Oracle® Database

VLDB and Partitioning Guide

23ai
F46740-09
October 2024

Oracle Database VLDB and Partitioning Guide, 23ai

F46740-09

Copyright © 2008, 2024, Oracle and/or its affiliates.

Primary Author: Eric Belden

Contributors: Penny Avril, Hermann Baer, Yasin Baskan, Gregg Christman, Jean-Pierre Dijcks, Sandeep Doraiswamy, Amit Ganesh, Lilian Hobbs, Kevin Jernigan, Dominique Jeunot, Hariharan Lakshmanan, Paul Lane, Sue K. Lee, Diana Lorentz, Vineet Marwah, Valarie Moore, Sujatha Muthulingam, Ajit Mylavarapu, Tony Morales, Ananth Raghavan, Venkatesh Radhakrishnan, Vivekanandhan Raja, Andy Rivenes, Chandrajith Unnithan, Mark Van de Wiel, Frederick Kush

This software and related documentation are provided under a license agreement containing restrictions on use and disclosure and are protected by intellectual property laws. Except as expressly permitted in your license agreement or allowed by law, you may not use, copy, reproduce, translate, broadcast, modify, license, transmit, distribute, exhibit, perform, publish, or display any part, in any form, or by any means. Reverse engineering, disassembly, or decompilation of this software, unless required by law for interoperability, is prohibited.

The information contained herein is subject to change without notice and is not warranted to be error-free. If you find any errors, please report them to us in writing.

If this is software, software documentation, data (as defined in the Federal Acquisition Regulation), or related documentation that is delivered to the U.S. Government or anyone licensing it on behalf of the U.S. Government, then the following notice is applicable:

U.S. GOVERNMENT END USERS: Oracle programs (including any operating system, integrated software, any programs embedded, installed, or activated on delivered hardware, and modifications of such programs) and Oracle computer documentation or other Oracle data delivered to or accessed by U.S. Government end users are "commercial computer software," "commercial computer software documentation," or "limited rights data" pursuant to the applicable Federal Acquisition Regulation and agency-specific supplemental regulations. As such, the use, reproduction, duplication, release, display, disclosure, modification, preparation of derivative works, and/or adaptation of i) Oracle programs (including any operating system, integrated software, any programs embedded, installed, or activated on delivered hardware, and modifications of such programs), ii) Oracle computer documentation and/or iii) other Oracle data, is subject to the rights and limitations specified in the license contained in the applicable contract. The terms governing the U.S. Government's use of Oracle cloud services are defined by the applicable contract for such services. No other rights are granted to the U.S. Government.

This software or hardware is developed for general use in a variety of information management applications. It is not developed or intended for use in any inherently dangerous applications, including applications that may create a risk of personal injury. If you use this software or hardware in dangerous applications, then you shall be responsible to take all appropriate fail-safe, backup, redundancy, and other measures to ensure its safe use. Oracle Corporation and its affiliates disclaim any liability for any damages caused by use of this software or hardware in dangerous applications.

Oracle®, Java, MySQL, and NetSuite are registered trademarks of Oracle and/or its affiliates. Other names may be trademarks of their respective owners.

Intel and Intel Inside are trademarks or registered trademarks of Intel Corporation. All SPARC trademarks are used under license and are trademarks or registered trademarks of SPARC International, Inc. AMD, Epyc, and the AMD logo are trademarks or registered trademarks of Advanced Micro Devices. UNIX is a registered trademark of The Open Group.

This software or hardware and documentation may provide access to or information about content, products, and services from third parties. Oracle Corporation and its affiliates are not responsible for and expressly disclaim all warranties of any kind with respect to third-party content, products, and services unless otherwise set forth in an applicable agreement between you and Oracle. Oracle Corporation and its affiliates will not be responsible for any loss, costs, or damages incurred due to your access to or use of third-party content, products, or services, except as set forth in an applicable agreement between you and Oracle.

Contents

Preface

Audience xvii
Documentation Accessibility xvii
Related Documents xvii
Conventions xviii

1 Introduction to Very Large Databases

1.1 Introduction to Partitioning 1-1
1.2 VLDB and Partitioning 1-2
1.3 Partitioning As the Foundation for Information Lifecycle Management 1-3
1.4 Partitioning for All Databases 1-3

2 Partitioning Concepts

2.1 Partitioning Overview 2-1
2.1.1 Basics of Partitioning 2-2
2.1.2 Partitioning Key 2-3
2.1.3 Partitioned Tables 2-3
2.1.3.1 When to Partition a Table 2-4
2.1.3.2 When to Partition an Index 2-4
2.1.4 Partitioned Index-Organized Tables 2-4
2.1.5 System Partitioning 2-5
2.1.6 Partitioning for Information Lifecycle Management 2-5
2.1.7 Range Partitioning for Hash Clusters 2-5
2.1.8 Partitioning and LOB Data 2-6
2.1.9 Partitioning on External Tables 2-6
2.1.10 Hybrid Partitioned Tables 2-6
2.1.11 Collections in XMLType and Object Data 2-9
2.2 Benefits of Partitioning 2-10
2.2.1 Partitioning for Performance 2-10
2.2.1.1 Partition Pruning for Performance 2-10
2.2.1.2 Partition-Wise Joins for Performance 2-11
2.2.2 Partitioning for Manageability 2-11

2.2.3 Partitioning for Availability 2-11
2.3 Partitioning Strategies 2-11
2.3.1 Single-Level Partitioning 2-12
2.3.1.1 Range Partitioning 2-13
2.3.1.2 Hash Partitioning 2-13
2.3.1.3 List Partitioning 2-13
2.3.2 Composite Partitioning 2-13
2.3.2.1 Composite Range-Range Partitioning 2-14
2.3.2.2 Composite Range-Hash Partitioning 2-14
2.3.2.3 Composite Range-List Partitioning 2-15
2.3.2.4 Composite List-Range Partitioning 2-15
2.3.2.5 Composite List-Hash Partitioning 2-15
2.3.2.6 Composite List-List Partitioning 2-15
2.3.2.7 Composite Hash-Hash Partitioning 2-15
2.3.2.8 Composite Hash-List Partitioning 2-15
2.3.2.9 Composite Hash-Range Partitioning 2-15
2.4 Partitioning Extensions 2-15
2.4.1 Manageability Extensions 2-16
2.4.1.1 Interval Partitioning 2-16
2.4.1.2 Partition Advisor 2-16
2.4.2 Partitioning Key Extensions 2-17
2.4.2.1 Reference Partitioning 2-17
2.4.2.2 Virtual Column-Based Partitioning 2-18
2.5 Indexing on Partitioned Tables 2-18
2.5.1 Deciding on the Type of Partitioned Index to Use 2-19
2.5.2 Local Partitioned Indexes 2-19
2.5.3 Global Partitioned Indexes 2-20
2.5.3.1 Global Range Partitioned Indexes 2-21
2.5.3.2 Global Hash Partitioned Indexes 2-21
2.5.3.3 Maintenance of Global Partitioned Indexes 2-21
2.5.4 Global Nonpartitioned Indexes 2-22
2.5.5 Miscellaneous Information about Creating Indexes on Partitioned Tables 2-23
2.5.6 Partial Indexes for Partitioned Tables 2-23
2.5.7 Partitioned Indexes on Composite Partitions 2-24

3 Partitioning for Availability, Manageability, and Performance

3.1 Partition Pruning 3-1
3.1.1 Benefits of Partition Pruning 3-1
3.1.2 Information That Can Be Used for Partition Pruning 3-2
3.1.3 How to Identify Whether Partition Pruning Has Been Used 3-3
3.1.4 Static Partition Pruning 3-3

3.1.5 Dynamic Partition Pruning 3-4
3.1.5.1 Dynamic Pruning with Bind Variables 3-4
3.1.5.2 Dynamic Pruning with Subqueries 3-5
3.1.5.3 Dynamic Pruning with Star Transformation 3-5
3.1.5.4 Dynamic Pruning with Nested Loop Joins 3-7
3.1.6 Partition Pruning with Zone Maps 3-7
3.1.7 Partition Pruning Tips 3-9
3.1.7.1 Data Type Conversions 3-9
3.1.7.2 Function Calls 3-12
3.1.7.3 Collection Tables 3-13
3.2 Partition-Wise Operations 3-14
3.2.1 Full Partition-Wise Joins 3-14
3.2.1.1 Querying a Full Partition-Wise Join 3-15
3.2.1.2 Full Partition-Wise Joins: Single-Level - Single-Level 3-15
3.2.1.3 Full Partition-Wise Joins: Composite - Single-Level 3-17
3.2.1.4 Full Partition-Wise Joins: Composite - Composite 3-19
3.2.2 Partial Partition-Wise Joins 3-19
3.2.2.1 Partial Partition-Wise Joins: Single-Level Partitioning 3-20
3.2.2.2 Partial Partition-Wise Joins: Composite 3-22
3.3 Index Partitioning 3-23
3.3.1 Local Partitioned Indexes 3-24
3.3.1.1 Local Prefixed Indexes 3-25
3.3.1.2 Local Nonprefixed Indexes 3-26
3.3.2 Global Partitioned Indexes 3-34
3.3.2.1 Prefixed and Nonprefixed Global Partitioned Indexes 3-34
3.3.2.2 Management of Global Partitioned Indexes 3-34
3.3.3 Summary of Partitioned Index Types 3-35
3.3.4 The Importance of Nonprefixed Indexes 3-36
3.3.5 Performance Implications of Prefixed and Nonprefixed Indexes 3-36
3.3.6 Advanced Index Compression With Partitioned Indexes 3-37
3.3.7 Guidelines for Partitioning Indexes 3-37
3.3.8 Physical Attributes of Index Partitions 3-38
3.4 Partitioning and Table Compression 3-39
3.4.1 Table Compression and Bitmap Indexes 3-40
3.4.2 Example of Table Compression and Partitioning 3-41
3.5 Recommendations for Choosing a Partitioning Strategy 3-42
3.5.1 When to Use Range or Interval Partitioning 3-42
3.5.2 When to Use Hash Partitioning 3-43
3.5.3 When to Use List Partitioning 3-45
3.5.4 When to Use Composite Partitioning 3-45
3.5.4.1 When to Use Composite Range-Hash Partitioning 3-46
3.5.4.2 When to Use Composite Range-List Partitioning 3-47

3.5.4.3 When to Use Composite Range-Range Partitioning 3-48
3.5.4.4 When to Use Composite List-Hash Partitioning 3-48
3.5.4.5 When to Use Composite List-List Partitioning 3-49
3.5.4.6 When to Use Composite List-Range Partitioning 3-50
3.5.5 When to Use Interval Partitioning 3-51
3.5.6 When to Use Reference Partitioning 3-51
3.5.7 When to Partition on Virtual Columns 3-52
3.5.8 Considerations When Using Read-Only Tablespaces 3-53

4 Partition Administration

4.1 Specifying Partitioning When Creating Tables and Indexes 4-1
4.1.1 About Creating Range-Partitioned Tables and Global Indexes 4-3
4.1.1.1 Creating a Range-Partitioned Table 4-3
4.1.1.2 Creating a Range-Partitioned Table With More Complexity 4-4
4.1.1.3 Creating a Range-Partitioned Global Index 4-5
4.1.2 Creating Range-Interval-Partitioned Tables 4-5
4.1.3 About Creating Hash Partitioned Tables and Global Indexes 4-6
4.1.3.1 Creating a Hash Partitioned Table 4-6
4.1.3.2 Creating a Hash Partitioned Global Index 4-7
4.1.4 About Creating List-Partitioned Tables 4-8
4.1.4.1 Creating a List-Partitioned Table 4-8
4.1.4.2 Creating a List-Partitioned Table With a Default Partition 4-9
4.1.4.3 Creating an Automatic List-Partitioned Table 4-10
4.1.4.4 Creating a Multi-column List-Partitioned Table 4-11
4.1.5 Creating Reference-Partitioned Tables 4-13
4.1.6 Creating Interval-Reference Partitioned Tables 4-15
4.1.7 Creating a Table Using In-Memory Column Store With Partitioning 4-16
4.1.8 Creating a Table with Read-Only Partitions or Subpartitions 4-17
4.1.9 Creating a Partitioned External Table 4-18
4.1.10 Specifying Partitioning on Key Columns 4-19
4.1.10.1 Creating a Multicolumn Range-Partitioned Table By Date 4-19
4.1.10.2 Creating a Multicolumn Range-Partitioned Table to Enforce Equal-Sized Partitions 4-21
4.1.11 Using Virtual Column-Based Partitioning 4-22
4.1.12 Using Table Compression with Partitioned Tables 4-23
4.1.13 Using Key Compression with Partitioned Indexes 4-23
4.1.14 Specifying Partitioning with Segments 4-23
4.1.14.1 Deferred Segment Creation for Partitioning 4-24
4.1.14.2 Truncating Segments That Are Empty 4-24
4.1.14.3 Maintenance Procedures for Segment Creation on Demand 4-25
4.1.15 Specifying Partitioning When Creating Index-Organized Tables 4-25

4.1.15.1 Creating Range-Partitioned Index-Organized Tables 4-26
4.1.15.2 Creating Hash Partitioned Index-Organized Tables 4-26
4.1.15.3 Creating List-Partitioned Index-Organized Tables 4-27
4.1.16 Partitioning Restrictions for Multiple Block Sizes 4-27
4.1.17 Partitioning of Collections in XMLType and Objects 4-28
4.1.17.1 Performing PMOs on Partitions that Contain Collection Tables 4-29
4.1.17.2 Partitioning of XMLIndex for Binary XML Tables 4-30
4.2 Specifying Composite Partitioning When Creating Tables 4-30
4.2.1 Creating Composite Hash-* Partitioned Tables 4-31
4.2.2 Creating Composite Interval-* Partitioned Tables 4-31
4.2.2.1 Creating Composite Interval-Hash Partitioned Tables 4-32
4.2.2.2 Creating Composite Interval-List Partitioned Tables 4-33
4.2.2.3 Creating Composite Interval-Range Partitioned Tables 4-34
4.2.3 Creating Composite List-* Partitioned Tables 4-34
4.2.3.1 Creating Composite List-Hash Partitioned Tables 4-35
4.2.3.2 Creating Composite List-List Partitioned Tables 4-35
4.2.3.3 Creating Composite List-Range Partitioned Tables 4-36
4.2.4 Creating Composite Range-* Partitioned Tables 4-37
4.2.4.1 About Creating Composite Range-Hash Partitioned Tables 4-38
4.2.4.2 About Creating Composite Range-List Partitioned Tables 4-39
4.2.4.3 Creating Composite Range-Range Partitioned Tables 4-42
4.2.5 Specifying Subpartition Templates to Describe Composite Partitioned Tables 4-43
4.2.5.1 Specifying a Subpartition Template for a *-Hash Partitioned Table 4-44
4.2.5.2 Specifying a Subpartition Template for a *-List Partitioned Table 4-45
4.3 Maintenance Operations Supported on Partitions 4-45
4.3.1 Updating Indexes Automatically 4-50
4.3.2 Asynchronous Global Index Maintenance for Dropping and Truncating Partitions 4-52
4.3.3 Modifying a Subpartition Template 4-53
4.3.4 Filtering Maintenance Operations 4-53
4.4 Maintenance Operations for Partitioned Tables and Indexes 4-54
4.4.1 Restrictions and Special Requirements for Multi-Partition Operations on Tables with Domain Indexes 4-55
4.4.2 About Adding Partitions and Subpartitions 4-56
4.4.2.1 Adding a Partition to a Range-Partitioned Table 4-57
4.4.2.2 Adding a Partition to a Hash-Partitioned Table 4-57
4.4.2.3 Adding a Partition to a List-Partitioned Table 4-57
4.4.2.4 Adding a Partition to an Interval-Partitioned Table 4-58
4.4.2.5 About Adding Partitions to a Composite *-Hash Partitioned Table 4-59
4.4.2.6 About Adding Partitions to a Composite *-List Partitioned Table 4-59
4.4.2.7 About Adding Partitions to a Composite *-Range Partitioned Table 4-60
4.4.2.8 About Adding a Partition or Subpartition to a Reference-Partitioned Table 4-61
4.4.2.9 Adding Index Partitions 4-62

4.4.2.10 Adding Multiple Partitions 4-62
4.4.3 About Coalescing Partitions and Subpartitions 4-63
4.4.3.1 Coalescing a Partition in a Hash Partitioned Table 4-64
4.4.3.2 Coalescing a Subpartition in a *-Hash Partitioned Table 4-64
4.4.3.3 Coalescing Hash Partitioned Global Indexes 4-64
4.4.4 About Dropping Partitions and Subpartitions 4-65
4.4.4.1 Dropping Table Partitions 4-65
4.4.4.2 Dropping Interval Partitions 4-68
4.4.4.3 Dropping Index Partitions 4-68
4.4.4.4 Dropping Multiple Partitions 4-68
4.4.5 About Exchanging Partitions and Subpartitions 4-69
4.4.5.1 Creating a Table for Exchange With a Partitioned Table 4-71
4.4.5.2 Exchanging a Range, Hash, or List Partition 4-72
4.4.5.3 Exchanging a Partition of an Interval Partitioned Table 4-73
4.4.5.4 Exchanging a Partition of a Reference-Partitioned Table 4-74
4.4.5.5 About Exchanging a Partition of a Table with Virtual Columns 4-75
4.4.5.6 Exchanging a Hash Partitioned Table with a *-Hash Partition 4-75
4.4.5.7 Exchanging a Subpartition of a *-Hash Partitioned Table 4-76
4.4.5.8 Exchanging a List-Partitioned Table with a *-List Partition 4-76
4.4.5.9 About Exchanging a Subpartition of a *-List Partitioned Table 4-77
4.4.5.10 Exchanging a Range-Partitioned Table with a *-Range Partition 4-77
4.4.5.11 About Exchanging a Subpartition of a *-Range Partitioned Table 4-78
4.4.5.12 About Exchanging a Partition with the Cascade Option 4-78
4.4.6 About Merging Partitions and Subpartitions 4-80
4.4.6.1 Merging Range Partitions 4-81
4.4.6.2 Merging Interval Partitions 4-82
4.4.6.3 Merging List Partitions 4-83
4.4.6.4 Merging *-Hash Partitions 4-83
4.4.6.5 About Merging *-List Partitions 4-84
4.4.6.6 About Merging *-Range Partitions 4-85
4.4.6.7 Merging Multiple Partitions 4-86
4.4.7 About Modifying Attributes of Tables, Partitions, and Subpartitions 4-87
4.4.7.1 About Modifying Default Attributes 4-87
4.4.7.2 About Modifying Real Attributes of Partitions 4-88
4.4.8 About Modifying List Partitions 4-90
4.4.8.1 About Modifying List Partitions: Adding Values 4-90
4.4.8.2 About Modifying List Partitions: Dropping Values 4-91
4.4.9 About Modifying the Partitioning Strategy 4-92
4.4.10 About Moving Partitions and Subpartitions 4-94
4.4.10.1 Moving Table Partitions 4-95
4.4.10.2 Moving Subpartitions 4-95
4.4.10.3 Moving Index Partitions 4-96

