVAL IS clause, 7–29, 7–31
Logical area, 4–13
Logical data, 1–4t
Logical design
  defined, 1–4
  requirements for, 1–4
  techniques of, 2–1
  tools, 2–1
Logical name
  RDBVMS$CREATE_DB, 9–46
  RDM$BIND_BUFFERS, 6–3, 6–14
  RDM$BIND_LOCK_TIMEOUT, 7–31
  RDMS$BIND_SEGMENTED_STRING_BUFFER, 4–30

M

Maintaining database, 7–1 to 7–87
MAINTENANCE IS DISABLED clause, 7–63
  privilege required, 9–12
Many-to-many relationship, 2–6
Mapping values compression, 3–70
  benefits, 3–71
MCS
  See DEC Multinational Character Set (MCS)
Memory
  allocating, 3–19
  temporary table and, 3–82
Metadata
  copying, 7–84
  creating using SQL, 10–28
  defined, 1–2
  definition of, 10–1
  deleting, 10–36
  examples of, 1–2
  integrating with database, 10–21
  listing, 1–2
  modifying, 10–21
  modifying using CDO, 10–17
  modifying using SQL, 10–23
METADATA CHANGES clause, 3–16, 7–18
Metadata update
  journaling of, 7–18, 7–19, 7–23, 7–25
    write-once storage area and, 4–34
  online, 7–10
Method
  design, 1–3
mf_personnel database
  creating, 1–10
  example
    requirements for, 1–5
MIA
  See Multivendor Integration Architecture (MIA)
Missing value
  loading from flat file, 6–40
Mixed page format, 4–14
Modifying .aij allocation size, 7–24, 7–26
Modifying buffer
  global, 7–33
  local, 7–33
  recovery, 7–34
Modifying column, 8–4, 8–7
  constraint, 8–19
  data type, 8–10
  default value, 8–13
  with date-time data type, 8–12
Modifying constraint, 8–18
Modifying database, 7–1 to 7–87
  EXPORT statement
    with no data, 7–84
  IMPORT statement, 7–84
    with no data, 7–84
  online, 7–10
  reorganizing, 7–75
  storage parameter, 7–40
Modifying database characteristics, 7–19 to 7–75
Modifying database file
  with INTEGRATE statement, 10–21
Modifying domain, 8–1
  in multischema database, 8–3
Modifying file space extent, 7–27, 7–41
Modifying index, 7–44e, 7–58
  hashed, 7–60
  sorted, 7–59
Modifying JOURNAL FAST COMMIT option, 7–26
Modifying lock characteristics, 7–29
Modifying maximum number of users, 7–19, 7–28
Modifying memory usage, 7–40
Modifying metadata, 10–21
Modifying number of nodes, 7–19, 7–29
Modifying repository definition, 10–17, 10–19
Modifying snapshot file, 7–34
  allocation size, 7–34, 7–38, 7–39
  deferred, 7–37
  disabling, 7–34
  enabling, 7–34
  extent, 7–38
  immediate, 7–38

Modifying storage area, 7–40, 7–44e, 7–47e
  moving, 7–48
  RDB$SYSTEM, 7–47
Modifying storage map, 7–2, 7–9, 7–65, 7–66
Modifying storage parameter, 7–3t, 7–40
Modifying table, 8–4
  in multischema database, 8–17
Modifying trigger, 8–19
Modifying view, 8–21
Module
  creating
    privilege required, 9–12
  dropping
    privilege required, 9–12
  executing
    privilege required, 9–9
  stored
    See Stored procedure
MONITOR DISK command (DCL), 4–3
Moving database, 7–85
Moving storage area, 7–9, 7–48
Multifile database, 1–6, 1–7, 3–14, 4–1 to 4–63
  defining index in, 4–6, 4–8
  defining storage map in, 4–6, 4–8
  sample personnel database, 1–10
  storage design, 4–1
Multiprocess load
  See Parallel load
Multischema database
  creating, 5–2 to 5–14
  definition of, 5–1
  deleting
    catalog, 8–23
    domain, 8–3
    index, 7–65
    schema, 8–22
    table, 8–17
    view, 8–22
  modifying
    domain, 8–3
    table, 8–17
  naming element, 5–5
  using qualified name, 5–6
Multisegmented key
  See Primary key
MULTITHREADED AREA ADDITIONS clause
  of CREATE DATABASE statement, 3–15
Multiuser environment
  designing for, 1–1
Multivendor Integration Architecture (MIA),
    3–10