4.4.11 About Rebuilding Index Partitions 4-96
4.4.11.1 About Rebuilding Global Index Partitions 4-96
4.4.11.2 About Rebuilding Local Index Partitions 4-96
4.4.12 About Renaming Partitions and Subpartitions 4-97
4.4.12.1 Renaming a Table Partition 4-98
4.4.12.2 Renaming a Table Subpartition 4-98
4.4.12.3 About Renaming Index Partitions 4-98
4.4.13 About Splitting Partitions and Subpartitions 4-98
4.4.13.1 Splitting a Partition of a Range-Partitioned Table 4-99
4.4.13.2 Splitting a Partition of a List-Partitioned Table 4-101
4.4.13.3 Splitting a Partition of an Interval-Partitioned Table 4-104
4.4.13.4 Splitting a *-Hash Partition 4-104
4.4.13.5 Splitting Partitions in a *-List Partitioned Table 4-105
4.4.13.6 Splitting a *-Range Partition 4-107
4.4.13.7 Splitting Index Partitions 4-108
4.4.13.8 Splitting into Multiple Partitions 4-108
4.4.13.9 Fast SPLIT PARTITION and SPLIT SUBPARTITION Operations 4-110
4.4.14 About Truncating Partitions and Subpartitions 4-111
4.4.14.1 About Truncating a Table Partition 4-111
4.4.14.2 Truncating Multiple Partitions 4-113
4.4.14.3 Truncating Subpartitions 4-115
4.4.14.4 Truncating a Partition with the Cascade Option 4-117
4.5 About Dropping Partitioned Tables 4-118
4.6 Changing a Nonpartitioned Table into a Partitioned Table 4-119
4.6.1 Using Online Redefinition to Partition Collection Tables 4-119
4.6.2 Converting a Non-Partitioned Table to a Partitioned Table 4-121
4.7 Managing Hybrid Partitioned Tables 4-122
4.7.1 Creating Hybrid Partitioned Tables 4-123
4.7.1.1 Creating a Hybrid Range-Partitioned Table 4-123
4.7.1.2 Creating a Hybrid Partitioned Table Containing Interval Partitions 4-125
4.7.1.3 Creating a Hybrid Partitioned Table With Automatic List Partitioning 4-126
4.7.2 Converting to Hybrid Partitioned Tables 4-126
4.7.3 Converting Hybrid Partitioned Tables to Internal Partitioned Tables 4-127
4.7.4 Using ADO With Hybrid Partitioned Tables 4-128
4.7.5 Splitting Partitions in a Hybrid Partitioned Table 4-129
4.7.6 Exchanging Data in Hybrid Partitioned Tables 4-130
4.8 Viewing Information About Partitioned Tables and Indexes 4-133

5 Managing and Maintaining Time-Based Information

5.1 Managing Data in Oracle Database With ILM 5-1
5.1.1 About Oracle Database for ILM 5-2

5.1.1.1 Oracle Database Manages All Types of Data 5-2
5.1.1.2 Regulatory Requirements 5-3
5.1.1.3 The Benefits of an Online Archive 5-3
5.1.2 Implementing ILM Using Oracle Database 5-4
5.1.2.1 Step 1: Define the Data Classes 5-4
5.1.2.2 Step 2: Create Storage Tiers for the Data Classes 5-7
5.1.2.3 Step 3: Create Data Access and Migration Policies 5-9
5.1.2.4 Step 4: Define and Enforce Compliance Policies 5-11
5.2 Implementing an ILM Strategy With Heat Map and ADO 5-12
5.2.1 Using Heat Map 5-12
5.2.1.1 Enabling and Disabling Heat Map 5-13
5.2.1.2 Displaying Heat Map Tracking Data With Views 5-14
5.2.1.3 Managing Heat Map Data With DBMS_HEAT_MAP Subprograms 5-16
5.2.2 Using Automatic Data Optimization 5-16
5.2.2.1 Managing Policies for Automatic Data Optimization 5-17
5.2.2.2 Creating a Table With an ILM ADO Policy 5-19
5.2.2.3 Adding ILM ADO Policies 5-20
5.2.2.4 Disabling and Deleting ILM ADO Policies 5-21
5.2.2.5 Specifying Segment-Level Compression and Storage Tiering With ADO 5-21
5.2.2.6 Specifying Row-Level Compression Tiering With ADO 5-22
5.2.2.7 Managing ILM ADO Parameters 5-23
5.2.2.8 Using PL/SQL Functions for Policy Management 5-24
5.2.2.9 Using Views to Monitor Policies for ADO 5-25
5.2.3 Limitations and Restrictions With ADO and Heat Map 5-26
5.3 Controlling the Validity and Visibility of Data in Oracle Database 5-26
5.3.1 Using In-Database Archiving 5-27
5.3.2 Using Temporal Validity 5-32
5.3.3 Creating a Table With Temporal Validity 5-34
5.3.4 Limitations and Restrictions With In-Database Archiving and Temporal Validity 5-35
5.4 Implementing an ILM System Manually Using Partitioning 5-36
5.5 Managing ILM Heat Map and ADO with Oracle Enterprise Manager 5-39
5.5.1 Accessing the Database Administration Menu 5-40
5.5.2 Viewing Automatic Data Optimization Activity at the Tablespace Level 5-40
5.5.3 Viewing the Segment Activity Details of Any Tablespace 5-40
5.5.4 Viewing the Segment Activity Details of Any Object 5-41
5.5.5 Viewing the Segment Activity History of Any Object 5-41
5.5.6 Searching Segment Activity in Automatic Data Optimization 5-42
5.5.7 Viewing Policies for a Segment 5-42
5.5.8 Disabling Background Activity 5-42
5.5.9 Changing Execution Frequency of Background Automatic Data Optimization 5-43
5.5.10 Viewing Policy Executions In the Last 24 Hours 5-43
5.5.11 Viewing Objects Moved In Last 24 Hours 5-44

5.5.12 Viewing Policy Details 5-44
5.5.13 Viewing Objects Associated With a Policy 5-44
5.5.14 Evaluating a Policy Before Execution 5-44
5.5.15 Executing a Single Policy 5-45
5.5.16 Stopping a Policy Execution 5-45
5.5.17 Viewing Policy Execution History 5-46
5.6 Using Automatic Storage Compression 5-46
5.6.1 Example: Using Automatic Storage Compression 5-47

6 Using Partitioning in a Data Warehouse Environment

6.1 What Is a Data Warehouse? 6-1
6.2 Scalability in a Data Warehouse 6-1
6.2.1 Bigger Databases 6-2
6.2.2 Bigger Individual Tables: More Rows in Tables 6-2
6.2.3 More Users Querying the System 6-2
6.2.4 More Complex Queries 6-2
6.3 Partitioning for Performance in a Data Warehouse 6-3
6.3.1 Partition Pruning in a Data Warehouse 6-3
6.3.1.1 Basic Partition Pruning Techniques 6-3
6.3.1.2 Advanced Partition Pruning Techniques 6-4
6.3.2 Partition-Wise Joins in a Data Warehouse 6-5
6.3.2.1 Full Partition-Wise Joins 6-6
6.3.2.2 Partial Partition-Wise Joins 6-7
6.3.2.3 Benefits of Partition-Wise Joins 6-9
6.3.2.4 Performance Considerations for Parallel Partition-Wise Joins 6-10
6.3.3 Indexes and Partitioned Indexes in a Data Warehouse 6-10
6.3.3.1 Local Partitioned Indexes 6-11
6.3.3.2 Nonpartitioned Indexes 6-11
6.3.3.3 Global Partitioned Indexes 6-12
6.3.4 Materialized Views and Partitioning in a Data Warehouse 6-12
6.3.4.1 Partitioned Materialized Views 6-13
6.4 Manageability in a Data Warehouse 6-14
6.4.1 Partition Exchange Load 6-14
6.4.2 Partitioning and Indexes 6-15
6.4.3 Removing Data from Tables 6-15
6.4.4 Partitioning and Data Compression 6-15

7 Using Partitioning in an Online Transaction Processing Environment

7.1 What Is an Online Transaction Processing System? 7-1
7.2 Performance in an Online Transaction Processing Environment 7-3

7.2.1 Deciding Whether to Partition Indexes 7-3
7.2.2 How to Use Partitioning on Index-Organized Tables 7-4
7.3 Manageability in an Online Transaction Processing Environment 7-5
7.3.1 Impact of a Partition Maintenance Operation on a Partitioned Table with Local Indexes 7-5
7.3.2 Impact of a Partition Maintenance Operation on Global Indexes 7-6
7.3.3 Common Partition Maintenance Operations in OLTP Environments 7-6
7.3.3.1 Removing (Purging) Old Data 7-6
7.3.3.2 Moving or Merging Older Partitions to a Low-Cost Storage Tier Device 7-7

8 Using Parallel Execution

8.1 Parallel Execution Concepts 8-1
8.1.1 When to Implement Parallel Execution 8-2
8.1.2 When Not to Implement Parallel Execution 8-2
8.1.3 Fundamental Hardware Requirements 8-3
8.1.4 How Parallel Execution Works 8-3
8.1.4.1 Parallel Execution of SQL Statements 8-4
8.1.4.2 Producer/Consumer Model 8-4
8.1.4.3 Granules of Parallelism 8-5
8.1.4.4 Distribution Methods Between Producers and Consumers 8-7
8.1.4.5 How Parallel Execution Servers Communicate 8-9
8.1.5 Parallel Execution Server Pool 8-10
8.1.5.1 Processing without Enough Parallel Execution Servers 8-10
8.1.6 Balancing the Workload to Optimize Performance 8-11
8.1.7 Multiple Parallelizers 8-11
8.1.8 Parallel Execution on Oracle RAC 8-12
8.2 Setting the Degree of Parallelism 8-13
8.2.1 Manually Specifying the Degree of Parallelism 8-14
8.2.2 Default Degree of Parallelism 8-14
8.2.3 Automatic Degree of Parallelism 8-15
8.2.4 Determining Degree of Parallelism in Auto DOP 8-15
8.2.5 Controlling Automatic Degree of Parallelism 8-17
8.2.6 Adaptive Parallelism 8-18
8.3 In-Memory Parallel Execution 8-18
8.3.1 Buffer Cache Usage in Parallel Execution 8-19
8.3.2 Automatic Big Table Caching 8-19
8.4 Parallel Statement Queuing 8-20
8.4.1 About Managing Parallel Statement Queuing with Oracle Database Resource Manager 8-21
8.4.1.1 About Managing the Order of the Parallel Statement Queue 8-23
8.4.1.2 About Limiting the Parallel Server Resources for a Consumer Group 8-23
8.4.1.3 Specifying a Parallel Statement Queue Timeout for Each Consumer Group 8-24

8.4.1.4 Specifying a Degree of Parallelism Limit for Consumer Groups 8-25
8.4.1.5 Critical Parallel Statement Prioritization 8-25
8.4.1.6 A Sample Scenario for Managing Statements in the Parallel Queue 8-26
8.4.2 Grouping Parallel Statements with BEGIN_SQL_BLOCK END_SQL_BLOCK 8-28
8.4.3 About Managing Parallel Statement Queuing with Hints 8-29
8.5 Types of Parallelism 8-29
8.5.1 About Parallel Queries 8-30
8.5.1.1 Parallel Queries on Index-Organized Tables 8-30
8.5.1.2 Parallel Queries on Object Types 8-31
8.5.1.3 Rules for Parallelizing Queries 8-31
8.5.2 About Parallel DDL Statements 8-32
8.5.2.1 DDL Statements That Can Be Parallelized 8-32
8.5.2.2 About Using CREATE TABLE AS SELECT in Parallel 8-33
8.5.2.3 Recoverability and Parallel DDL 8-34
8.5.2.4 Space Management for Parallel DDL 8-34
8.5.2.5 Storage Space When Using Dictionary-Managed Tablespaces 8-34
8.5.2.6 Free Space and Parallel DDL 8-34
8.5.2.7 Rules for DDL Statements 8-35
8.5.2.8 Rules for CREATE TABLE AS SELECT 8-36
8.5.3 About Parallel DML Operations 8-36
8.5.3.1 When to Use Parallel DML 8-37
8.5.3.2 Enable Parallel DML Mode 8-38
8.5.3.3 Rules for UPDATE, MERGE, and DELETE 8-39
8.5.3.4 Rules for INSERT SELECT 8-40
8.5.3.5 Transaction Restrictions for Parallel DML 8-40
8.5.3.6 Rollback Segments 8-41
8.5.3.7 Recovery for Parallel DML 8-41
8.5.3.8 Space Considerations for Parallel DML 8-42
8.5.3.9 Restrictions on Parallel DML 8-42
8.5.3.10 Data Integrity Restrictions 8-43
8.5.3.11 Trigger Restrictions 8-44
8.5.3.12 Distributed Transaction Restrictions 8-45
8.5.3.13 Examples of Distributed Transaction Parallelization 8-45
8.5.3.14 Concurrent Execution of Union All 8-45
8.5.4 About Parallel Execution of Functions 8-46
8.5.4.1 Functions in Parallel Queries 8-47
8.5.4.2 Functions in Parallel DML and DDL Statements 8-47
8.5.5 About Other Types of Parallelism 8-48
8.5.6 Degree of Parallelism Rules for SQL Statements 8-48
8.6 About Initializing and Tuning Parameters for Parallel Execution 8-50
8.6.1 Default Parameter Settings 8-50
8.6.2 Forcing Parallel Execution for a Session 8-52

8.6.3 Tuning General Parameters for Parallel Execution 8-52
8.6.3.1 Parameters Establishing Resource Limits for Parallel Operations 8-52
8.6.3.2 Parameters Affecting Resource Consumption 8-59
8.6.3.3 Parameters Related to I/O 8-64
8.7 Monitoring Parallel Execution Performance 8-65
8.7.1 Monitoring Parallel Execution Performance with Dynamic Performance Views 8-66
8.7.1.1 V$PX_BUFFER_ADVICE 8-66
8.7.1.2 V$PX_SESSION 8-66
8.7.1.3 V$PX_SESSTAT 8-66
8.7.1.4 V$PX_PROCESS 8-67
8.7.1.5 V$PX_PROCESS_DETAIL 8-67
8.7.1.6 V$PX_PROCESS_SYSSTAT 8-67
8.7.1.7 V$PQ_SESSTAT 8-67
8.7.1.8 V$PQ_TQSTAT 8-68
8.7.1.9 V$RSRC_CONS_GROUP_HISTORY 8-68
8.7.1.10 V$RSRC_CONSUMER_GROUP 8-69
8.7.1.11 V$RSRC_PLAN 8-69
8.7.1.12 V$RSRC_PLAN_HISTORY 8-69
8.7.1.13 V$RSRC_SESSION_INFO 8-69
8.7.1.14 V$RSRCMGRMETRIC 8-69
8.7.2 Monitoring Session Statistics 8-70
8.7.3 Monitoring System Statistics 8-71
8.7.4 Monitoring Operating System Statistics 8-72
8.8 Tips for Tuning Parallel Execution 8-72
8.8.1 Implementing a Parallel Execution Strategy 8-73
8.8.2 Optimizing Performance by Creating and Populating Tables in Parallel 8-73
8.8.3 Using EXPLAIN PLAN to Show Parallel Operations Plans 8-74
8.8.3.1 Example: Using EXPLAIN PLAN to Show Parallel Operations 8-75
8.8.4 Additional Considerations for Parallel DML 8-75
8.8.4.1 Parallel DML and Direct-Path Restrictions 8-75
8.8.4.2 Limitation on the Degree of Parallelism 8-76
8.8.4.3 When to Increase INITRANS 8-76
8.8.4.4 Limitation on Available Number of Transaction Free Lists for Segments 8-76
8.8.4.5 Multiple Archivers for Large Numbers of Redo Logs 8-77
8.8.4.6 Database Writer Process (DBWn) Workload 8-77
8.8.4.7 [NO]LOGGING Clause 8-77
8.8.5 Optimizing Performance by Creating Indexes in Parallel 8-78
8.8.6 Parallel DML Tips 8-79
8.8.6.1 Parallel DML Tip 1: INSERT 8-79
8.8.6.2 Parallel DML Tip 2: Direct-Path INSERT 8-80
8.8.6.3 Parallel DML Tip 3: Parallelizing INSERT, MERGE, UPDATE, and DELETE 8-80
8.8.7 Incremental Data Loading in Parallel 8-82

8.8.7.1 Optimizing Performance for Updating the Table in Parallel 8-82
8.8.7.2 Efficiently Inserting the New Rows into the Table in Parallel 8-83
8.8.7.3 Optimizing Performance by Merging in Parallel 8-83