N

Name
  of constraint, 3–48e
    in multischema database, 5–11
  of database element, 3–22
  of domain, 3–28
    in multischema database, 5–11
  of element
    in multischema database, 5–5, 5–6
  of schema
    in multischema database, 5–3, 5–6
  SQL
    with multischema, 5–7
  stored, 5–7
  user-supplied, 3–22
Naming conventions
  for repository, 10–2
NATURAL JOIN, 3–81
NCHAR data type, 3–31, 3–42
NCHAR VARYING data type, 3–31, 3–42
NO DATA option
  on EXPORT/IMPORT statements, 7–84
Nodes, cluster
  modifying number of, 7–19, 7–29
NODE SIZE clause
  of CREATE INDEX statement, 4–34, 4–35,
    7–59
Non-key
  finding, 2–2
Nonranked sorted index
  See Sorted index, nonranked
Normalization
  first normal form, 2–9
  overview of, 2–8
  second normal form, 2–9

Normalization (cont'd)
third normal form, 2–9
Not deferrable constraint, 3–46
NOT NULL constraint, 3–46
Notrigger_Relations qualifier
RMU Load command, 6–34
NOT UPDATABLE keyword
ALTER STORAGE MAP statement, 7–66
CREATE STORAGE MAP statement, 4–12
NULL constraint, 3–51
Null value
loading from flat files, 6–40
unloading from flat files, 6–42
NUMBER OF BUFFERS clause
modifying, 7–33
NUMBER OF CLUSTER NODES clause
modifying, 7–19, 7–29
for single-file databases, 7–19
Number of nodes
modifying, 7–19
NUMBER OF RECOVERY BUFFERS clause
modifying, 7–34
NUMBER OF USERS clause
modifying, 7–19, 7–28
for single-file databases, 7–19

O

Object
in database security, 9–2
Offline modification
of database, 7–10t
On-disk structure
components, 1–6
multifile, 1–8, 1–9f
One-to-many relationship, 2–6
One-to-one relationship, 2–6
Online modification
of database, 7–10t, 7–13t
Opening database, 3–21
OPEN IS clause, 3–21
OPEN statement
privilege required, 9–10
OpenVMS privilege, 9–34
OPER privilege, 9–34
Optical disk
See WORM optical device
Oracle CDD/Repository
See Repository
Oracle Expert software, 1–6
Oracle Trace software, 1–6
OTHERWISE clause
in index definition, 4–9
in storage map definition, 4–8
omitting, 4–9, 7–62, 7–70
Outline
creating
privilege required, 9–12
dropping
privilege required, 9–12
Overflow partition, 7–6
in index definition, 4–9
in storage map definition, 4–8
omitting, 4–9, 7–62, 7–70
Overhead
page, 4–41

P

Packed decimal data type
restriction, 6–39
Page divisions of storage area
space area management page, 4–14, 4–16
Page format, 4–2, 4–13
mixed, 4–14, 4–21, 4–25, 4–30
uniform, 4–13, 4–20
PAGE FORMAT clause
modifying for storage area, 7–40
Page-level locking
specifying, 7–32
Page overhead, 4–41
Page size, 4–56
calculating
considerations, 4–41
example, 4–51
estimating
considerations, 6–15
modifying, 7–40, 7–49

PAGE SIZE clause, 3–20
  importance for hashed index, 4–21
  modifying for storage area, 7–40
Parallel load, 6–1, 6–52 to 6–60
  guidelines, 6–54
  performance, 6–17, 6–55
  sorting data, 6–2
  troubleshooting, 6–17
Parallel storage area creation, 3–15
Partitioning, 7–61
  enforcing, 4–11
  horizontal, 1–8, 4–4, 4–7, 4–8e, 7–61
  parallel
    of multiple tables and indexes, 4–26e
  strict, 4–11
  vertical, 1–8, 4–4, 4–10, 4–28e
    modifying, 7–66
PARTITIONING IS (NOT) UPDATABLE clause
  ALTER STORAGE MAP statement, 4–12, 7–66
  CREATE STORAGE MAP statement, 4–11
PATHNAME clause
  invoking by, 3–5
  invoking database by, 3–5
PERCENT FILL clause
  of CREATE INDEX statement, 4–34, 4–38, 7–59
Performance
  affected by constraint, 3–51
  degradation
    caused by index, 3–66, 3–68
  improving
    during load, 6–1
    for functions, 3–67
    for update operation, 3–67
    in searching large table, 3–66
    range retrieval, 6–1
    specifying appropriate page format, 4–13
    storage design to support exact match retrieval, 4–21
    storage design to support range retrieval, 4–19
    with hashed index, 3–63
    with physical design, 4–1
    with sorted index, 3–61
Performance (cont'd)
  index structures, 4–34
  monitoring, 4–3
  optimizing index structures, 4–34
  parallel load, 6–55
  run-length compression, 3–70
  using deferred snapshot file, 7–37
  using index, 3–66
  using list, 4–30, 4–32
  using RMU Load, 6–36
Performance monitor, 6–18
personnel database
  creating, 1–10
Physical database design, 1–5
  default, 3–1
  making a prototype, 4–1
  using Oracle Expert, 1–6
  using Oracle Trace, 1–6
Physical definition
  of database, 3–2
Pieces tracking
  defined, 1–2
  definition of, 10–1
PLACEMENT ONLY RETURNING DBKEY clause
  using to load data, 6–5
PLACEMENT VIA INDEX clause
  ALTER STORAGE MAP statement, 7–68, 7–69
  for clustering rows from different tables, 4–24
  specifying hashed index, 4–21
Plan file
  definition, 6–56
  generating, 6–58
  using in load, 6–59
Precompiled SQL
  loading data with, 6–28
Preventing database creation, 9–46
Primary key
  compound, 2–3
  finding, 2–2
  multisegmented, 2–3
  referential integrity and, 3–54