9 Backing Up and Recovering VLDBs

9.1 Data Warehouses 9-1
9.1.1 Data Warehouse Characteristics 9-2
9.2 Oracle Backup and Recovery 9-2
9.2.1 Physical Database Structures Used in Recovering Data 9-3
9.2.1.1 Data files 9-3
9.2.1.2 Redo Logs 9-3
9.2.1.3 Control Files 9-3
9.2.2 Backup Type 9-4
9.2.3 Backup Tools 9-4
9.2.3.1 Oracle Recovery Manager (RMAN) 9-5
9.2.3.2 Oracle Data Pump 9-5
9.2.3.3 User-Managed Backups 9-5
9.3 Data Warehouse Backup and Recovery 9-6
9.3.1 Recovery Time Objective (RTO) 9-6
9.3.2 Recovery Point Objective (RPO) 9-6
9.3.2.1 More Data Means a Longer Backup Window 9-7
9.3.2.2 Divide and Conquer 9-7
9.4 The Data Warehouse Recovery Methodology 9-7
9.4.1 Best Practice 1: Use ARCHIVELOG Mode 9-8
9.4.1.1 Is Downtime Acceptable? 9-8
9.4.2 Best Practice 2: Use RMAN 9-9
9.4.3 Best Practice 3: Use Block Change Tracking 9-9
9.4.4 Best Practice 4: Use RMAN Multisection Backups 9-9
9.4.5 Best Practice 5: Leverage Read-Only Tablespaces 9-10
9.4.6 Best Practice 6: Plan for NOLOGGING Operations in Your Backup/Recovery Strategy 9-10
9.4.6.1 Extract, Transform, and Load 9-11
9.4.6.2 The Extract, Transform, and Load Strategy 9-12
9.4.6.3 Incremental Backup 9-12
9.4.6.4 The Incremental Approach 9-13
9.4.6.5 Flashback Database and Guaranteed Restore Points 9-13
9.4.7 Best Practice 7: Not All Tablespaces Should Be Treated Equally 9-14

10 Storage Management for VLDBs

10.1 High Availability 10-1

10.1.1 Hardware-Based Mirroring 10-2
10.1.1.1 RAID 1 Mirroring 10-2
10.1.1.2 RAID 5 Mirroring 10-2
10.1.2 Mirroring Using Oracle ASM 10-3
10.2 Performance 10-3
10.2.1 Hardware-Based Striping 10-4
10.2.1.1 RAID 0 Striping 10-4
10.2.1.2 RAID 5 Striping 10-4
10.2.2 Striping Using Oracle ASM 10-4
10.2.3 Information Lifecycle Management 10-5
10.2.4 Partition Placement 10-5
10.2.5 Bigfile Tablespaces 10-6
10.2.6 Oracle Database File System (DBFS) 10-6
10.3 Scalability and Manageability 10-7
10.3.1 Stripe and Mirror Everything (SAME) 10-7
10.3.2 SAME and Manageability 10-7
10.4 Oracle ASM Settings Specific to VLDBs 10-7

Glossary

Index

Preface

This book contains an overview of very large database (VLDB) topics, with emphasis on partitioning and parallel execution as a key component of the VLDB strategy. Partitioning enhances the performance, manageability, and availability of a wide variety of applications and helps reduce the total cost of ownership for storing large amounts of data. Parallel execution enables the processing of large volumes of data and expensive SQL operations, speeding up processing times significantly.

This Preface contains the following topics:

- Audience
- Documentation Accessibility
- Related Documents
- Conventions

Audience

This document is intended for database administrators (DBAs) and developers who create, manage, and write applications for very large databases (VLDB).

Documentation Accessibility

For information about Oracle's commitment to accessibility, visit the Oracle Accessibility Program website at http://www.oracle.com/pls/topic/lookup?ctx=acc&id=docacc.

Access to Oracle Support

Oracle customers that have purchased support have access to electronic support through My Oracle Support. For information, visit http://www.oracle.com/pls/topic/lookup?ctx=acc&id=info or visit http://www.oracle.com/pls/topic/lookup?ctx=acc&id=trs if you are hearing impaired.

Related Documents

For more information, see the following Oracle resources:

- *Oracle Database Concepts*
- *Oracle Database Administrator's Guide*
- *Oracle Database Data Warehousing Guide*
- *Oracle Database Reference*
- *Oracle Database SQL Language Reference*
- *Oracle Database SQL Tuning Guide*

- *Oracle Database Performance Tuning Guide*

> **See Also:**
>
> - *Oracle Database Licensing Information User Manual* to determine whether a feature is available on your edition of Oracle Database
> - *Oracle Database New Features* for a complete description of the new features in this release
> - *Oracle Database Upgrade Guide* for a complete description of the deprecated and desupported features in this release

Conventions

The following text conventions are used in this document:

Convention	Meaning
boldface	Boldface type indicates graphical user interface elements associated with an action, or terms defined in text or the glossary.
italic	Italic type indicates book titles, emphasis, or placeholder variables for which you supply particular values.
`monospace`	Monospace type indicates commands within a paragraph, URLs, code in examples, text that appears on the screen, or text that you enter.

1

Introduction to Very Large Databases

Very large databases (VLDBs) present administration challenges that require multiple strategies. Partitioning is a key component of the VLDB strategy.

Modern enterprises frequently run mission-critical databases containing upwards of several hundred gigabytes, and often several terabytes of data. These enterprises are challenged by the support and maintenance requirements of very large databases (VLDB), and must devise methods to meet those challenges.

This chapter contains the following sections:

- #unique_17
- Introduction to Partitioning
- VLDB and Partitioning
- Partitioning As the Foundation for Information Lifecycle Management
- Partitioning for All Databases

> **Note:**
>
> Partitioning functionality is available only if you purchase the Oracle Partitioning option.

1.1 Introduction to Partitioning

Partitioning provides support for very large tables and indexes by subdividing them into smaller and more manageable pieces.

Partitioning addresses key issues in supporting very large tables and indexes by decomposing them into smaller and more manageable pieces called **partitions**, which are entirely transparent to an application. SQL queries and Data Manipulation Language (DML) statements do not need to be modified to access partitioned tables. However, after partitions are defined, data definition language (DDL) statements can access and manipulate individual partitions rather than entire tables or indexes. This is how partitioning can simplify the manageability of large database objects.

Each partition of a table or index must have the same logical attributes, such as column names, data types, and constraints, but each partition can have separate physical attributes, such as compression enabled or disabled, physical storage settings, and tablespaces.

Partitioning is useful for many different types of applications, particularly applications that manage large volumes of data. OLTP systems often benefit from improvements in manageability and availability, while data warehousing systems benefit from performance and manageability.

Partitioning offers these advantages:

- It enables data management operations such as data loads, index creation and rebuilding, and backup and recovery at the partition level, rather than on the entire table. This results in significantly reduced times for these operations.
- It improves query performance. Often the results of a query can be achieved by accessing a subset of partitions, rather than the entire table. For some queries, this technique (called **partition pruning**) can provide order-of-magnitude gains in performance.
- It significantly reduces the impact of scheduled downtime for maintenance operations.

  Partition independence for partition maintenance operations lets you perform concurrent maintenance operations on different partitions of the same table or index. You can also run concurrent `SELECT` and DML operations against partitions that are unaffected by maintenance operations.
- It increases the availability of mission-critical databases if critical tables and indexes are divided into partitions to reduce the maintenance windows, recovery times, and impact of failures.
- Parallel execution provides specific advantages to optimize resource utilization, and minimize execution time. Parallel execution is supported for queries and for DML and DDL.

Partitioning enables faster data access within Oracle Database. Whether a database has 10 GB or 10 TB of data, partitioning can improve data access by orders of magnitude. Partitioning can be implemented without requiring any modifications to your applications. For example, you could convert a nonpartitioned table to a partitioned table without needing to modify any of the `SELECT` statements or DML statements that access that table. You do not need to rewrite your application code to take advantage of partitioning.

1.2 VLDB and Partitioning

Partitioning is a valuable strategy for managing for very large databases (VLDBs).

A very large database has no minimum absolute size. Although a VLDB is a database like smaller databases, there are specific challenges in managing a VLDB. These challenges are related to the sheer size and the cost-effectiveness of performing operations against a system of that size.

Several trends have been responsible for the steady growth in database size:

- For a long time, systems have been developed in isolation. Companies have started to see the benefits of combining these systems to enable cross-departmental analysis while reducing system maintenance costs. Consolidation of databases and applications is a key factor in the ongoing growth of database size.
- Many companies face regulations for storing data for a minimum amount of time. The regulations generally result in more data being stored for longer periods of time.
- Companies grow by expanding sales and operations or through mergers and acquisitions, causing the amount of generated and processed data to increase. At the same time, the user population that relies on the database for daily activities increases.

Partitioning is a critical feature for managing very large databases. Growth is the basic challenge that partitioning addresses for very large databases, and partitioning enables a *divide and conquer* technique for managing the tables and indexes in the database, especially as those tables and indexes grow. Partitioning is the feature that allows a database to scale for very large data sets while maintaining consistent performance, without unduly increasing administrative or hardware resources.

See Also:

- Partitioning for Availability, Manageability, and Performance for information about availability, manageability, and performance considerations for partitioning implementations
- Backing Up and Recovering VLDBs for information about the challenges surrounding backup and recovery for very large databases
- Storage Management for VLDBs for information about best practices for storage, which is a key component of very large databases

1.3 Partitioning As the Foundation for Information Lifecycle Management

Partitioning provides support for Information Lifecycle Management (ILM).

Information Lifecycle Management (ILM) is a set of processes and policies for managing data throughout its useful life. One important component of an ILM strategy is determining the most appropriate and cost-effective medium for storing data at any point during its lifetime: newer data used in day-to-day operations is stored on the fastest, most highly-available storage tier, while older data which is accessed infrequently may be stored on a less expensive and less efficient storage tier. Older data may also be updated less frequently so it makes sense to compress and store the data as read-only.

Oracle Database provides the ideal environment for implementing your ILM solution. Oracle supports multiple storage tiers, and because all of the data remains in Oracle Database, multiple storage tiers are transparent to the application and the data continues to be secure. Partitioning provides the fundamental technology that enables data in tables to be stored in different partitions.

Although multiple storage tiers and sophisticated ILM policies are most often found in enterprise-level systems, most companies and most databases need some degree of information lifecycle management. The most basic of ILM operations, archiving older data and purging or removing that data from the database, can be orders of magnitude faster when using partitioning.

See Also:

Managing and Maintaining Time-Based Information for more information about ILM

1.4 Partitioning for All Databases

Partitioning provides benefits for large and small databases.

The benefits of partitioning are not just for very large databases; all databases, even small databases, can benefit from partitioning. Even a database whose size is measured in megabytes can gain the same type of performance and manageability benefits from partitioning as the largest multi-terabyte system.

See Also:

- Using Partitioning in a Data Warehouse Environment for more information about how partitioning can provide benefits in a data warehouse environment
- Using Partitioning in an Online Transaction Processing Environment for more information about how partitioning can provide benefits in an OLTP environment

2

Partitioning Concepts

Partitioning enhances the performance, manageability, and availability of a wide variety of applications and helps reduce the total cost of ownership for storing large amounts of data.

Partitioning allows tables, indexes, and index-organized tables to be subdivided into smaller pieces, enabling these database objects to be managed and accessed at a finer level of granularity. Oracle provides a rich variety of partitioning strategies and extensions to address every business requirement. Because it is entirely transparent, partitioning can be applied to almost any application without the need for potentially expensive and time consuming application changes.

This chapter contains the following sections:

- Partitioning Overview
- Benefits of Partitioning
- Partitioning Strategies
- Partitioning Extensions
- Indexing on Partitioned Tables

2.1 Partitioning Overview

Partitioning provides a technique to subdivide objects into smaller pieces.

Partitioning allows a table, index, or index-organized table to be subdivided into smaller pieces, where each piece of such a database object is called a partition. Each partition has its own name, and may optionally have its own storage characteristics.

The following topics are discussed:

- Basics of Partitioning
- Partitioning Key
- Partitioned Tables
- Partitioned Index-Organized Tables
- System Partitioning
- Partitioning for Information Lifecycle Management
- Range Partitioning for Hash Clusters
- Partitioning and LOB Data
- Partitioning on External Tables
- Hybrid Partitioned Tables
- Collections in XMLType and Object Data

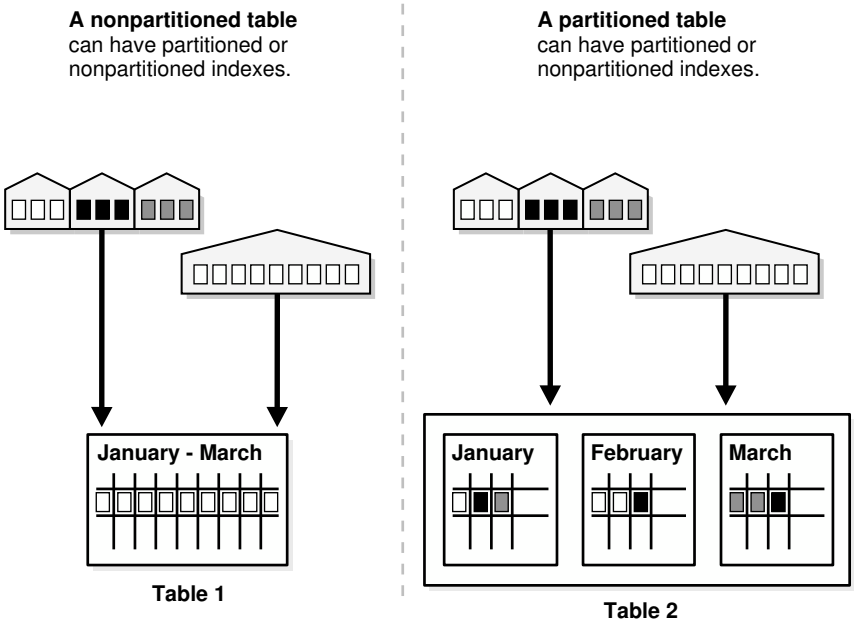
2.1.1 Basics of Partitioning

Partitioning enables administration of an object either collectively or individually.

From the perspective of a database administrator, a partitioned object has multiple pieces that can be managed either collectively or individually. This gives an administrator considerable flexibility in managing partitioned objects. However, from the perspective of the application, a partitioned table is identical to a nonpartitioned table; no modifications are necessary when accessing a partitioned table using SQL queries and DML statements.

Figure 2-1 offers a graphical view of how partitioned tables differ from nonpartitioned tables.

Figure 2-1 Views of Partitioned and Nonpartitioned Tables



> **Note:**
>
> All partitions of a partitioned object must reside in tablespaces of the same block size.

> **See Also:**
>
> - *Oracle Database Concepts* for more information about multiple block sizes
> - *Oracle Database SQL Language Reference* for general restrictions on partitioning, the exact syntax of the partitioning clauses for creating and altering partitioned tables and indexes, any restrictions on their use, and specific privileges required for creating and altering tables

2.1.2 Partitioning Key

Each row in a partitioned table is unambiguously assigned to a single partition using a key.

The partitioning key consists of one or more columns that determine the partition where each row is stored. Oracle automatically directs insert, update, and delete operations to the appropriate partition with the partitioning key.

Validating Partition Content

You can identify whether rows in a partition are conformant to the partition definition or whether the partition key of the row is violating the partition definition with the `ORA_PARTITION_VALIDATION` SQL function. The SQL function takes a rowid as input and returns `1` if the row is in the correct partition and `0` otherwise. The function is applicable for internal, external, and hybrid partitioned tables for both internal and external partitions and subpartitions.

For example:

```
SQL>  CREATE TABLE test1 (column1 NUMBER)
             PARTITION BY RANGE(column1)
             (PARTITION p1 VALUES LESS THAN (10),
              PARTITION p2 VALUES LESS THAN (20));

SQL> CREATE TABLE test2 (column1 NUMBER);

SQL> INSERT INTO test1 VALUES (1);

SQL> INSERT INTO test2 VALUES (99);

SQL> ALTER TABLE test1 EXCHANGE PARTITION p2 WITH TABLE test2 WITHOUT VALIDATION;

SQL> SELECT test1.*, ORA_PARTITION_VALIDATION(rowid) FROM test1;

COL1       ORA_PARTITION_VALIDATION(ROWID)
---------- -------------------------------
         1                               1
        99                               0
```

> **See Also:**
>
> - *Oracle Database SQL Language Reference* for information about SQL functions

2.1.3 Partitioned Tables

Most tables can be partitioned.

Any table can be partitioned up to a million separate partitions except those tables containing columns with `LONG` or `LONG RAW` data types. You can, however, use tables containing columns with `CLOB` or `BLOB` data types.

The following topics are discussed:

- When to Partition a Table
- When to Partition an Index

> **Note:**
>
> To reduce disk and memory usage (specifically, the buffer cache), you can store tables and partitions of a partitioned table in a compressed format inside the database. This often improves scaleup for read-only operations. Table compression can also speed up query execution. There is, however, a slight cost in CPU overhead.

> **See Also:**
>
> *Oracle Database Administrator's Guide* for information about guidelines for managing tables

2.1.3.1 When to Partition a Table

There are certain situations when you would want to partition a table.

Here are some suggestions for situations when you should consider partitioning a table:

- Tables that are greater than 2 GB.

  These tables should always be considered as candidates for partitioning.
- Tables that contain historical data, in which new data is added into the newest partition.

  A typical example is a historical table where only the current month's data is updatable and the other 11 months are read only.
- Tables whose contents must be distributed across different types of storage devices.

2.1.3.2 When to Partition an Index

There are certain situations when you would want to partition an index.

Here are some suggestions for when to consider partitioning an index:

- Avoid index maintenance when data is removed.
- Perform maintenance on parts of the data without invalidating the entire index.
- Reduce the effect of index skew caused by an index on a column with a monotonically increasing value.

2.1.4 Partitioned Index-Organized Tables

Partitioned index-organized tables are very useful for providing improved performance, manageability, and availability for index-organized tables.