PRIMARY KEY constraint, 3–46
Privilege
See also Access control list (ACL); Access privilege set; Protection
ACL-style, 9–3
advantages of, 9–8
ANSI/ISO-style, 9–3
advantages of, 9–8
building access privilege set, 9–14
building ACL, 9–14
database, 9–8
DBADM, 9–33
DBCTRL, 9–14
default access to database, 9–30
defining
using SQL, 9–19
denying, 9–19, 9–20
to a group of users, 9–20e
differences between ANSI/ISO and ACL style, 9–2, 9–6
Digital UNIX root account, 9–34
displaying, 9–14, 9–35
granting, 9–19
implicit, 9–33
modifying
effect on user, 9–20
OpenVMS OPER, 9–34
override, 9–9, 9–33
required for data definition, 9–9t
required for data manipulation, 9–9t
required for using Oracle RMU commands, 9–38t
required for using RMU commands, 9–35, 9–43t
SECURITY, 9–33
SELECT, 9–19
setting, 9–36
table or view, 9–8
temporary table and, 3–74
to attach to database, 9–19
to create ACLs, 9–5, 9–14
to create database, 9–46
Privilege mask, 9–7
Procedure
creating
privilege required, 9–12
dropping
privilege required, 9–12
external, 3–4
Production database
physical design of, 1–6
Program
embedding data definition statement in, 3–5
Protection
See also Access control list (ACL); Privilege
column level, 9–5, 9–24e
database level, 9–5
default, 9–30
defining, 3–9
for column, 9–24
for repository, 9–47
for table, 9–22
for view, 9–25, 9–27, 9–29
defining with RMU privilege, 9–35
defining with RMU privileges, 9–35
repository, 10–13
table level, 9–5
using view for, 3–83

Q

Qualified name, 5–5
using, 5–6
QUERY HEADER clause, 3–34
for column, 3–39
in view definition, 3–84
in domain definition, 3–34
QUERY NAME clause
for column, 3–39
Quota
file open, 3–15
Quotation mark (")
as delimiting identifier, 5–9

R

Range retrieval
  index use, 3–61
RANKED keyword
  CREATE INDEX, 3–61
Ranked sorted index
  See Sorted index, ranked
.rbr file, 7–84
  See Interchange (.rbr) file
.rda file
  See Storage area (.rda) file
RDB$CATALOG, 5–3
  dropping, 8–24
RDB$DBHANDLE
  alias, 7–87
RDB$SCHEMA, 5–3
  dropping, 8–23
RDB$SYSTEM storage area, 1–7, 3–8, 3–14, 4–1, 4–13, 7–28, 7–41
  modifying parameter, 7–47
  page format, 4–13
  table with data, 7–72
.rdb file
  See Database root (.rdb) file, Database directory (.rdb) file
Rdb Management Utility
  See RMU
RDBVMS$CREATE_DB logical name, 9–46
RDBVMS$CREATE_DB rights identifier, 9–47
RDB_BIND_BUFFERS configuration parameter, 6–14
  loading data and, 6–3
RDB_BIND_LOCK_TIMEOUT configuration parameter, 7–31
RDB_BIND_SEGMENTED_STRING_BUFFER
  configuration parameter, 4–30
rdb_system.rdb file, 3–7
RDM$BIND_BUFFERS logical name
  for load operation, 6–14
  loading data and, 6–3
RDM$BIND_LOCK_TIMEOUT logical name, 7–31
RDMS$BIND_SEGMENTED_STRING_BUFFER logical name, 4–30
READALL privilege, 9–34
Read-only storage area, 4–17, 7–54
Read-only transaction
  for snapshot, 3–17
Read/write storage area, 4–17
Record definition (.rrd) file
  format, 6–37
  restriction, 6–39, 6–45
Record-level locking
  See Row-level locking
Record Management Services file
  See Flat file
Record statistics
  load operation and, 6–18
Recovery buffer
  modifying, 7–34
Recovery-unit journal (.ruj) file, 1–8, 7–1
Reducing I/O
  with clustering, 7–61
  with FAST COMMIT clause, 7–26
  with global buffers, 7–33
  with hashed index, 7–60
  with partitioning, 7–61
  with sorted index, 7–59, 7–60
Reducing lock conflict, 7–36, 7–59
Reducing locking
  during load, 6–2
Reference monitor concept
  in database security, 9–1
REFERENCES clause
  CREATE TABLE statement, 3–46
Referential integrity, 3–53, 3–54
  deleting, 8–19
  maintaining, 3–51
  trigger and, 3–54
Reflexive relationship, 2–7
Relational database
  theory, 1–1