For partitioning an index-organized table:

- Partition columns must be a subset of the primary key columns.
- Secondary indexes can be partitioned (both locally and globally).
- `OVERFLOW` data segments are always equipartitioned with the table partitions.

> **See Also:**
>
> *Oracle Database Concepts* for more information about index-organized tables

2.1.5 System Partitioning

System partitioning enables application-controlled partitioning without having the database controlling the data placement.

The database simply provides the ability to break down a table into partitions without knowing what the individual partitions are going to be used for. All aspects of partitioning have to be controlled by the application. For example, an attempt to insert into a system partitioned table without the explicit specification of a partition fails.

System partitioning provides the well-known benefits of partitioning (scalability, availability, and manageability), but the partitioning and actual data placement are controlled by the application.

> **See Also:**
>
> *Oracle Database Data Cartridge Developer's Guide* for more information about system partitioning

2.1.6 Partitioning for Information Lifecycle Management

Information Lifecycle Management (ILM) is concerned with managing data during its lifetime.

Partitioning plays a key role in ILM because it enables groups of data (that is, partitions) to be distributed across different types of storage devices and managed individually.

> **See Also:**
>
> Managing and Maintaining Time-Based Information for more information about Information Lifecycle Management

2.1.7 Range Partitioning for Hash Clusters

Partitioned hash clusters are supported in Oracle Database.

Only single-level range partitioning is supported for partitioned hash clusters.

> **See Also:**
>
> *Oracle Database Reference* for information about partitioned hash clusters.

2.1.8 Partitioning and LOB Data

Unstructured data, such as images and documents, which is stored in a LOB column in a database can also be partitioned.

When a table is partitioned, all of the columns reside in the tablespace for that partition, except LOB columns, which can be stored in their own tablespace.

This technique is very useful when a table consists of large LOBs because they can be stored separately from the main data. This can be beneficial if the main data is being frequently updated but the LOB data is not. For example, an employee record may contain a photo which is unlikely to change frequently. However, the employee personnel details (such as address, department, manager, and so on) could change. This approach also means that you can use less expensive storage for storing the LOB data and more expensive, faster storage can be used for the employee record.

2.1.9 Partitioning on External Tables

Partitioning is supported on external tables.

This functionality enables optimizations, such as static partition pruning, dynamic pruning, and partition wise join for queries over partitioned external tables. This functionality also provides incremental, partition-based statistics collection for each external table partition, which enables better optimizer plans.

Tip:

For data sources in the Cloud, you can create external tables manually, but Oracle recommends that you use the `DBMS_CLOUD` package to do so.

Note that as of Oracle Database 23ai, you can use external table partitioning with folder names as part of the file paths. External table columns also can return the filename of the source file for each row.

See Also:

- *Oracle Database Utilities* for information about external tables.
  Also in *Oracle Database Utilities*, see *Using SQL*Loader for External Tables with Partition Values in File Paths*
- *Database PL/SQL Packages and Types Reference* for a complete description of the `DBMS_CLOUD` PL/SQL package.

2.1.10 Hybrid Partitioned Tables

Oracle hybrid partitioned tables combine classical internal partitioned tables with Oracle external partitioned tables to form a more general partitioning called hybrid partitioned tables.

Hybrid partitioned tables enable you to easily integrate internal partitions and external partitions (those residing on sources outside the database) into a single partition table. Using

this feature also enables you to easily move non-active partitions to external files for a cheaper storage solution.

Internal partitions of hybrid partitioned tables reside in Oracle tablespaces. The data of external partitions can reside in the Hadoop ecosystem as Oracle Data Pump, Parquet, ORC, and CSV files in HDFS, and also in HBase. External partition data can also reside in Oracle NoSQL Database, Apache Kafka, and in object stores in the Cloud such as Oracle's OCI Object Storage, Microsoft Azure, and Amazon S3.

Access to external partitions supports all existing table types and the following access drivers: `ORACLE_LOADER`, `ORACLE_DATAPUMP`, `ORACLE_HDFS`, `ORACLE_HIVE`, and `ORACLE_BIGDATA`.

For external partitions created with the `ORACLE_LOADER` and `ORACLE_DATAPUMP` access drivers, you must grant the following privileges to the user:

- READ privileges on directories with data files
- WRITE privileges on directories with logging and bad files
- EXECUTE privileges on directories with pre-processor programs

Table-level external parameters apply to all external partitions of hybrid partitioned tables. For example, the `DEFAULT DIRECTORY` value defined in the `EXTERNAL PARTITION ATTRIBUTES` clause is the default location for data files and logging and bad files. You can override the default directory location with a `DEFAULT DIRECTORY` value in a partition clause. For external partitions of `ORACLE_HIVE` and `ORACLE_HDFS` access driver type, the `DEFAULT DIRECTORY` is only used to store specifications for log files.

Enforcement of constraints is not supported on data stored in external partitions because the constraints apply to the entire table. For example, primary or foreign key constraints cannot be enforced on a hybrid partitioned table. These particular constraints can be used in all modes other than `QUERY_REWRITE_INTEGRITY=ENFORCED`, in which the database only uses data whose integrity it can guarantee. Data in external partitions cannot be guaranteed. Only constraints in the `RELY DISABLE` mode, such as `NOT NULL`, primary key, unique, and foreign-primary key are supported on hybrid partitioned tables. To activate optimizations based on these constraints, set the session parameter `QUERY_REWRITE_INTEGRITY` to `TRUSTED` or `STALE_TOLERATED`.

Hybrid partitioned tables can use partition-based optimizations across internal and external partitions. Partition-based optimizations include the following across internal and external data sources:

- Static partition pruning
- Dynamic partition pruning
- Bloom pruning

Hybrid partitioned tables provide users with the capability to move data between internal and external partitions for cost effective purposes. However, Automatic Data Optimization (ADO) defined on the table level only has an effect on internal partitions of the table.

Supported Operations on Hybrid Partitioned Tables

The following are operations supported on hybrid partitioned tables.

- Creating single level `RANGE` and `LIST` partitioning methods
- Using `ALTER TABLE` and DDLs such as `ADD`, `DROP`, and `RENAME` partitions
- Modifying for external partitions the location of the external data sources at the partition level

- Altering an existing partitioned internal table to a hybrid partitioned table containing both internal and external partitions
- Changing the existing location to an empty location resulting in an empty external partition
- Creating global partial non-unique indexes on internal partitions
- Creating materialized views on internal partitions
- Creating materialized views that include `AUTOLIST` or `INTERVAL` partitioning.
- Creating materialized views that include external partitions in `QUERY_REWRITE_INTEGRITY` stale tolerated mode only
- Full partition wise refreshing on external partitions
- DML trigger operations on a hybrid partitioned table on internal partitions
- Validating with `ANALYZE TABLE ... VALIDATE STRUCTURE` on internal partitions only on hybrid partitioned tables
- Altering an existing hybrid partitioned table with no external partitions to a partitioned table with internal partitions only
- An external partition can be exchanged with an external nonpartitioned table. Also, an internal partition can be exchanged with an internal nonpartitioned table. In addition, `EXCHANGE PARTITION` supports the exchange from internal partitions with external tables and from external partitions with internal tables for hybrid partitioned tables

Restrictions on Hybrid Partitioned Tables

The following are restrictions and limitations on hybrid partitioned tables.

- Restrictions that apply to external tables also apply to hybrid partitioned tables unless explicitly noted
- No unique indexes or global unique indexes. Only partial indexes are allowed and unique indexes cannot be partial.
- Only single level list partitioning is supported for `HIVE`.
- Attribute clustering (`CLUSTERING` clause) is not allowed.
- DML operations only on internal partitions of a hybrid partitioned table (external partitions are treated as read-only partitions)
- In-memory defined on the table level only has an effect on internal partitions of the hybrid partitioned table.
- No column default value
- Invisible columns are not allowed.
- The `CELLMEMORY` clause is not allowed.
- `SPLIT`, `MERGE`, and `MOVE` maintenance operations are not allowed on external partitions.
- LOB, LONG, and ADT types are not allowed.
- Only `RELY` constraints are allowed

See Also:

- Managing Hybrid Partitioned Tables for information about administering hybrid partitioned tables
- *Oracle Database Administrator's Guide* for information about hybrid partitioned external tables
- *Oracle Database Concepts* for conceptual information about partitioned tables
- *Oracle Database In-Memory Guide* for information about the In-Memory Column Store and hybrid partition tables
- *Oracle Database SQL Tuning Guide* for information about optimizations for hybrid partitioned tables
- *Oracle Database SQL Language Reference* for information about creating and altering hybrid partitioned tables using the `CREATE TABLE` and `ALTER TABLE` SQL commands
- *Oracle Database Utilities* for information about using SQL*Loader with hybrid partitioned tables, using Oracle Data Pump with hybrid partitioned tables, and managing external tables
- *Oracle Database PL/SQL Packages and Types Reference* for information about the using PL/SQL with hybrid partitioned tables, including the `CREATE_HYBRID_PARTNED_TABLE` procedure in the DBMS_HADOOP package
- *Oracle Database Reference* for information about hybrid partition tables in data dictionary views, including the external family of data dictionary views and *_TABLES views
- *Oracle Database Data Warehousing Guide* for information about materialized views and hybrid partitioned tables

2.1.11 Collections in XMLType and Object Data

Partitioning when using `XMLType` and object tables and columns offers the standard advantages of partitioning, such as enabling tables and indexes to be subdivided into smaller pieces, thus enabling these database objects to be managed and accessed at a finer level of granularity.

When you partition an `XMLType` table or a table with an `XMLType` column using list, range, or hash partitioning, any ordered collection tables (OCTs) within the data are automatically partitioned accordingly, by default. This equipartitioning means that the partitioning of an OCT follows the partitioning scheme of its parent (base) table. There is a corresponding collection-table partition for each partition of the base table. A child element is stored in the collection-table partition that corresponds to the base-table partition of its parent element.

If you partition a table that has a nested table, then Oracle Database uses the partitioning scheme of the original base table as the basis for how the nested table is partitioned. This partitioning of one base table partition for each nested table partition is called equipartitioning. By default, nested tables are automatically partitioned when the base table is partitioned. Note, however, that composite partitioning is not supported for OCTs or nested tables.

See Also:

- Partitioning of Collections in XMLType and Objects for information about partitioning an `XMLType` table
- *Oracle Database SQL Language Reference* for syntax of nested tables
- *Oracle Database Object-Relational Developer's Guide*

2.2 Benefits of Partitioning

Partitioning can provide tremendous benefit to a wide variety of applications by improving performance, manageability, and availability.

It is not unusual for partitioning to greatly improve the performance of certain queries or maintenance operations. Moreover, partitioning can greatly simplify common administration tasks.

Partitioning also enables database designers and administrators to solve some difficult problems posed by cutting-edge applications. Partitioning is a key tool for building multi-terabyte systems or systems with extremely high availability requirements.

The following topics are discussed:

- Partitioning for Performance
- Partitioning for Manageability
- Partitioning for Availability

2.2.1 Partitioning for Performance

You can use partitioning to improve performance.

By limiting the amount of data to be examined or operated on, and by providing data distribution for parallel execution, partitioning provides multiple performance benefits. Partitioning features include:

- Partition Pruning for Performance
- Partition-Wise Joins for Performance

2.2.1.1 Partition Pruning for Performance

Partition pruning is the simplest and also the most substantial means to improve performance using partitioning.

Partition pruning can often improve query performance by several orders of magnitude. For example, suppose an application contains an `Orders` table containing a historical record of orders, and that this table has been partitioned by week. A query requesting orders for a single week would only access a single partition of the `Orders` table. If the `Orders` table had 2 years of historical data, then this query would access one partition instead of 104 partitions. This query could potentially execute 100 times faster simply because of partition pruning.

Partition pruning works with all of Oracle performance features. Oracle uses partition pruning with any indexing or join technique, or parallel access method.

2.2.1.2 Partition-Wise Joins for Performance

Partitioning can also improve the performance of multi-table joins by using a technique known as partition-wise joins.

Partition-wise joins can be applied when two tables are being joined and both tables are partitioned on the join key, or when a reference partitioned table is joined with its parent table. Partition-wise joins break a large join into smaller joins that occur between each of the partitions, completing the overall join in less time. This offers significant performance benefits both for serial and parallel execution.

2.2.2 Partitioning for Manageability

Partitioning enables you to partition tables and indexes into smaller, more manageable units, providing database administrators with the ability to pursue a *divide and conquer* approach to data management.

With partitioning, maintenance operations can be focused on particular portions of tables. For example, you could back up a single partition of a table, rather than back up the entire table. For maintenance operations across an entire database object, it is possible to perform these operations on a per-partition basis, thus dividing the maintenance process into more manageable chunks.

A typical usage of partitioning for manageability is to support a *rolling window* load process in a data warehouse. Suppose that you load new data into a table on a weekly basis. That table could be partitioned so that each partition contains one week of data. The load process is simply the addition of a new partition using a partition exchange load. Adding a single partition is much more efficient than modifying the entire table, because you do not need to modify any other partitions.

2.2.3 Partitioning for Availability

Partitioned database objects provide partition independence. This characteristic of partition independence can be an important part of a high-availability strategy.

For example, if one partition of a partitioned table is unavailable, then all of the other partitions of the table remain online and available. The application can continue to execute queries and transactions against the available partitions for the table, and these database operations can run successfully, provided they do not need to access the unavailable partition.

The database administrator can specify that each partition be stored in a separate tablespace; the most common scenario is having these tablespaces stored on different storage tiers. Storing different partitions in different tablespaces enables you to do backup and recovery operations on each individual partition, independent of the other partitions in the table. Thus allowing the active parts of the database to be made available sooner so access to the system can continue, while the inactive data is still being restored. Moreover, partitioning can reduce scheduled downtime. The performance gains provided by partitioning may enable you to complete maintenance operations on large database objects in relatively small batch windows.

2.3 Partitioning Strategies

Oracle Partitioning offers three fundamental data distribution methods as basic partitioning strategies that control how data is placed into individual partitions.

These strategies are:

- Range
- Hash
- List

Using these data distribution methods, a table can either be partitioned as a single-level or as a composite-partitioned table:

- Single-Level Partitioning
- Composite Partitioning

Each partitioning strategy has different advantages and design considerations. Thus, each strategy is more appropriate for a particular situation.

2.3.1 Single-Level Partitioning

Single-level partitioning includes range, hash, and list partitioning.

A table is defined by specifying one of the following data distribution methodologies, using one or more columns as the partitioning key:

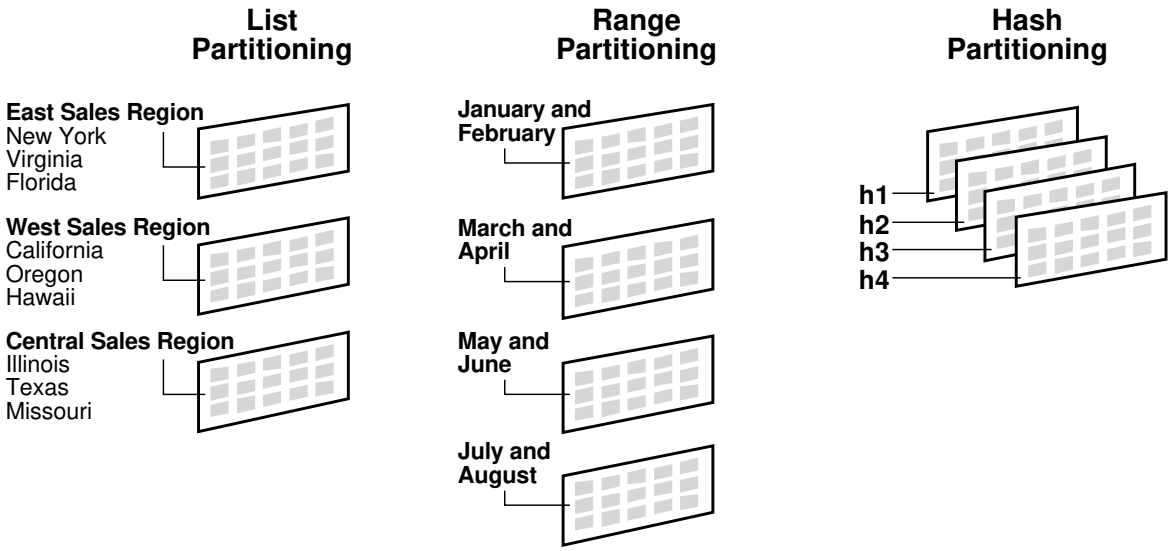
- Range Partitioning
- Hash Partitioning
- List Partitioning

For example, consider a table with a column of type `NUMBER` as the partitioning key and two partitions **less_than_five_hundred** and **less_than_one_thousand**. The **less_than_one_thousand** partition contains rows where the following condition is true:

```
500 <= partitioning key < 1000
```

Figure 2-2 offers a graphical view of the basic partitioning strategies for a single-level partitioned table.

Figure 2-2 List, Range, and Hash Partitioning



2.3.1.1 Range Partitioning

Range partitioning maps data to partitions based on ranges of values of the partitioning key that you establish for each partition.

Range partitioning is the most common type of partitioning and is often used with dates. For a table with a date column as the partitioning key, the **January-2017** partition would contain rows with partitioning key values from **01-Jan-2017** to **31-Jan-2017**.

Each partition has a `VALUES LESS THAN` clause, that specifies a non-inclusive upper bound for the partitions. Any values of the partitioning key equal to or higher than this literal are added to the next higher partition. All partitions, except the first, have an implicit lower bound specified by the `VALUES LESS THAN` clause of the previous partition.

A `MAXVALUE` literal can be defined for the highest partition. `MAXVALUE` represents a virtual infinite value that sorts higher than any other possible value for the partitioning key, including the NULL value.

2.3.1.2 Hash Partitioning

Hash partitioning maps data to partitions based on a hashing algorithm that Oracle applies to the partitioning key that you identify.

The hashing algorithm evenly distributes rows among partitions, giving partitions approximately the same size.