Relationship
among transactions, 2–10
many-to-many, 2–7
one-to-many, 2–6
one-to-one, 2–7
reflexive, 2–7
revising, 2–6, 2–7
types of, 2–3
Renaming column, 8–16
Renaming table, 8–16
REORGANIZE clause
ALTER STORAGE MAP statement, 7–2, 7–66, 7–70
Reorganizing database, 7–75 to 7–82
database files, 7–79
IMPORT statement
definition sequence, 7–77
multifile, 7–79
single-file, 7–76, 7–79
storage areas
with storage map REORGANIZE clause, 7–66
Repository
anchor, 10–2
assigning directory names, 10–24e
copying data definitions from, 6–24
creating constraints, 10–14
creating data definitions, 10–22
using CDO, 10–7e
using SQL, 10–28e
creating new version of definition, 10–17
creating shareable definitions, 9–5, 10–6
criteria for using, 10–4
definition
creating, 10–22, 10–28
deleting, 10–36, 10–40
definition of, 10–1
deleting definitions, 10–36
dropping link with database, 10–36
ENTER command, 6–24, 10–24e
field definitions, 10–1
loading data
using repository definition, 6–24
modifying CDD$FILE name, 10–43
modifying definitions, 10–19, 10–21
Repository (cont'd)
modifying file name of database, 10–43
naming conventions of, 10–2
path name, 10–2
pieces tracking commands, 10–3t
replacing existing definition, 10–19
requiring for update, 7–18
restructuring database without, 7–40
specifying in database definition, 3–5
tracking definitions, 10–1, 10–3
trade-offs of using, 3–5, 10–5
updating using SQL, 10–23, 10–24e
updating with database file, 10–22t
using to create table, 3–36
version number, 10–2
Repository element
definition of, 10–1
REPOSITORY OPERATOR command, 10–2, 10–3
Requirements analysis
in logical design, 2–1
RESERVE JOURNALS clause, 3–7, 3–12, 3–13, 7–25
RESERVE STORAGE AREAS clause, 3–7, 3–14, 7–42
RESTRICTED ACCESS clause
of IMPORT statement, 7–2
Restricting access
during import operation, 7–2
to subset of rows, 9–27
to table, 9–25, 9–27, 9–29
Restricting database creation, 9–46
RESTRICT keyword, 8–4
DROP STORAGE AREA statement, 7–54
DROP TABLE statement, 8–4
Restructuring database, 7–1 to 7–87
using ALTER DATABASE statement, 7–2
using IMPORT and EXPORT statements, 7–2
using RMU Load, 7–2
without repository, 7–40
Retrieval methods
See Retrieving data
Retrieving data
access methods, 1–9

Retrieving data (cont'd)
index use when specifying a range of values, 3–61
optimizing performance of specific queries through storage design, 4–19
REVOKE statement, 9–2, 9–19
alias, 9–19
effect on transaction share mode, 9–20
effect on user, 9–20
privilege required, 9–10
Rights identifiers, 9–6
RMS file
See Flat file
RMU
privilege required, 9–35
RMU Analyze command, 4–3
RMU Backup command
for moving database, 7–86
use before database restructuring, 7–3
RMU Close command, 3–21
RMU Copy_Database command, 7–9, 7–82
RMU Dump command
Header qualifier, 7–38
Users qualifier, 7–36
RMU Extract command
using to load data, 6–47, 6–49
using to unload data, 6–47, 6–48e
RMU Load command, 6–1, 6–32 to 6–60, 7–2
Commit_Every qualifier, 6–34
Constraints qualifier, 6–2, 6–15, 6–35
Defer_Index_Updates qualifier, 6–2, 6–55
Fields qualifier, 6–48
List_Plan qualifier, 6–58
loading database, 6–32, 6–45, 6–46, 6–51
multiprocess
See also Parallel load, 6–52
Noconstraints qualifier, 6–2, 6–15
Notrigger_Relations qualifier, 6–2, 6–34
null value, 6–40
parallel load, 6–52
See also Parallel load
performance, 6–3, 6–34, 6–36
Place qualifier, 6–2, 6–34
plan file, 6–58, 6–59
Record_Definition qualifier, 6–33, 6–38
RMU Load command (cont'd)
Row_Count qualifier, 6–35
Statistics qualifier, 6–17, 6–36
troubleshooting, 6–16
RMU Load Plan command
using plan file, 6–59
RMU Move_Area command, 7–9, 7–48, 7–50, 7–85
Area qualifier, 7–49
Blocks_per_page qualifier, 7–50
Directory qualifier, 7–49
Online qualifier, 7–49
Snapshots qualifier, 7–49
Thresholds Qualifier, 7–50
RMU Open command, 3–21
RMU Restore command, 7–9
for moving database, 7–86
RMU Set Privilege command, 9–36
RMU Show privilege command, 9–35
RMU Show Statistics command
See Performance Monitor
RMU Unload command, 6–32 to 6–60, 7–2
null value, 6–42
Record_Definition qualifier, 6–38
Role-oriented access
database privilege, 9–9
restricting, 9–29
ROLLBACK statement
with CREATE DATABASE statement, 3–10
Roll forward
using after-image journal file, 7–23, 7–25
root account
privilege and, 9–34
Root file
See Database root (.rdb) file
Routine
external, 3–4
invoking from trigger, 3–58
ROW CACHE IS clause
of ALTER DATABASE statement, 3–19
Row-level cache, 3–19
Row-level locking, 7–32
Row_Count option
RMU Load Parallel command, 6–35