Hash partitioning is the ideal method for distributing data evenly across devices. Hash partitioning is also an easy-to-use alternative to range partitioning, especially when the data to be partitioned is not historical or has no obvious partitioning key.

> **Note:**
>
> You cannot change the hashing algorithms used by partitioning.

2.3.1.3 List Partitioning

List partitioning enables you to explicitly control how rows map to partitions by specifying a list of discrete values for the partitioning key in the description for each partition.

The advantage of list partitioning is that you can group and organize unordered and unrelated sets of data in a natural way. For a table with a region column as the partitioning key, the **East Sales Region** partition might contain values **New York**, **Virginia**, and **Florida**.

The `DEFAULT` partition enables you to avoid specifying all possible values for a list-partitioned table by using a default partition, so that all rows that do not map to any other partition do not generate an error.

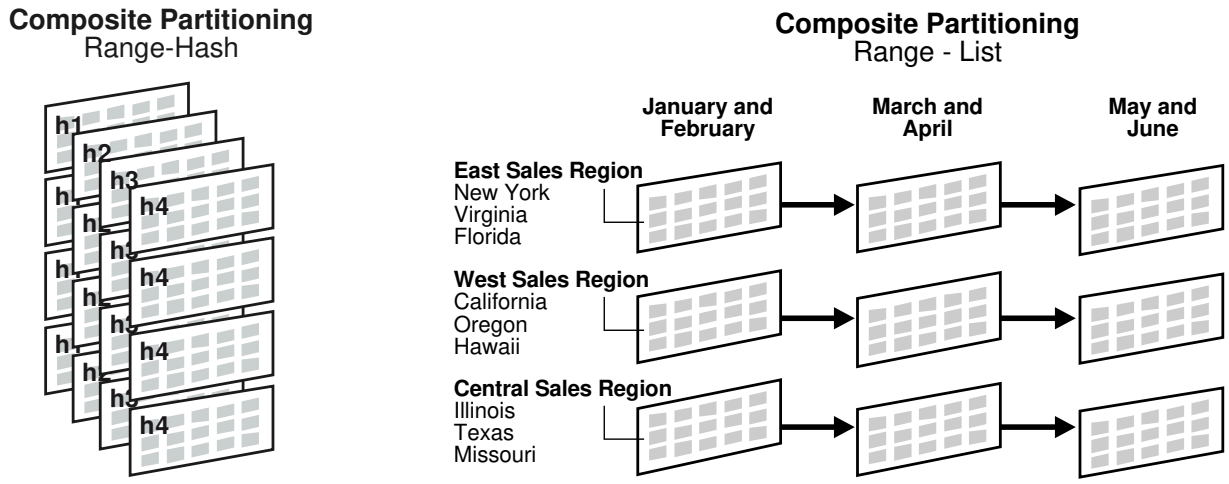
2.3.2 Composite Partitioning

Composite partitioning is a combination of the basic data distribution methods.

With composite partitioning, a table is partitioned by one data distribution method and then each partition is further subdivided into subpartitions using a second data distribution method. All subpartitions for a given partition represent a logical subset of the data.

Composite partitioning supports historical operations, such as adding new range partitions, but also provides higher degrees of potential partition pruning and finer granularity of data placement through subpartitioning. Figure 2-3 offers a graphical view of range-hash and range-list composite partitioning, as an example.

Figure 2-3 Composite Range—List Partitioning



The types of composite partitioning are:

- Composite Range-Range Partitioning
- Composite Range-Hash Partitioning
- Composite Range-List Partitioning
- Composite List-Range Partitioning
- Composite List-Hash Partitioning
- Composite List-List Partitioning
- Composite Hash-Hash Partitioning
- Composite Hash-List Partitioning
- Composite Hash-Range Partitioning

2.3.2.1 Composite Range-Range Partitioning

Composite range-range partitioning enables logical range partitioning along two dimensions.

An example of composite range-range partitioning is partition by `order_date` and range subpartition by `shipping_date`.

2.3.2.2 Composite Range-Hash Partitioning

Composite range-hash partitioning partitions data using the range method, and within each partition, subpartitions it using the hash method.

Composite range-hash partitioning provides the improved manageability of range partitioning and the data placement, striping, and parallelism advantages of hash partitioning.

2.3.2.3 Composite Range-List Partitioning

Composite range-list partitioning partitions data using the range method, and within each partition, subpartitions it using the list method.

Composite range-list partitioning provides the manageability of range partitioning and the explicit control of list partitioning for the subpartitions.

2.3.2.4 Composite List-Range Partitioning

Composite list-range partitioning enables logical range subpartitioning within a given list partitioning strategy.

An example of composite list-range partitioning is list partition by `country_id` and range subpartition by `order_date`.

2.3.2.5 Composite List-Hash Partitioning

Composite list-hash partitioning enables hash subpartitioning of a list-partitioned object.

A composite list-hash partitioning, is useful to enable partition-wise joins.

2.3.2.6 Composite List-List Partitioning

Composite list-list partitioning enables logical list partitioning along two dimensions.

An example of composite list-list partitioning is list partition by `country_id` and list subpartition by `sales_channel`.

2.3.2.7 Composite Hash-Hash Partitioning

Composite hash-hash partitioning enables hash partitioning along two dimensions.

The composite hash-hash partitioning technique is beneficial to enable partition-wise joins along two dimensions.

2.3.2.8 Composite Hash-List Partitioning

Composite hash-list partitioning is introduced in this topic.

Composite hash-list partitioning enables hash partitioning along two dimensions.

2.3.2.9 Composite Hash-Range Partitioning

Composite hash-range partitioning is introduced in this topic.

Composite hash-range partitioning enables hash partitioning along two dimensions.

2.4 Partitioning Extensions

In addition to the basic partitioning strategies, Oracle Database provides partitioning extensions.

Oracle Database provides the following types of partitioning extensions:

- Manageability Extensions
- Partitioning Key Extensions

2.4.1 Manageability Extensions

Manageability extensions for partitioning are introduced in this topic.

The following extensions significantly enhance the manageability of partitioned tables:

- Interval Partitioning
- Partition Advisor

2.4.1.1 Interval Partitioning

Interval partitioning is an extension of range partitioning .

Interval partitioning instructs the database to automatically create partitions of a specified interval when data inserted into the table exceeds all of the existing range partitions. You must specify at least one range partition. The range partitioning key value determines the high value of the range partitions, which is called the transition point, and the database creates interval partitions for data with values that are beyond that transition point. The lower boundary of every interval partition is the non-inclusive upper boundary of the previous range or interval partition.

For example, if you create an interval partitioned table with monthly intervals and you set the transition point at January 1, 2007, then the lower boundary for the January 2007 interval is January 1, 2007. The lower boundary for the July 2007 interval is July 1, 2007, regardless of whether the June 2007 partition was created.

You can create single-level interval partitioned tables and the following composite partitioned tables:

- Interval-range
- Interval-hash
- Interval-list

Interval partitioning supports a subset of the capabilities of range partitioning.

> **See Also:**
>
> *Oracle Database SQL Language Reference* for information about restrictions when using interval partitioning

2.4.1.2 Partition Advisor

The Partition Advisor is part of the SQL Access Advisor.

The Partition Advisor can recommend a partitioning strategy for a table based on a supplied workload of SQL statements which can be supplied by the SQL Cache, a SQL Tuning set, or be defined by the user.

2.4.2 Partitioning Key Extensions

Extensions to partitioning keys are introduced in this topic.

The following extensions extend the flexibility in defining partitioning keys:

- Reference Partitioning
- Virtual Column-Based Partitioning

2.4.2.1 Reference Partitioning

Reference partitioning enables the partitioning of two tables that are related to one another by referential constraints.

The partitioning key is resolved through an existing parent-child relationship, enforced by enabled and active primary key and foreign key constraints.

The benefit of this extension is that tables with a parent-child relationship can be logically equipartitioned by inheriting the partitioning key from the parent table without duplicating the key columns. The logical dependency also automatically cascades partition maintenance operations, thus making application development easier and less error-prone.

An example of reference partitioning is the `Orders` and `LineItems` tables related to each other by a referential constraint `orderid_refconstraint`. Namely, `LineItems.order_id` references `Orders.order_id`. The `Orders` table is range partitioned on `order_date`. Reference partitioning on `orderid_refconstraint` for `LineItems` leads to creation of the following partitioned table, which is equipartitioned on the `Orders` table, as shown in Figure 2-4 and Figure 2-5.

Figure 2-4 Before Reference Partitioning

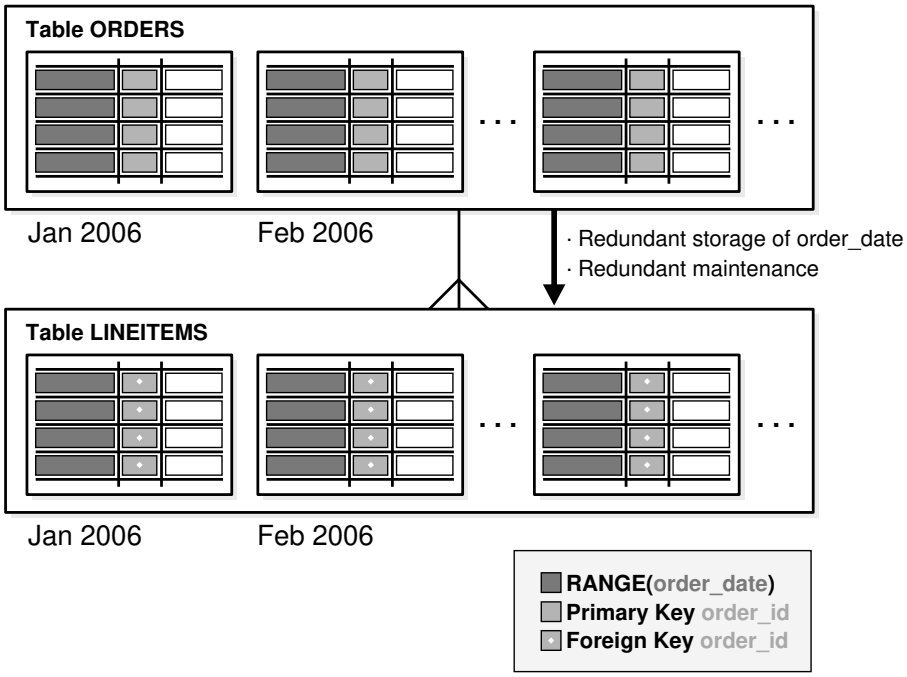
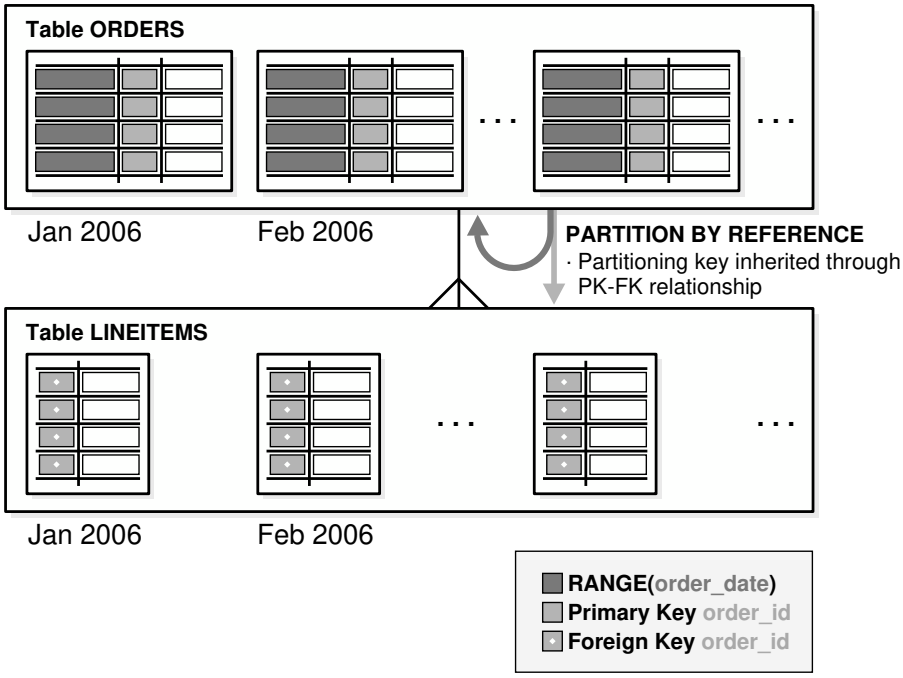


Figure 2-5 With Reference Partitioning



All basic partitioning strategies are available for reference partitioning. Interval partitioning can also be used with reference partitioning.

> **Note:**
>
> Reference partitioning is not supported with the online redefinition package (`DBMS_REDEFINITION`).

2.4.2.2 Virtual Column-Based Partitioning

Oracle partitioning includes a partitioning strategy defined on virtual columns.

Virtual columns enable the partitioning key to be defined by an expression, using one or more existing columns of a table. The expression is stored as metadata only. For example, a ten-digit account ID can include account branch information as the leading three digits. With the extension of virtual column based partitioning, an `ACCOUNTS` table containing an `ACCOUNT_ID` column can be extended with a virtual (derived) column `ACCOUNT_BRANCH`. `ACCOUNT_BRANCH` is derived from the first three digits of the `ACCOUNT_ID` column, which becomes the partitioning key for this table.

Virtual column-based partitioning is supported with all basic partitioning strategies, including reference partitioning, and interval and interval-* composite partitioning.

2.5 Indexing on Partitioned Tables

Indexes on partitioned tables can either be nonpartitioned or partitioned.

As with partitioned tables, partitioned indexes improve manageability, availability, performance, and scalability. They can either be partitioned independently (global indexes) or automatically linked to a table's partitioning method (local indexes). In general, you should use global indexes for OLTP applications and local indexes for data warehousing or decision support systems (DSS) applications.

The following topics are discussed:

- Deciding on the Type of Partitioned Index to Use
- Local Partitioned Indexes
- Global Partitioned Indexes
- Global Nonpartitioned Indexes
- Miscellaneous Information about Creating Indexes on Partitioned Tables
- Partial Indexes for Partitioned Tables
- Partitioned Indexes on Composite Partitions

2.5.1 Deciding on the Type of Partitioned Index to Use

The type of partitioned index to use should be chosen after reviewing various factors.

When deciding what kind of partitioned index to use, you should consider the following guidelines in this order:

1. If the table partitioning column is a subset of the index keys, then use a local index. If this is the case, then you are finished. If this is not the case, then continue to guideline 2.
2. If the index is unique and does not include the partitioning key columns, then use a global index. If this is the case, then you are finished. Otherwise, continue to guideline 3.
3. If your priority is manageability, then consider a local index. If this is the case, then you are finished. If this is not the case, continue to guideline 4.
4. If the application is an OLTP type and users need quick response times, then use a global index. If the application is a DSS type and users are more interested in throughput, then use a local index.

> **See Also:**
>
> - Using Partitioning in a Data Warehouse Environment for information about partitioned indexes and how to decide which type to use in data warehouse environment
> - Using Partitioning in an Online Transaction Processing Environment for information about partitioned indexes and how to decide which type to use in an online transaction processing environment

2.5.2 Local Partitioned Indexes

Local partitioned indexes are easier to manage than other types of partitioned indexes.

They also offer greater availability and are common in DSS environments. The reason for this is equipartitioning: each partition of a local index is associated with exactly one partition of the table. This functionality enables Oracle to automatically keep the index partitions synchronized

with the table partitions, and makes each table-index pair independent. Any actions that make one partition's data invalid or unavailable only affect a single partition.

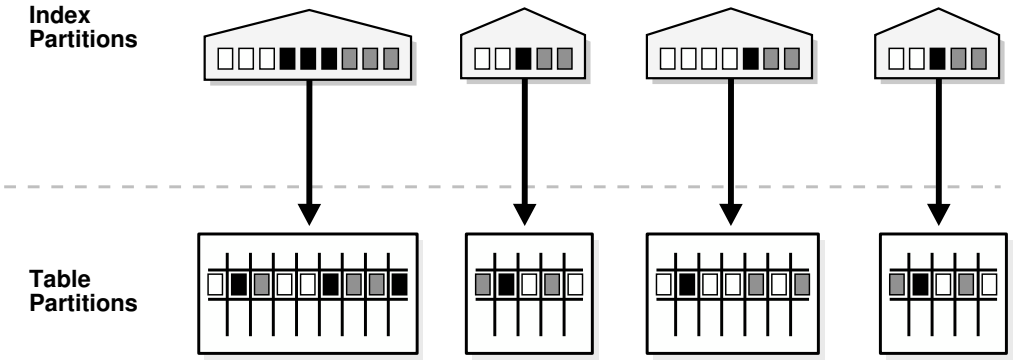
Local partitioned indexes support more availability when there are partition or subpartition maintenance operations on the table. A type of index called a local nonprefixed index is very useful for historical databases. In this type of index, the partitioning is not on the left prefix of the index columns.

You cannot explicitly add a partition to a local index. Instead, new partitions are added to local indexes only when you add a partition to the underlying table. Likewise, you cannot explicitly drop a partition from a local index. Instead, local index partitions are dropped only when you drop a partition from the underlying table.

A local index can be unique. However, in order for a local index to be unique, the partitioning key of the table must be part of the index's key columns.

Figure 2-6 offers a graphical view of local partitioned indexes.

Figure 2-6 Local Partitioned Index



> **See Also:**
>
> - Index Partitioning for more information about prefixed indexes
> - Local Partitioned Indexes for more information about local partitioned indexes

2.5.3 Global Partitioned Indexes

Global partitioned indexes are introduced in the topic.

Oracle offers global range partitioned indexes and global hash partitioned indexes, discussed in the following topics:

- Global Range Partitioned Indexes
- Global Hash Partitioned Indexes
- Maintenance of Global Partitioned Indexes

2.5.3.1 Global Range Partitioned Indexes

Global range partitioned indexes are flexible in that the degree of partitioning and the partitioning key are independent from the table's partitioning method.