.rrd file
See Record definition (.rrd) file
.ruj file
See Recovery-unit journal (.ruj) file
Run-length compression, 3–70, 3–71
benefits, 3–70
Run-time library routine
invoking from trigger, 3–58

S

Sample database
creating, 1–10
online command procedure to define, 1–10
Schema, 3–4
creating, 5–3
privilege required, 9–12
definition of, 5–1
deleting, 8–22
privilege required, 9–12
Schema element
creating in multischema database, 5–10
Scratch table
creating, 3–73
Script (SQL)
advantages of using, 3–5
Search key
hashed, 1–10
Second normal form, 2–9
Security
See also Access control list (ACL); Privilege; Protection
audit journal, 9–2
database, 9–35
level, 9–30
object, 9–2
subject, 9–1
overview, 9–1
reference monitor concept, 9–1
SECURITY privilege, 9–9, 9–33, 9–34
Segmented string data type
See LIST OF BYTE VARYING data type
Select expression
optimizing combining data in multiple tables, 4–24
SELECT privilege
needed to attach to database, 9–19
SELECT statement
privilege required, 9–9
Semicolon (;)
in CREATE DATABASE statement, 3–9
Sequential retrieval, 1–9
SESSION_USER function, 3–32
SET CATALOG statement, 5–3
SET CHARACTER LENGTH statement, 3–25
CHARACTERS option, 3–25
OCTETS option, 3–25
restriction, 3–26
SET DIALECT statement, 3–11e
SET QUOTING RULES statement, 5–9
SET TRANSACTION statement
privilege required for distributed transaction, 9–10
SHARED DATA DEFINITION clause, 3–69
Shadowing, 4–61
SHARED DATA DEFINITION clause, 3–69
Shared transaction type
load operation and, 6–36
Share mode
effect of GRANT statement, 9–20
effect of REVOKE statement, 9–20
for restructuring database, 7–10
SHOW DOMAIN statement, 8–1
SHOW PROTECTION statement, 9–2, 9–14, 9–19, 9–31
SHOW statement
system-defined metadata, 1–2
user-defined metadata, 1–2
Single-file database, 1–6
modifying, 7–19
SIZE IS segment truncation, 3–70
benefits, 3–71
SMALLINT data type, 3–23
Snapshot (.snp) file, 1–7, 3–8, 3–17
allocation size
modifying, 7–39
deferred, 7–37

Snapshot (.snp) file (cont'd)
determining current size, 7–35
DISABLED option, 3–17
disabling, 6–13, 7–34, 7–36
effects on performance, 7–36
ENABLED DEFERRED option, 3–17
ENABLED IMMEDIATE option, 3–17
enabling, 7–34
extent
modifying, 7–38
for storage area, 4–1
immediate, 7–38
loading data and, 6–3
modifying size of, 7–38
transaction sequence number, 7–37
using, 3–17 to 3–19
SNAPSHOT ALLOCATION IS clause
modifying, 7–38, 7–39
of ALTER DATABASE statement, 7–34
SNAPSHOT EXTENT IS clause
modifying, 7–38
SNAPSHOT FILENAME clause
modifying for storage area, 7–40
SNAPSHOT IS clause
DEFERRED, 7–37
IMMEDIATE, 7–38
SNAPSHOT IS ENABLED clause
of ALTER DATABASE statement, 7–34
.snp file
See Snapshot (.snp) file
Sorted index, 1–10, 3–60, 3–61
See also Index
calculating node size
nonranked, 4–36
ranked, 4–35
compressing, 3–70
duplicate, 3–62
duplicate compression, 3–62
fullness percentage
NODE SIZE and, 4–38
load operation and, 6–15
modifying, 7–59
NODE SIZE, 4–34
fullness percentage and, 4–38
nonranked
Sorted index
nonranked (cont'd)
calculating node size, 4–36
overflow partition, 7–62
page format required, 4–19
PERCENT FILL, 4–34
ranked
calculating node size, 4–35
retrieval with data values, 1–9
retrieval with dbkey, 1–9
setting characteristics, 4–34, 4–38
USAGE QUERY, 4–34
USAGE UPDATE, 4–34
Sorting rows
collating sequence, 3–33
Space, for file
extending, 7–27, 7–41
Space area management (SPAM) page, 4–14, 4–16, 4–54
estimating number of, 4–54
SPAM page
See Space area management (SPAM) page
SQL editor
to create database, 3–5
SQL interface, 1–2
SQL module language
loading data with, 6–20
SQL statement
ALTER DATABASE, 7–2, 7–3
ADJUSTABLE LOCK GRANULARITY clause, 7–29
ALTER STORAGE AREA clause, 7–54
CARRY OVER LOCKS clause, 7–31
creating journal file, 7–25
DICTIONARY IS REQUIRED clause, 7–40
DROP STORAGE AREA clause, 7–54
enabling journaling, 7–23
EXTENT IS extent-pages PAGES clause, 7–27
GLOBAL BUFFERS clause, 7–33
LOCKING IS PAGE LEVEL clause, 7–32
LOCKING IS ROW LEVEL clause, 7–32
LOCK TIMEOUT clause, 7–31
modifying database characteristics, 7–19