The highest partition of a global index must have a partition bound, all of whose values are `MAXVALUE`. This ensures that all rows in the underlying table can be represented in the index. Global prefixed indexes can be unique or nonunique.

You cannot add a partition to a global index because the highest partition always has a partition bound of `MAXVALUE`. To add a new highest partition, use the `ALTER INDEX SPLIT PARTITION` statement. If a global index partition is empty, you can explicitly drop it by issuing the `ALTER INDEX DROP PARTITION` statement. If a global index partition contains data, dropping the partition causes the next highest partition to be marked unusable. You cannot drop the highest partition in a global index.

2.5.3.2 Global Hash Partitioned Indexes

Global hash partitioned indexes improve performance by spreading out contention when the index is monotonically growing.

In other words, most of the index insertions occur only on the right edge of an index, which is uniformly spread across N hash partitions for a global hash partitioned index.

2.5.3.3 Maintenance of Global Partitioned Indexes

The maintenance of global partitioned indexes is introduced in this topic.

By default, the following operations on partitions on a heap-organized table mark all global indexes as unusable:

```
ADD (HASH)
COALESCE (HASH)
DROP
EXCHANGE
MERGE
MOVE
SPLIT
TRUNCATE
```

These indexes can be maintained by appending the clause `UPDATE INDEXES` to the SQL statements for the operation. Note, however, that appending the `UPDATE INDEXES` clause maintains the global index as part of the partition maintenance operation, potentially elongating the run time of the operation and increasing the resource requirements.

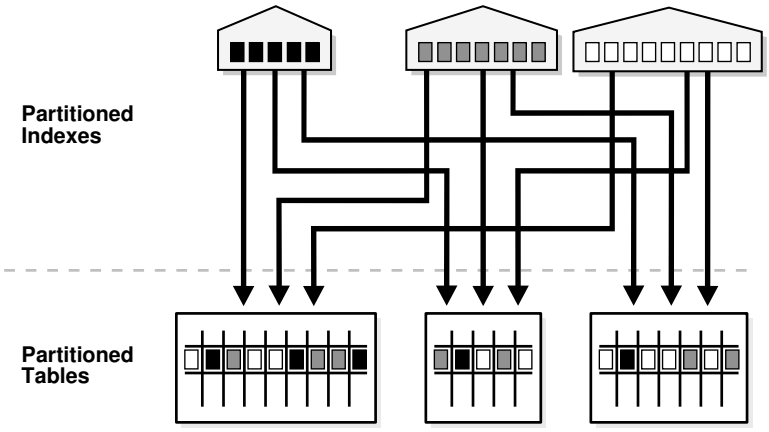
The two advantages to maintaining global indexes are:

- The index remains available and online throughout the operation. Hence no other applications are affected by this operation.
- The index does not have to be rebuilt after the operation.
- The global index maintenance for `DROP` and `TRUNCATE` is implemented as metadata-only operation.

> **Note:**
>
> This feature is supported only for heap-organized tables.

Figure 2-7 offers a graphical view of global partitioned indexes.

Figure 2-7 Global Partitioned Index



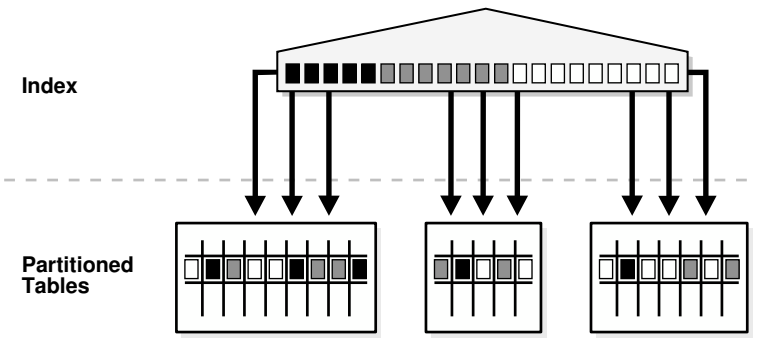
> **See Also:**
>
> Global Partitioned Indexes for more information about global partitioned indexes

2.5.4 Global Nonpartitioned Indexes

Global nonpartitioned indexes behave just like local nonpartitioned indexes.

Figure 2-8 offers a graphical view of global nonpartitioned indexes.

Figure 2-8 Global Nonpartitioned Index



2.5.5 Miscellaneous Information about Creating Indexes on Partitioned Tables

You can create bitmap indexes on partitioned tables, with some restrictions.

Bitmap indexes must be local to the partitioned table. They cannot be global indexes.

Global indexes can be unique. Local indexes can only be unique if the partitioning key is a part of the index key.

2.5.6 Partial Indexes for Partitioned Tables

You can create local and global indexes on a subset of the partitions of a table, enabling more flexibility in index creation.

This feature is supported using a default table indexing property. When a table is created or altered, a default indexing property can be specified for the table or its partitions. The table indexing property is only considered for partial indexes.

When an index is created as `PARTIAL` on a table:

- Local indexes: An index partition is created usable if indexing is turned on for the table partition, and unusable otherwise. You can override this behavior by specifying `USABLE`/ `UNUSABLE` at the index or index partition level.
- Global indexes: Includes only those partitions for which indexing is turned on, and exclude the others.

This feature is not supported for unique indexes, or for indexes used for enforcing unique constraints. `FULL` is the default if neither `FULL` nor `PARTIAL` is specified.

By default, any index is created as `FULL` index, which decouples the index from the table indexing property.

The `INDEXING` clause may also be specified at the partition and subpartition levels.

The following SQL DDL creates a table with these items:

- Partitions `ORD_P1` and `ORD_P3` are included in all partial global indexes
- Local index partitions (for indexes created `PARTIAL`) corresponding to the above two table partitions are created usable by default.
- Other partitions are excluded from all partial global indexes, and created unusable in local indexes (for indexes created `PARTIAL`).

```
CREATE TABLE orders (
  order_id NUMBER(12),
  order_date DATE CONSTRAINT order_date_nn NOT NULL,
  order_mode VARCHAR2(8),
  customer_id NUMBER(6) CONSTRAINT order_customer_id_nn NOT NULL,
  order_status NUMBER(2),
  order_total NUMBER(8,2),
  sales_rep_id NUMBER(6),
  promotion_id NUMBER(6),
  CONSTRAINT order_mode_lov CHECK (order_mode in ('direct','online')),
  CONSTRAINT order_total_min CHECK (order_total >= 0))
   INDEXING OFF
   PARTITION BY RANGE (ORDER_DATE)
   (PARTITION ord_p1 VALUES LESS THAN (TO_DATE('01-MAR-1999','DD-MON-YYYY'))
```

```
  INDEXING ON,
PARTITION ord_p2 VALUES LESS THAN (TO_DATE('01-JUL-1999','DD-MON-YYYY'))
  INDEXING OFF,
PARTITION ord_p3 VALUES LESS THAN (TO_DATE('01-OCT-1999','DD-MON-YYYY'))
  INDEXING ON,
PARTITION ord_p4 VALUES LESS THAN (TO_DATE('01-MAR-2000','DD-MON-YYYY')),
PARTITION ord_p5 VALUES LESS THAN (TO_DATE('01-MAR-2010','DD-MON-YYYY')));
```

A local or global partial index, can be created to follow the table indexing properties of the previous SQL example by specification of the `INDEXING PARTIAL` clause.

```
CREATE INDEX ORDERS_ORDER_TOTAL_GIDX ON ORDERS (ORDER_TOTAL)
   GLOBAL INDEXING PARTIAL;
```

The `ORDERS_ORDER_TOTAL_GIDX` index is created to index only those partitions that have `INDEXING ON`, and excludes the remaining partitions.

Updates to views include the following:

- Table Indexing Property - The column `INDEXING` is added to `*_PART_TABLES`, `*_TAB_PARTITIONS`, and `*_TAB_SUBPARTITIONS` views.

  This column has one of two values `ON` or `OFF`, specifying indexing on or indexing off.
- Partial Global Indexes as an Index Level Property - A new column `INDEXING` is added to the `USER_INDEXES` view. This column can be set to `FULL` or `PARTIAL`.
- Partial Global Index Optimization - The column `ORPHANED_ENTRIES` is added to the dictionary views `USER_INDEXES` and `USER_IND_PARTITIONS` to represent if a global index (partition) contains stale entries owing to deferred index maintenance during `DROP`/ `TRUNCATE PARTITION`, or `MODIFY PARTITION INDEXING OFF`. The column can have one of the following values:
  - `YES` => the index (partition) contains orphaned entries
  - `NO` => the index (partition) does not contain any orphaned entries

> **See Also:**
>
> *Oracle Database Reference* for information about the database views

2.5.7 Partitioned Indexes on Composite Partitions

There a few items to consider when partitioned indexes on composite partitions

When using partitioned indexes on composite partitions, note the following:

- Subpartitioned indexes are always local and stored with the table subpartition by default.
- Tablespaces can be specified at either index or index subpartition levels.

3

Partitioning for Availability, Manageability, and Performance

Partitioning enables availability, manageability, and performance.

This chapter provides high-level insight into how partitioning enables availability, manageability, and performance. Guidelines are provided on when to use a given partitioning strategy. The main focus is the use of table partitioning, although most of the recommendations and considerations apply to index partitioning as well.

This chapter contains the following sections:

- Partition Pruning
- Partition-Wise Operations
- Index Partitioning
- Partitioning and Table Compression
- Recommendations for Choosing a Partitioning Strategy

3.1 Partition Pruning

Partition pruning is an essential performance feature for data warehouses.

In partition pruning, the optimizer analyzes `FROM` and `WHERE` clauses in SQL statements to eliminate unneeded partitions when building the partition access list. This functionality enables Oracle Database to perform operations only on those partitions that are relevant to the SQL statement.

The following topics are discussed:

- Benefits of Partition Pruning
- Information That Can Be Used for Partition Pruning
- How to Identify Whether Partition Pruning Has Been Used
- Static Partition Pruning
- Dynamic Partition Pruning
- Partition Pruning with Zone Maps
- Partition Pruning Tips

3.1.1 Benefits of Partition Pruning

Partition pruning dramatically reduces the amount of data retrieved from disk and shortens processing time, thus improving query performance and optimizing resource utilization.

If you partition the index and table on different columns (with a global partitioned index), then partition pruning also eliminates index partitions even when the partitions of the underlying table cannot be eliminated.

Depending upon the actual SQL statement, Oracle Database may use static or dynamic pruning. Static pruning occurs at compile-time, with the information about the partitions accessed beforehand. Dynamic pruning occurs at run-time, meaning that the exact partitions to be accessed by a statement are not known beforehand. A sample scenario for static pruning is a SQL statement containing a `WHERE` condition with a constant literal on the partition key column. An example of dynamic pruning is the use of operators or functions in the `WHERE` condition.

Partition pruning affects the statistics of the objects where pruning occurs and also affects the execution plan of a statement.

3.1.2 Information That Can Be Used for Partition Pruning

Partition pruning can be performed on partitioning columns.

Oracle Database prunes partitions when you use range, `LIKE`, equality, and `IN`-list predicates on the range or list partitioning columns, and when you use equality and `IN`-list predicates on the hash partitioning columns.

On composite partitioned objects, Oracle Database can prune at both levels using the relevant predicates. For example, see the table `sales_range_hash`, which is partitioned by range on the column `s_saledate` and subpartitioned by hash on the column `s_productid` in Example 3-1.

Oracle uses the predicate on the partitioning columns to perform partition pruning as follows:

- When using range partitioning, Oracle accesses only partitions `sal99q2` and `sal99q3`, representing the partitions for the third and fourth quarters of 1999.
- When using hash subpartitioning, Oracle accesses only the one subpartition in each partition that stores the rows with `s_productid=1200`. The mapping between the subpartition and the predicate is calculated based on Oracle's internal hash distribution function.

A reference-partitioned table can take advantage of partition pruning through the join with the referenced table. Virtual column-based partitioned tables benefit from partition pruning for statements that use the virtual column-defining expression in the SQL statement.

Example 3-1 Creating a table with partition pruning

```
CREATE TABLE sales_range_hash(
  s_productid  NUMBER,
  s_saledate   DATE,
  s_custid     NUMBER,
  s_totalprice NUMBER)
PARTITION BY RANGE (s_saledate)
SUBPARTITION BY HASH (s_productid) SUBPARTITIONS 8
 (PARTITION sal99q1 VALUES LESS THAN
   (TO_DATE('01-APR-1999', 'DD-MON-YYYY')),
  PARTITION sal99q2 VALUES LESS THAN
   (TO_DATE('01-JUL-1999', 'DD-MON-YYYY')),
  PARTITION sal99q3 VALUES LESS THAN
   (TO_DATE('01-OCT-1999', 'DD-MON-YYYY')),
  PARTITION sal99q4 VALUES LESS THAN
   (TO_DATE('01-JAN-2000', 'DD-MON-YYYY')));

SELECT * FROM sales_range_hash
WHERE s_saledate BETWEEN (TO_DATE('01-JUL-1999', 'DD-MON-YYYY'))
  AND (TO_DATE('01-OCT-1999', 'DD-MON-YYYY')) AND s_productid = 1200;
```

3.1.3 How to Identify Whether Partition Pruning Has Been Used

Whether Oracle uses partition pruning is reflected in the execution plan of a statement, either in the plan table for the `EXPLAIN PLAN` statement or in the shared SQL area.

The partition pruning information is reflected in the plan columns `PSTART` (`PARTITION_START`) and `PSTOP` (`PARTITION_STOP`). For serial statements, the pruning information is also reflected in the `OPERATION` and `OPTIONS` columns.

> **See Also:**
>
> *Oracle Database SQL Tuning Guide* for more information about `EXPLAIN PLAN` and how to interpret it

3.1.4 Static Partition Pruning

Oracle determines when to use static pruning primarily based on static predicates.

For many cases, Oracle determines the partitions to be accessed at compile time. Static partition pruning occurs if you use static predicates, except for the following cases:

- Partition pruning occurs using the result of a subquery.
- The optimizer rewrites the query with a star transformation and pruning occurs after the star transformation.
- The most efficient execution plan is a nested loop.

These three cases result in the use of dynamic pruning.

If at parse time Oracle can identify which contiguous set of partitions is accessed, then the `PSTART` and `PSTOP` columns in the execution plan show the begin and the end values of the partitions being accessed. Any other cases of partition pruning, including dynamic pruning, show the KEY value in `PSTART` and `PSTOP`, optionally with an additional attribute.

The following is an example:

```
SQL> explain plan for select * from sales where time_id = to_date('01-jan-2001', 'dd-mon-yyyy');
Explained.

SQL> select * from table(dbms_xplan.display);
PLAN_TABLE_OUTPUT
-------------------------------------------------------------------------------------------------
Plan hash value: 3971874201
-------------------------------------------------------------------------------------------------
| Id | Operation              | Name  | Rows | Bytes | Cost (%CPU)| Time     | Pstart| Pstop |
-------------------------------------------------------------------------------------------------
|  0 | SELECT STATEMENT       |       | 673  | 19517 | 27      (8)| 00:00:01 |       |       |
|  1 |  PARTITION RANGE SINGLE|       | 673  | 19517 | 27      (8)| 00:00:01 | 17    | 17    |
|* 2 |   TABLE ACCESS FULL    | SALES | 673  | 19517 | 27      (8)| 00:00:01 | 17    | 17    |
-------------------------------------------------------------------------------------------------
Predicate Information (identified by operation id):
---------------------------------------------------
   2 - filter("TIME_ID"=TO_DATE('2001-01-01 00:00:00', 'yyyy-mm-dd hh24:mi:ss'))
```

This plan shows that Oracle accesses partition number 17, as shown in the `PSTART` and `PSTOP` columns. The `OPERATION` column shows `PARTITION RANGE SINGLE`, indicating that only a single partition is being accessed. If `OPERATION` shows `PARTITION RANGE ALL`, then all partitions are being accessed and effectively no pruning takes place. `PSTART` then shows the very first partition of the table and `PSTOP` shows the very last partition.

An execution plan with a full table scan on an interval-partitioned table shows 1 for `PSTART`, and 1048575 for `PSTOP`, regardless of how many interval partitions were created.

3.1.5 Dynamic Partition Pruning

Oracle dynamic partition pruning is introduced in this topic.

Dynamic pruning occurs if pruning is possible and static pruning is not possible. The following examples show multiple dynamic pruning cases:

- Dynamic Pruning with Bind Variables
- Dynamic Pruning with Subqueries
- Dynamic Pruning with Star Transformation
- Dynamic Pruning with Nested Loop Joins

3.1.5.1 Dynamic Pruning with Bind Variables

Statements that use bind variables against partition columns result in dynamic pruning.

The following SQL statement is an example.

```
SQL> explain plan for select * from sales s where time_id in ( :a, :b, :c, :d);
Explained.

SQL> select * from table(dbms_xplan.display);
PLAN_TABLE_OUTPUT
-------------------------------------------------------------------------------------------------------
Plan hash value: 513834092
-------------------------------------------------------------------------------------------------------
| Id | Operation                          |    Name |Rows|Bytes|Cost (%CPU)|  Time   | Pstart| Pstop|
-------------------------------------------------------------------------------------------------------
|  0 | SELECT STATEMENT                   |         |2517|72993|    292 (0)|00:00:04|       |      |
|  1 |  INLIST ITERATOR                   |         |    |     |           |        |       |      |
|  2 |   PARTITION RANGE ITERATOR         |         |2517|72993|    292 (0)|00:00:04|KEY(I) |KEY(I)|
|  3 |    TABLE ACCESS BY LOCAL INDEX ROWID| SALES |2517|72993|    292 (0)|00:00:04|KEY(I) |KEY(I)|
|  4 |     BITMAP CONVERSION TO ROWIDS    |         |    |     |           |        |       |      |
|* 5 |      BITMAP INDEX SINGLE VALUE     |SALES_TIME_BIX| |   |           |        |KEY(I) |KEY(I)|
-------------------------------------------------------------------------------------------------------
Predicate Information (identified by operation id):
---------------------------------------------------
5 - access("TIME_ID"=:A OR "TIME_ID"=:B OR "TIME_ID"=:C OR "TIME_ID"=:D)
```

For parallel execution plans, only the partition start and stop columns contain the partition pruning information; the operation column contains information for the parallel operation, as shown in the following example:

```
SQL> explain plan for select * from sales where time_id in (:a, :b, :c, :d);
Explained.

SQL> select * from table(dbms_xplan.display);
PLAN_TABLE_OUTPUT
-------------------------------------------------------------------------------------------------------
Plan hash value: 4058105390
-------------------------------------------------------------------------------------------------------
| Id| Operation          | Name  |Rows|Bytes|Cost(%CP|  Time  |Pstart| Pstop|  TQ |INOUT| PQ Dis|
-------------------------------------------------------------------------------------------------------
```

```
|  0| SELECT STATEMENT   |       |2517|72993|  75(36)|00:00:01|      |      |     |     |       |
|  1|  PX COORDINATOR    |       |    |     |        |        |      |      |     |     |       |
|  2|  PX SEND QC(RANDOM)|:TQ10000|2517|72993| 75(36)|00:00:01|      |      |Q1,00| P->S|QC(RAND|
|  3|   PX BLOCK ITERATOR|       |2517|72993|  75(36)|00:00:01|KEY(I)|KEY(I)|Q1,00| PCWC|       |
|* 4|   TABLE ACCESS FULL| SALES |2517|72993|  75(36)|00:00:01|KEY(I)|KEY(I)|Q1,00| PCWP|       |
-------------------------------------------------------------------------------------------------------
Predicate Information (identified by operation id):
---------------------------------------------------
  4 - filter("TIME_ID"=:A OR "TIME_ID"=:B OR "TIME_ID"=:C OR "TIME_ID"=:D)
```

> **See Also:**
>
> *Oracle Database SQL Tuning Guide* for more information about `EXPLAIN PLAN` and how to interpret it

3.1.5.2 Dynamic Pruning with Subqueries

Statements that explicitly use subqueries against partition columns result in dynamic pruning.

The following SQL statement is an example.

```
SQL> explain plan for select sum(amount_sold) from sales where time_id in
     (select time_id from times where fiscal_year = 2000);
Explained.

SQL> select * from table(dbms_xplan.display);
PLAN_TABLE_OUTPUT
PLAN_TABLE_OUTPUT
-------------------------------------------------------------------------------------------------------
Plan hash value: 3827742054

-------------------------------------------------------------------------------------------------------
| Id  | Operation                  | Name  | Rows  | Bytes | Cost (%CPU)| Time     | Pstart| Pstop |
-------------------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT           |       |     1 |    25 |   523   (5)| 00:00:07 |       |       |
|   1 |  SORT AGGREGATE            |       |     1 |    25 |            |          |       |       |
|*  2 |   HASH JOIN                |       |   191K|  4676K|   523   (5)| 00:00:07 |       |       |
|*  3 |    TABLE ACCESS FULL       | TIMES |   304 |  3648 |    18   (0)| 00:00:01 |       |       |
|   4 |    PARTITION RANGE SUBQUERY|       |   918K|    11M|   498   (4)| 00:00:06 |KEY(SQ)|KEY(SQ)|
|   5 |     TABLE ACCESS FULL      | SALES |   918K|    11M|   498   (4)| 00:00:06 |KEY(SQ)|KEY(SQ)|
-------------------------------------------------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------

   2 - access("TIME_ID"="TIME_ID")
   3 - filter("FISCAL_YEAR"=2000)
```

> **See Also:**
>
> *Oracle Database SQL Tuning Guide* for more information about `EXPLAIN PLAN` and how to interpret it

3.1.5.3 Dynamic Pruning with Star Transformation

Statements that get transformed by the database using the star transformation result in dynamic pruning.

The following SQL statement is an example.

```
SQL> explain plan for select p.prod_name, t.time_id, sum(s.amount_sold)
     from sales s, times t, products p
     where s.time_id = t.time_id and s.prod_id = p.prod_id and t.fiscal_year = 2000
     and t.fiscal_week_number = 3 and p.prod_category = 'Hardware'
     group by t.time_id, p.prod_name;
Explained.

SQL> select * from table(dbms_xplan.display);
PLAN_TABLE_OUTPUT
------------------------------------------------------------------------------------------------------------
Plan hash value: 4020965003

------------------------------------------------------------------------------------------------------------
| Id  | Operation                             | Name                 | Rows  | Bytes | Pstart| Pstop |
------------------------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT                      |                      |     1 |    79 |       |       |
|   1 |  HASH GROUP BY                        |                      |     1 |    79 |       |       |
|*  2 |   HASH JOIN                           |                      |     1 |    79 |       |       |
|*  3 |    HASH JOIN                          |                      |     2 |    64 |       |       |
|*  4 |     TABLE ACCESS FULL                 | TIMES                |     6 |    90 |       |       |
|   5 |     PARTITION RANGE SUBQUERY          |                      |   587 |  9979 |KEY(SQ)|KEY(SQ)|
|   6 |      TABLE ACCESS BY LOCAL INDEX ROWID| SALES                |   587 |  9979 |KEY(SQ)|KEY(SQ)|
|   7 |       BITMAP CONVERSION TO ROWIDS     |                      |       |       |       |       |
|   8 |        BITMAP AND                     |                      |       |       |       |       |
|   9 |         BITMAP MERGE                  |                      |       |       |       |       |
|  10 |          BITMAP KEY ITERATION         |                      |       |       |       |       |
|  11 |           BUFFER SORT                 |                      |       |       |       |       |
|* 12 |            TABLE ACCESS FULL          | TIMES                |     6 |    90 |       |       |
|* 13 |           BITMAP INDEX RANGE SCAN     | SALES_TIME_BIX       |       |       |KEY(SQ)|KEY(SQ)|
|  14 |         BITMAP MERGE                  |                      |       |       |       |       |
|  15 |          BITMAP KEY ITERATION         |                      |       |       |       |       |
|  16 |           BUFFER SORT                 |                      |       |       |       |       |
|  17 |            TABLE ACCESS BY INDEX ROWID| PRODUCTS             |    14 |   658 |       |       |
|* 18 |             INDEX RANGE SCAN          | PRODUCTS_PROD_CAT_IX |    14 |       |       |       |
|* 19 |           BITMAP INDEX RANGE SCAN     | SALES_PROD_BIX       |       |       |KEY(SQ)|KEY(SQ)|
|  20 |    TABLE ACCESS BY INDEX ROWID        | PRODUCTS             |    14 |   658 |       |       |
|* 21 |     INDEX RANGE SCAN                  | PRODUCTS_PROD_CAT_IX |    14 |       |       |       |
------------------------------------------------------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------

   2 - access("S"."PROD_ID"="P"."PROD_ID")
   3 - access("S"."TIME_ID"="T"."TIME_ID")
   4 - filter("T"."FISCAL_WEEK_NUMBER"=3 AND "T"."FISCAL_YEAR"=2000)
  12 - filter("T"."FISCAL_WEEK_NUMBER"=3 AND "T"."FISCAL_YEAR"=2000)
  13 - access("S"."TIME_ID"="T"."TIME_ID")
  18 - access("P"."PROD_CATEGORY"='Hardware')
  19 - access("S"."PROD_ID"="P"."PROD_ID")
  21 - access("P"."PROD_CATEGORY"='Hardware')

Note
-----
   - star transformation used for this statement
```

> **Note:**
>
> The `Cost (%CPU)` and `Time` columns were removed from the plan table output in this example.

> **See Also:**
>
> *Oracle Database SQL Tuning Guide* for more information about `EXPLAIN PLAN` and how to interpret it

3.1.5.4 Dynamic Pruning with Nested Loop Joins

Statements that are most efficiently executed using a nested loop join use dynamic pruning.

The following SQL statement is an example.

```
SQL> explain plan for select t.time_id, sum(s.amount_sold)
     from sales s, times t
     where s.time_id = t.time_id and t.fiscal_year = 2000 and t.fiscal_week_number = 3
     group by t.time_id;
Explained.

SQL> select * from table(dbms_xplan.display);
PLAN_TABLE_OUTPUT
-----------------------------------------------------------------------------------------------------------
Plan hash value: 50737729

-----------------------------------------------------------------------------------------------------------
| Id  | Operation                  | Name  | Rows  | Bytes | Cost (%CPU)| Time     | Pstart| Pstop |
-----------------------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT           |       |     6 |   168 |   126   (4)| 00:00:02 |       |       |
|   1 |  HASH GROUP BY             |       |     6 |   168 |   126   (4)| 00:00:02 |       |       |
|   2 |   NESTED LOOPS             |       |  3683 |   100K|   125   (4)| 00:00:02 |       |       |
|*  3 |    TABLE ACCESS FULL       | TIMES |     6 |    90 |    18   (0)| 00:00:01 |       |       |
|   4 |    PARTITION RANGE ITERATOR|       |   629 |  8177 |    18   (6)| 00:00:01 |   KEY |   KEY |
|*  5 |     TABLE ACCESS FULL      | SALES |   629 |  8177 |    18   (6)| 00:00:01 |   KEY |   KEY |
-----------------------------------------------------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------

   3 - filter("T"."FISCAL_WEEK_NUMBER"=3 AND "T"."FISCAL_YEAR"=2000)
   5 - filter("S"."TIME_ID"="T"."TIME_ID")
```

> **See Also:**
>
> *Oracle Database SQL Tuning Guide* for more information about `EXPLAIN PLAN` and how to interpret it

3.1.6 Partition Pruning with Zone Maps

Partition pruning is enhanced to take advantage of zone maps for pruning of complete partitions. Providing enhanced pruning capabilities provides better performance with less resource consumption and shorter time-to-information.

A zone map is a independent access structure that can be built for a table. During table scans, zone maps enable you to prune disk blocks of a table and partitions of a partitioned table based on predicates on the table columns. Zone maps have no correlation to the partition key columns of a partitioned table, so statements on partitioned tables with zone maps can prune partitions based on non-partition key columns.

> **See Also:**
>
> *Oracle Database Data Warehousing Guide* for information about zone maps and attribute clustering

Partition pruning with zone maps is especially effective when the zone map column values correlate with partition key column values. For example, the correlation can be between columns of the partitioned table itself, such as a shipping date that has a correlation to the partition key column order date in the same partitioned table, or within the join zone map columns and the partitioned table, such as a join zone map column month description from a dimension table times that correlates with the partition key column day of the partitioned table.

Example 3-2 illustrates partition pruning with zone maps for correlated columns of a partitioned table. Column `s_shipdate` in the partitioned table `sales_range` correlates with the partition key column `order_date` because orders are normally shipped within a couple of days after an order was received.

Due to the correlation of `s_shipdate` and the partition key column any selective predicate on this column has a high likelihood to enable partition pruning for the partitioned table `sales_range`, without having the column as part of the partitioning key.

The following `SELECT` statement looks for all orders that were shipped in the first quarter of 1999:

```
SELECT * FROM sales_range
      WHERE s_shipdate BETWEEN to_date('01/01/1999','dd/mm/yyyy')
      AND to_date('03/01/1999','mm/dd/yyyy');
```

In the following execution plan for the previous `SELECT` statement, zone maps are used for partition pruning and also to prune blocks from the partitions that have to be accessed.

Partition pruning with zone maps is identified by having `KEY(ZM)` in the `PSTART` and `PSTOP` columns of the execution plan. The block level pruning of all accessed partitions is identified by the filter predicate at table access time (`id 2`).

Example 3-2 Partitioned table sales_range with attribute clustering and a zone map on a correlated column

```
CREATE TABLE sales_range(
  s_productid      NUMBER,
  s_saledate       DATE,
  s_shipdate       DATE,
  s_custid         NUMBER,
  s_totalprice     NUMBER)
CLUSTERING BY (s_shipdate)
WITH MATERIALIZED ZONEMAP
PARTITION BY RANGE (s_saledate)
 (PARTITION sal99q1 VALUES LESS THAN
   (TO_DATE('01-APR-1999', 'DD-MON-YYYY')),
  PARTITION sal99q2 VALUES LESS THAN
   (TO_DATE('01-JUL-1999', 'DD-MON-YYYY')),
  PARTITION sal99q3 VALUES LESS THAN
   (TO_DATE('01-OCT-1999', 'DD-MON-YYYY')),
  PARTITION sal99q4 VALUES LESS THAN
   (TO_DATE('01-JAN-2000', 'DD-MON-YYYY')));
```

Example 3-3 Execution plan for partition pruning with zone maps

```
-----------------------------------------------------------------------------------------------------------------
| Id  | Operation                       | Name         | Rows  | Bytes | Cost (%CPU)| Time     | Pstart| Pstop |
-----------------------------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT                |              |       |       |     3 (100)|          |       |       |
|   1 |  PARTITION RANGE ITERATOR       |              |    58 |  3306 |     3   (0)| 00:00:01 |KEY(ZM)|KEY(ZM)|
|*  2 |   TABLE ACCESS FULL WITH ZONEMAP| SALES_RANGE  |    58 |  3306 |     3   (0)| 00:00:01 |KEY(ZM)|KEY(ZM)|
-----------------------------------------------------------------------------------------------------------------


Predicate Information (identified by operation id):
---------------------------------------------------

2 - filter((SYS_ZMAP_FILTER('/* ZM_PRUNING */ SELECT "ZONE_ID$", CASE WHEN
 BITAND(zm."ZONE_STATE$",1)=1 THEN 1 ELSE CASE WHEN (zm."MAX_1_S_SHIPDATE" < :1 OR
 zm."MIN_1_S_SHIPDATE" > :2) THEN 3 ELSE 2 END END FROM "SH"."ZMAP$_SALES_RANGE" zm WHERE
 zm."ZONE_LEVEL$"=0 ORDER BY zm."ZONE_ID$"',SYS_OP_ZONE_ID(ROWID),TO_DATE(' 1999-01-01 00:00:00',
 'syyyy-mm-dd hh24:mi:ss'),TO_DATE(' 1999-03-01 00:00:00', 'syyyy-mm-dd hh24:mi:ss'))<3 AND
 "S_SHIPDATE">=TO_DATE(' 1999-01-01 00:00:00','syyyy-mm-dd hh24:mi:ss') AND "S_SHIPDATE"<=TO_DATE('
  1999-03-01 00:00:00', 'syyyy-mm-dd hh24:mi:ss')))
```

3.1.7 Partition Pruning Tips

Tips for partition pruning are introduced in this topic.

When using partition pruning, you should consider the following:

- Data Type Conversions
- Function Calls
- Collection Tables

> **Note:**
>
> When you manipulate a partition column with any function or transformation, such as `CAST` or `TRUNC`, partition pruning is not taking place.

3.1.7.1 Data Type Conversions

To get the maximum performance benefit from partition pruning, you should avoid constructs that require the database to convert the data type you specify.

Data type conversions typically result in dynamic pruning when static pruning would have otherwise been possible. SQL statements that benefit from static pruning perform better than statements that benefit from dynamic pruning.

A common case of data type conversions occurs when using the Oracle `DATE` data type. An Oracle `DATE` data type is not a character string but is only represented as such when querying the database; the format of the representation is defined by the NLS setting of the instance or the session. Consequently, the same reverse conversion has to happen when inserting data into a `DATE` field or when specifying a predicate on such a field.

A conversion can either happen implicitly or explicitly by specifying a `TO_DATE` conversion. Only a properly applied `TO_DATE` function guarantees that the database can uniquely determine the date value and using it potentially for static pruning, which is especially beneficial for single partition access.

Consider the following example that runs against the `sales` table. You would like to know the total revenue number for the year 2000. There are multiple ways you can retrieve the answer to the query, but not every method is equally efficient.

```
explain plan for SELECT SUM(amount_sold) total_revenue
FROM sales,
WHERE time_id between '01-JAN-00' and '31-DEC-00';
```

The plan should now be similar to the following:

```
-------------------------------------------------------------------------------------------------
| Id  | Operation                 | Name  | Rows  | Bytes | Cost (%CPU)| Time     | Pstart| Pstop |
-------------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT          |       |     1 |    13 |   525   (8)| 00:00:07 |       |       |
|   1 |  SORT AGGREGATE           |       |     1 |    13 |            |          |       |       |
|*  2 |   FILTER                  |       |       |       |            |          |       |       |
|   3 |    PARTITION RANGE ITERATOR|       |   230K|  2932K|   525   (8)| 00:00:07 |   KEY |   KEY |
|*  4 |     TABLE ACCESS FULL     | SALES |   230K|  2932K|   525   (8)| 00:00:07 |   KEY |   KEY |
-------------------------------------------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------

   2 - filter(TO_DATE('01-JAN-00')<=TO_DATE('31-DEC-00'))
   4 - filter("TIME_ID">='01-JAN-00' AND "TIME_ID"<='31-DEC-00')
```

In this case, the keyword `KEY` for both `PSTART` and `PSTOP` means that dynamic partition pruning occurs at run-time. Consider the following case.

```
explain plan for select sum(amount_sold)
from sales
where time_id between '01-JAN-2000' and '31-DEC-2000' ;
```

The execution plan now shows the following:

```
-------------------------------------------------------------------------------------------
| Id  | Operation                 | Name  | Rows  | Bytes | Cost (%CPU)| Pstart| Pstop |
-------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT          |       |     1 |    13 |   127   (4)|       |       |
|   1 |  SORT AGGREGATE           |       |     1 |    13 |            |       |       |
|   2 |   PARTITION RANGE ITERATOR|       |   230K|  2932K|   127   (4)|    13 |    16 |
|*  3 |    TABLE ACCESS FULL      | SALES |   230K|  2932K|   127   (4)|    13 |    16 |
-------------------------------------------------------------------------------------------
Predicate Information (identified by operation id):
---------------------------------------------------

   3 - filter("TIME_ID"<=TO_DATE(' 2000-12-31 00:00:00', "syyyy-mm-dd hh24:mi:ss"))
```

> **Note:**
>
> The `Time` column was removed from the execution plan.