SQL statement
ALTER DATABASE (cont'd)
NUMBER OF BUFFERS IS clause, 7–33
NUMBER OF CLUSTER NODES IS clause, 7–19, 7–29
NUMBER OF RECOVERY BUFFERS IS clause, 7–34
NUMBER OF USERS IS clause, 7–19, 7–28
privilege required, 9–10, 9–13
SNAPSHOT clause, 7–34
SNAPSHOT IS ENABLED clause
DEFERRED, 7–37
IMMEDIATE, 7–38
ALTER DOMAIN, 8–1, 8–3
privilege required, 9–12
ALTER INDEX, 7–44
privilege required, 9–12
ALTER STORAGE AREA clause
EXTENT clause, 7–41
ALTER STORAGE MAP, 7–9, 7–44, 7–65
privilege required, 9–13
ALTER TABLE, 8–4, 8–17
privilege required, 9–11, 9–12, 9–13
ATTACH
privilege required, 9–9
COMMENT ON, 3–26
CREATE CATALOG, 5–3
privilege required, 9–11
CREATE COLLATING SEQUENCE
privilege required, 9–11
CREATE DATABASE, 3–6, 4–28
multischema option, 5–2
OPEN IS clause, 3–21
CREATE DOMAIN, 3–27
in multischema database, 5–10
privilege required, 9–12
CREATE FUNCTION, 3–58
privilege required, 9–12
CREATE INDEX, 3–60, 4–6, 4–7, 4–8
hashed, 3–65e
privilege required, 9–12
sorted, 3–62e, 3–63e
CREATE MODULE, 3–3
privilege required, 9–12
CREATE OUTLINE

SQL statement
CREATE OUTLINE (cont'd)
privilege required, 9–12
CREATE PROCEDURE
privilege required, 9–12
Create Routine, 3–4
privilege required, 9–12
CREATE SCHEMA, 5–3
privilege required, 9–12
CREATE STORAGE AREA, 4–1
CREATE STORAGE MAP, 4–6, 4–7, 4–8
PLACEMENT VIA INDEX clause, 7–80
privilege required, 9–13
CREATE TABLE, 3–37, 3–38
in multischema database, 5–11
privilege required, 9–13
CREATE TEMPORARY TABLE, 3–72
CREATE TRIGGER, 3–55
in multischema database, 5–14
privilege required, 9–13
CREATE VIEW, 3–83
in multischema database, 5–13
privilege required, 9–13
DECLARE TRANSACTION
privilege required, 9–10
DELETE
privilege required, 9–10
DROP CATALOG, 8–23
privilege required, 9–11
DROP COLLATING SEQUENCE
privilege required, 9–11
DROP CONSTRAINT, 8–4
privilege required, 9–12
DROP DATABASE, 7–87
privilege required, 9–10
DROP DOMAIN, 8–3
privilege required, 9–12
DROP FUNCTION
privilege required, 9–12
DROP INDEX, 7–64
privilege required, 9–12
DROP MODULE
privilege required, 9–12
DROP OUTLINE
privilege required, 9–12
DROP PROCEDURE

SQL statement
DROP PROCEDURE (cont'd)
privilege required, 9–12
DROP SCHEMA, 8–22
privilege required, 9–12
DROP STORAGE MAP, 7–75
privilege required, 9–13
DROP TABLE, 8–4, 8–17
privilege required, 9–13
DROP VIEW, 8–21
privilege required, 9–13
EXPORT, 7–2, 7–75, 7–76, 7–79, 7–84
privilege required, 9–13
GRANT, 9–2, 9–6, 9–19, 9–20
privilege required, 9–10
IMPORT, 7–2, 7–75, 7–76, 7–79, 7–84
INSERT
privilege required, 9–9
INTEGRATE, 10–21, 10–22, 10–23, 10–28
privilege required, 9–13
OPEN
privilege required, 9–10
privilege required for, 9–9t
REVOKE, 9–2, 9–19, 9–20
privilege required, 9–10
SELECT
privilege required, 9–9
SET DIALECT, 3–11e
SET QUOTING RULES, 5–9
SET TRANSACTION
privilege required, 9–10
SHOW PROTECTION, 9–14, 9–31
TRUNCATE TABLE, 8–6
privilege required, 9–13
UPDATE
privilege required, 9–9
Statistics
during load operation, 6–36
Statistics qualifier
RMU Load command, 6–17
Storage area, 4–11
adding, 7–42, 7–44, 7–70
asynchronous creation of, 3–15
calculating
file allocation size, 4–52, 6–15
Storage area
calculating (cont'd)
overhead, 4–42
page size, 4–41, 6–15
size for mixed page format, 4–40
clustering parent and child rows, 4–62
clustering rows from different tables, 4–15
creating, 3–14, 7–80
asynchronously, 3–15
in parallel, 3–15
multiple, 4–1
privilege required, 9–13
default, 3–8, 3–15
deleting, 7–54
privilege required, 9–13
disabling automatic expansion, 7–26, 7–38
distributing tables and indexes among, 4–3
estimating file allocation size, 6–14
extent value, 7–27, 7–41
list data type and, 4–16, 4–30, 4–32
mixed page format
estimating number of SPAM pages, 4–54
modifying, 7–44e, 7–47e
for loading data, 6–14
from write-once to read/write, 7–52
privilege required, 9–13
moving, 7–9, 7–48
overflow partition, 4–8
parallel creation of, 3–15
PLACEMENT VIA INDEX clause, 4–59, 4–60, 4–61
RDB$SYSTEM, 3–8, 4–1
read/write, 4–17
read-only, 4–17, 7–54
reorganizing database
definition sequence, 7–77
reserving slots for, 3–7, 3–14, 7–19, 7–42
snapshot file and, 3–18
space area management (SPAM) page in, 4–14, 4–16
specifying, 3–14
specifying on-disk structure for
hashed index, 4–15
manipulating SPAM pages, 4–14, 4–16
mixed page format, 4–14

Storage area
  specifying on-disk structure for (cont'd)
    sorted indexes, 4–19
    uniform page format, 4–13
  using more than one for
    an index, 4–8e
    a table, 4–4, 4–8e, 4–10, 4–28e
  without PLACEMENT VIA INDEX clause, 4–58
  write-once, 4–17, 4–33
    creating, 7–52
    journaling, 4–34
    moving data from, 7–52
Storage area (.rda) file, 1–7, 3–7
Storage map
  adding, 7–2, 7–47
  adding partition, 4–9, 7–70
  creating, 3–14, 4–7, 7–80
    guidelines, 4–6, 4–8
    privilege required, 9–13
  deleting, 7–75
    privilege required, 9–13
  effect of changes on row placement, 7–66
  enforcing partitioning, 4–11
  list data type and, 4–30
    storing randomly, 4–32
    storing sequentially, 4–32
  load operation and, 6–16
  modifying, 7–2, 7–9, 7–65 to 7–75
    PARTITIONING IS clause, 4–12
    privilege required, 9–13
  options, 4–10
  REORGANIZE clause, 7–66
  reorganizing database
    definition sequence, 7–77
  strict partitioning, 4–11
  table with data and, 7–72
  when indexes change, 7–62
Storage method, 1–8
STORE clause
  in CREATE INDEX statement, 4–1, 4–7
  in CREATE STORAGE MAP statement, 4–1
Stored function, 3–3
  executing
    privilege required, 9–9
Stored name, 5–7
Stored procedure, 3–3
  creating
    privilege required, 9–12
  dropping
    privilege required, 9–12
  executing
    privilege required, 9–9
Stored routine, 3–3
Storing list, 4–30
  on WORM optical device, 4–33
  randomly across storage areas, 4–32
  sequentially across storage areas, 4–32
Subject
  in database security, 9–1
SYSPRV privilege, 9–34
System-defined identifier, 9–7
System index
  compressing, 3–16, 7–19
SYSTEM INDEX COMPRESSION clause, 3–16
System record, 4–41
System relation
  See System table
System table, 3–8
  examples of, 1–2
  moving, 7–73
SYSTEM_USER function, 3–32

T

Table, 1–1f, 3–1
  creating, 3–36, 3–38, 3–40e, 3–48e
    from repository, 3–36
    in multischema database, 5–11
    privilege required, 9–13
    using character set, 3–42
  defined, 1–1, 3–1
  deleting, 8–4, 8–5e
    in multischema database, 8–17
    privilege required, 9–13
    quickly, 8–6
  dependent data identifying, 2–8
  elements of, 3–38
  integrating with repository, 10–21, 10–30, 10–35

Table (cont'd)
modifying, 8–4, 8–6e, 8–7, 8–8e, 8–10e
column with date-time data type, 8–12
in multischema database, 8–17
privilege required, 9–11, 9–13
on-disk storage of, 1–5
partitioning
across multiple storage areas, 4–3, 4–4, 4–8e, 4–10, 4–26, 4–28e
vertical, 7–66
protection for, 9–22
removing redundancy from, 2–8
specifying storage area for, 4–7
system
moving, 7–73
vertical partitioning, 4–28e
virtual
See View
Table constraint, 3–45
See also Constraint
Table row size
calculating, 4–49
Temporary table
creating, 3–72
data type in, 3–74
declared local, 3–79
deleting, 3–74
global, 3–75
local, 3–75
modifying, 3–74
restriction, 3–74
truncating, 3–74
types of, 3–73
virtual memory
estimating, 3–82
Third normal form, 2–9
Threshold, 4–10, 4–16
for logical area, 4–14, 7–71
for storage area, 4–14
modifying
for storage area, 7–40
values, 7–40, 7–49, 7–66, 7–71, 7–83
THRESHOLDS clause
modifying for storage area, 7–40
of CREATE STORAGE MAP statement, 4–10
TIME data type, 3–24
Timestamp
using to record actions, 3–54
TIMESTAMP data type, 3–24
modifying, 8–12
Tools
logical design, 2–1
TRACE clause, 7–85
Tracking changes to a database
using trigger, 3–54
Transaction
activity rates of, 2–10
analysis, 2–9
archiving, 2–10, 2–14
database, 1–4
labeling, 2–10
map, 2–11f
mapping, 2–10
paths, 2–10
prototype, 2–13
types of, 2–10
Transaction sequence number (TSN)
index, 4–41
snapshot file, 7–37
Transaction type
load operation and, 6–36
Trigger, 3–53
creating, 3–54
in multischema database, 5–14
privilege required, 9–13
deleting, 8–19
external function and, 3–58
index and, 3–58
loading data and, 6–2, 6–15, 6–34
modifying, 8–19
restriction, 3–56, 3–57, 3–58
TRUNCATE TABLE and, 8–6
update, 3–57
using to restrict access, 9–27, 9–28
using to track changes to a database, 3–54
TRUNCATE TABLE statement, 8–6
privilege required, 9–13
temporary table, 3–74

Truncation
of column, 8–10e
of snapshot file, 7–40

U

UIC
See User identifier
UID
See User identifier
Uniform page format, 4–13
UNIQUE constraint, 3–46, 3–51
UNIQUE keyword, 3–63, 3–65
.unl file, 6–33
Unloading data, 6–32, 6–45, 6–46
null value, 6–42
to flat file
using RMU Unload command, 6–32
restriction, 6–39
UPDATABLE keyword
ALTER STORAGE MAP statement, 7–66
CREATE STORAGE MAP statement, 4–12
UPDATE statement
performance with index, 3–66
privilege required, 9–9
Updating data
through view, 3–84
Updating database file
using the repository, 10–31
Updating repository
using database file, 10–22
using SQL, 10–23, 10–24e
USAGE clause
of ALTER INDEX statement, 7–60
of CREATE INDEX statement, 4–34, 4–38
User access sets
See Access control entry (ACE)
USER function, 3–32
trigger and, 3–55
User identification code (UIC)
See User identifier
User identifier, 9–5, 9–6
Users
modifying maximum number of, 7–19, 7–28
User-supplied name, 3–22

V

VARCHAR data type, 3–23, 3–30e
Variable page overhead, 4–41, 4–42
Varying-length character data types
unloading, 6–40
Vertical partitioning, 1–8, 4–4, 4–10, 4–28e
modifying, 7–66
View
CHECK OPTION clause, 3–84
creating, 3–83, 3–85 to 3–91
for security reasons, 3–83
guidelines, 3–83
privilege required, 9–13
to support reports, 3–83
CURRENT_INFO view, 3–89e
CURRENT_JOB view, 3–85e
CURRENT_SALARY view, 3–87e
data update using, 3–84
defining protection for, 9–25, 9–27, 9–29
deleting, 8–21
from repository, 8–22
in multischema database, 8–22
privilege required, 9–13
to modify table, 8–6e
index and, 3–84
modifying, 8–21
read-only, 3–84
restrict access by using, 9–27, 9–29
using to calculate date, 3–90e
using to restrict access, 9–28
Virtual column
in view definition, 3–83
Virtual memory
temporary table and, 3–82
Virtual table
See View
VMScluster
See Cluster nodes
Volume table
developing, 2–14

W

WAIT clause
  to close database, 3–21
WHEN clause
  CREATE TRIGGER statement, 3–56
Working set parameter
  load operation and, 6–36, 6–55
WORM optical device, 4–17
  creating storage area on, 4–17
  for storing lists, 4–33
  journaling, 4–34
  moving data from, 7–52
  moving data to, 7–50
Write-once, read-many optical device
  See WORM optical device
Write-once storage area, 4–17, 4–33
  creating, 7–52
  journaling, 4–34
  moving data from, 7–52
  moving data to, 7–50