The execution plan shows static partition pruning. The query accesses a contiguous list of partitions 13 to 16. In this particular case, the way the date format was specified matches the NLS date format setting. Though this example shows the most efficient execution plan, you cannot rely on the NLS date format setting to define a certain format.

```
alter session set nls_date_format='fmdd Month yyyy';

explain plan for select sum(amount_sold)
from sales
where time_id between '01-JAN-2000' and '31-DEC-2000' ;
```

The execution plan now shows the following:

```
-------------------------------------------------------------------------------------------
| Id  | Operation                  | Name  | Rows  | Bytes | Cost (%CPU)| Pstart| Pstop |
-------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT           |       |     1 |    13 |   525   (8)|       |       |
|   1 |  SORT AGGREGATE            |       |     1 |    13 |            |       |       |
|*  2 |   FILTER                   |       |       |       |            |       |       |
|   3 |    PARTITION RANGE ITERATOR|       |   230K|  2932K|   525   (8)|   KEY |   KEY |
|*  4 |     TABLE ACCESS FULL      | SALES |   230K|  2932K|   525   (8)|   KEY |   KEY |
-------------------------------------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------

   2 - filter(TO_DATE('01-JAN-2000')<=TO_DATE('31-DEC-2000'))
   4 - filter("TIME_ID">='01-JAN-2000' AND "TIME_ID"<='31-DEC-2000')
```

> **Note:**
>
> The `Time` column was removed from the execution plan.

This plan, which uses dynamic pruning, again is less efficient than the static pruning execution plan. To guarantee a static partition pruning plan, you should explicitly convert data types to match the partition column data type. For example:

```
explain plan for select sum(amount_sold)
from sales
where time_id between to_date('01-JAN-2000','dd-MON-yyyy')
  and to_date('31-DEC-2000','dd-MON-yyyy') ;


-------------------------------------------------------------------------------------------
| Id  | Operation                  | Name  | Rows  | Bytes | Cost (%CPU)| Pstart| Pstop |
-------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT           |       |     1 |    13 |   127   (4)|       |       |
|   1 |  SORT AGGREGATE            |       |     1 |    13 |            |       |       |
|   2 |   PARTITION RANGE ITERATOR|       |   230K|  2932K|   127   (4)|    13 |    16 |
|*  3 |    TABLE ACCESS FULL       | SALES |   230K|  2932K|   127   (4)|    13 |    16 |
-------------------------------------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------

   3 - filter("TIME_ID"<=TO_DATE(' 2000-12-31 00:00:00', 'syyyy-mm-dd hh24:mi:ss'))
```

> **Note:**
>
> The `Time` column was removed from the execution plan.

> **See Also:**
>
> - *Oracle Database SQL Language Reference* for details about the `DATE` data type
> - *Oracle Database Globalization Support Guide* for details about NLS settings and globalization issues

3.1.7.2 Function Calls

Functions can limit the ability of the optimizer to perform pruning.

There are several cases when the optimizer cannot perform pruning. One common reasons is when an operator is used on top of a partitioning column. This could be an explicit operator (for example, a function) or even an implicit operator introduced by Oracle as part of the necessary data type conversion for executing the statement. For example, consider the following query:

```
EXPLAIN PLAN FOR
SELECT SUM(quantity_sold)
FROM sales
WHERE time_id = TO_TIMESTAMP('1-jan-2000', 'dd-mon-yyyy');
```

Because `time_id` is of type `DATE` and Oracle must promote it to the `TIMESTAMP` type to get the same data type, this predicate is internally rewritten as:

```
TO_TIMESTAMP(time_id) = TO_TIMESTAMP('1-jan-2000', 'dd-mon-yyyy')
```

The execution plan for this statement is as follows:

```
-------------------------------------------------------------------------------------------
|Id | Operation            | Name  | Rows  | Bytes | Cost (%CPU)| Time     | Pstart| Pstop |
-------------------------------------------------------------------------------------------
| 0 | SELECT STATEMENT     |       |     1 |    11 |     6  (17)| 00:00:01 |       |       |
| 1 |  SORT AGGREGATE      |       |     1 |    11 |            |          |       |       |
| 2 |   PARTITION RANGE ALL|       |    10 |   110 |     6  (17)| 00:00:01 |     1 |    16 |
|*3 |    TABLE ACCESS FULL | SALES |    10 |   110 |     6  (17)| 00:00:01 |     1 |    16 |
-------------------------------------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------
3 - filter(INTERNAL_FUNCTION("TIME_ID")=TO_TIMESTAMP('1-jan-2000',:B1))

15 rows selected
```

The `SELECT` statement accesses all partitions even though pruning down to a single partition could have taken place. Consider the example to find the total sales revenue number for 2000. Another way to construct the query would be:

```
EXPLAIN PLAN FOR
SELECT SUM(amount_sold)
FROM sales
WHERE TO_CHAR(time_id,'yyyy') = '2000';
```

This query applies a function call to the partition key column, which generally disables partition pruning. The execution plan shows a full table scan with no partition pruning:

```
-------------------------------------------------------------------------------------------
| Id  | Operation            | Name  | Rows  | Bytes | Cost (%CPU)| Time     | Pstart| Pstop |
-------------------------------------------------------------------------------------------
```

```
|   0 | SELECT STATEMENT     |       |     1 |    13 |   527   (9)| 00:00:07 |       |       |
|   1 |  SORT AGGREGATE      |       |     1 |    13 |            |          |       |       |
|   2 |   PARTITION RANGE ALL|       |  9188 |   116K|   527   (9)| 00:00:07 |     1 |    28 |
|*  3 |    TABLE ACCESS FULL | SALES |  9188 |   116K|   527   (9)| 00:00:07 |     1 |    28 |
-------------------------------------------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------

   3 - filter(TO_CHAR(INTERNAL_FUNCTION("TIME_ID"),'yyyy')='2000')
```

Avoid using implicit or explicit functions on the partition columns. If your queries commonly use function calls, then consider using a virtual column and virtual column partitioning to benefit from partition pruning in these cases.

> **Note:**
>
> If any function is applied to the partition key column as part of the filter predicate - explicit or implicit as part of any necessary processing, then partition pruning is not taking place. This is true irrespective of the nature of the function and is true for example for any kind of type conversion, or truncation of values.

3.1.7.3 Collection Tables

Collection tables can limit the ability of the optimizer to perform pruning.

The following example illustrates what an `EXPLAIN PLAN` statement might look like when it contains Collection Tables, which, for the purposes of this discussion, are ordered collection tables or nested tables. A full table access is not performed because it is constrained to just the partition in question.

```
EXPLAIN PLAN FOR
SELECT p.ad_textdocs_ntab
FROM print_media_part p;

Explained.

PLAN_TABLE_OUTPUT
------------------------------------------------------------------------------
Plan hash value: 2207588228

------------------------------------------------------------------------------
| Id  | Operation                  | Name             | Pstart| Pstop |
------------------------------------------------------------------------------
|   0 | SELECT STATEMENT           |                  |       |       |
|   1 |  PARTITION REFERENCE SINGLE|                  |   KEY |   KEY |
|   2 |   TABLE ACCESS FULL        | TEXTDOC_NT       |   KEY |   KEY |
|   3 |  PARTITION RANGE ALL       |                  |     1 |     2 |
|   4 |   TABLE ACCESS FULL        | PRINT_MEDIA_PART |     1 |     2 |
------------------------------------------------------------------------------

Note
-----
  - dynamic sampling used for this statement
```

> **See Also:**
>
> Partitioning of Collections in XMLType and Objects for an example of the `CREATE TABLE` statement on which the `EXPLAIN PLAN` is based

3.2 Partition-Wise Operations

Partition-wise operations significantly reduce response time and improve the use of both CPU and memory resources.

Partition-wise joins can reduce query response time by minimizing the amount of data exchanged among parallel execution servers when joins execute in parallel. In Oracle Real Application Clusters (Oracle RAC) environments, partition-wise joins also avoid or at least limit the data traffic over the interconnect, which is the key to achieving good scalability for massive join operations. Parallel partition-wise joins are used commonly for processing large joins efficiently and fast. Partition-wise joins can be full or partial. Oracle Database decides which type of join to use.

In addition to parallel partition-wise joins, queries using the `SELECT DISTINCT` clause and SQL window functions can perform parallel partition-wise operations.

The following topics are discussed:

- Full Partition-Wise Joins
- Partial Partition-Wise Joins

> **See Also:**
>
> - Partition-Wise Joins in a Data Warehouse for information about parallel partition-wise operations in a data warehouse environment
> - *Oracle Database Data Warehousing Guide* for information about data warehousing and optimization techniques

3.2.1 Full Partition-Wise Joins

A full partition-wise join divides a large join into smaller joins between a pair of partitions from the two joined tables.

To use full partition-wise joins, you must equipartition both tables on their join keys, or use reference partitioning.

You can use various partitioning methods to equipartition both tables. These methods are described at a high level in the following topics:

- Querying a Full Partition-Wise Join
- Full Partition-Wise Joins: Single-Level - Single-Level
- Full Partition-Wise Joins: Composite - Single-Level
- Full Partition-Wise Joins: Composite - Composite

3.2.1.1 Querying a Full Partition-Wise Join

You can query using a full partition-wise join.

Consider a large join between a sales table and a customer table on the column `cust_id`, as shown in Example 3-4. The query "find the records of all customers who bought more than 100 articles in Quarter 3 of 1999" is a typical example of a SQL statement performing such a join.

Such a large join is typical in data warehousing environments. In this case, the entire customer table is joined with one quarter of the sales data. In large data warehouse applications, this might mean joining millions of rows. The join method to use in that case is obviously a hash join. You can reduce the processing time for this hash join even more if both tables are equipartitioned on the `cust_id` column. This functionality enables a full partition-wise join.

When you execute a full partition-wise join in parallel, the granule of parallelism is a partition. Consequently, the degree of parallelism is limited to the number of partitions. For example, you require at least 16 partitions to set the degree of parallelism of the query to 16.

Example 3-4 Querying with a full partition-wise join

```
SELECT c.cust_last_name, COUNT(*)
  FROM sales s, customers c
  WHERE s.cust_id = c.cust_id AND
        s.time_id BETWEEN TO_DATE('01-JUL-1999', 'DD-MON-YYYY') AND
        (TO_DATE('01-OCT-1999', 'DD-MON-YYYY'))
  GROUP BY c.cust_last_name HAVING COUNT(*) > 100;
```

3.2.1.2 Full Partition-Wise Joins: Single-Level - Single-Level

A single-level to single-level full partition-wise join is the simplest method: two tables are both partitioned by the join column.

In the example, the `customers` and `sales` tables are both partitioned on the `cust_id` columns. This partitioning method enables full partition-wise joins when the tables are joined on `cust_id`, both representing the same customer identification number. This scenario is available for range-range, list-list, and hash-hash partitioning. Interval-range and interval-interval full partition-wise joins are also supported and can be compared to range-range.

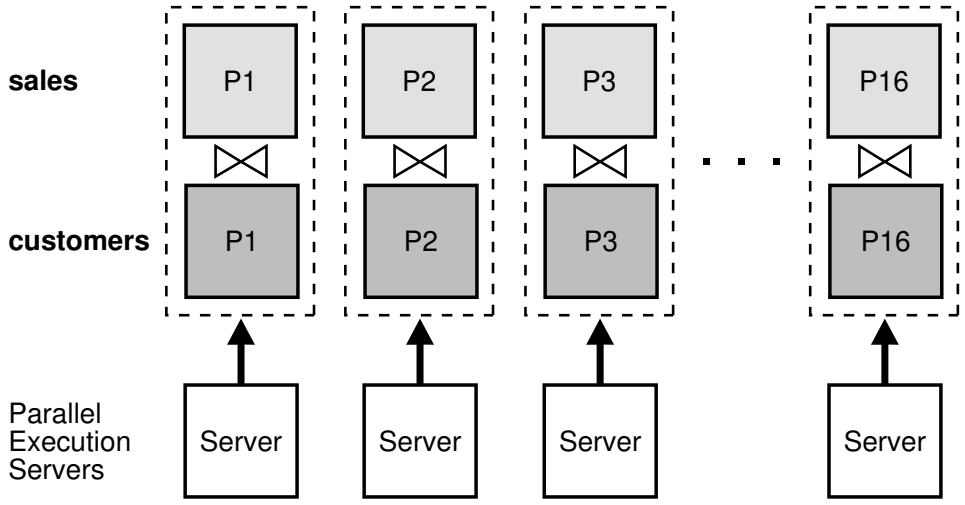
In serial, this join is performed between pairs of matching hash partitions, one at a time. When one partition pair has been joined, the join of another partition pair begins. The join completes when all partition pairs have been processed. To ensure a good workload distribution, you should either have many more partitions than the requested degree of parallelism or use equisize partitions with as many partitions as the requested degree of parallelism. Using hash partitioning on a unique or almost-unique column, with the number of partitions equal to a power of 2, is a good way to create equisized partitions.

Note:

- A pair of matching hash partitions is defined as one partition with the same partition number from each table. For example, with full partition-wise joins based on hash partitioning, the database joins partition 0 of `sales` with partition 0 of `customers`, partition 1 of `sales` with partition 1 of `customers`, and so on.
- Reference partitioning is an easy way to co-partition two tables so that the optimizer can always consider a full partition-wise join if the tables are joined in a statement.

Parallel execution of a full partition-wise join is a straightforward parallelization of the serial execution. Instead of joining one partition pair at a time, partition pairs are joined in parallel by the query servers. Figure 3-1 illustrates the parallel execution of a full partition-wise join.

Figure 3-1 Parallel Execution of a Full Partition-wise Join



The following example shows the execution plan for `sales` and `customers` co-partitioned by hash with the same number of partitions. The plan shows a full partition-wise join.

```
explain plan for SELECT c.cust_last_name, COUNT(*)
FROM sales s, customers c
WHERE s.cust_id = c.cust_id AND
s.time_id BETWEEN TO_DATE('01-JUL-1999', 'DD-MON-YYYY') AND
     (TO_DATE('01-OCT-1999', 'DD-MON-YYYY'))
GROUP BY c.cust_last_name HAVING COUNT(*) > 100;

-------------------------------------------------------------------------------------------------------------------
| Id  | Operation                    | Name      | Rows  | Bytes | Pstart| Pstop|    TQ  |IN-OUT| PQ Distrib|
-------------------------------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT             |           |    46 |  1196 |       |      |        |      |           |
|   1 |  PX COORDINATOR              |           |       |       |       |      |        |      |           |
|   2 |   PX SEND QC (RANDOM)        | :TQ10001  |    46 |  1196 |       |      |  Q1,01 | P->S | QC (RAND) |
|*  3 |    FILTER                    |           |       |       |       |      |  Q1,01 | PCWC |           |
|   4 |     HASH GROUP BY            |           |    46 |  1196 |       |      |  Q1,01 | PCWP |           |
|   5 |      PX RECEIVE              |           |    46 |  1196 |       |      |  Q1,01 | PCWP |           |
|   6 |       PX SEND HASH           | :TQ10000  |    46 |  1196 |       |      |  Q1,00 | P->P | HASH      |
|   7 |        HASH GROUP BY         |           |    46 |  1196 |       |      |  Q1,00 | PCWP |           |
|   8 |         PX PARTITION HASH ALL|           | 59158 |  1502K|     1 |   16 |  Q1,00 | PCWC |           |
|*  9 |          HASH JOIN           |           | 59158 |  1502K|       |      |  Q1,00 | PCWP |           |
|  10 |           TABLE ACCESS FULL  | CUSTOMERS | 55500 |   704K|     1 |   16 |  Q1,00 | PCWP |           |
|* 11 |           TABLE ACCESS FULL  | SALES     | 59158 |   751K|     1 |   16 |  Q1,00 | PCWP |           |
```

```
-------------------------------------------------------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------

   3 - filter(COUNT(SYS_OP_CSR(SYS_OP_MSR(COUNT(*)),0))>100)
   9 - access("S"."CUST_ID"="C"."CUST_ID")
  11 - filter("S"."TIME_ID"<=TO_DATE(' 1999-10-01 00:00:00', 'syyyy-mm-dd hh24:mi:ss') AND
"S"."TIME_ID">=TO_DATE(' 1999-07-01
              00:00:00', 'syyyy-mm-dd hh24:mi:ss'))
```

> **Note:**
>
> The `Cost (%CPU)` and `Time` columns were removed from the plan table output in this example.

In Oracle RAC environments running on massive parallel processing (MPP) platforms, placing partitions on nodes is critical to achieving good scalability. To avoid remote I/O, both matching partitions should have affinity to the same node. Partition pairs should be spread over all of the nodes to avoid bottlenecks and to use all CPU resources available on the system.

Nodes can host multiple pairs when there are more pairs than nodes. For example, with an 8-node system and 16 partition pairs, each node receives two pairs.

> **See Also:**
>
> *Oracle Real Application Clusters Administration and Deployment Guide* for more information about data affinity

3.2.1.3 Full Partition-Wise Joins: Composite - Single-Level

A composite to single-level full partition-wise join is a variation of the single-level - single-level method.

In this scenario, one table (typically the larger table) is composite partitioned on two dimensions, using the join columns as the subpartition key. In the example, the `sales` table is a typical example of a table storing historical data. Using range partitioning is a logical initial partitioning method for a table storing historical information.

For example, assume you want to partition the `sales` table into eight partitions by range on the column `time_id`. Also assume you have two years and that each partition represents a quarter. Instead of using range partitioning, you can use composite partitioning to enable a full partition-wise join while preserving the partitioning on `time_id`. For example, partition the `sales` table by range on `time_id` and then subpartition each partition by hash on `cust_id` using 16 subpartitions for each partition, for a total of 128 subpartitions. The `customers` table can use hash partitioning with 16 partitions.

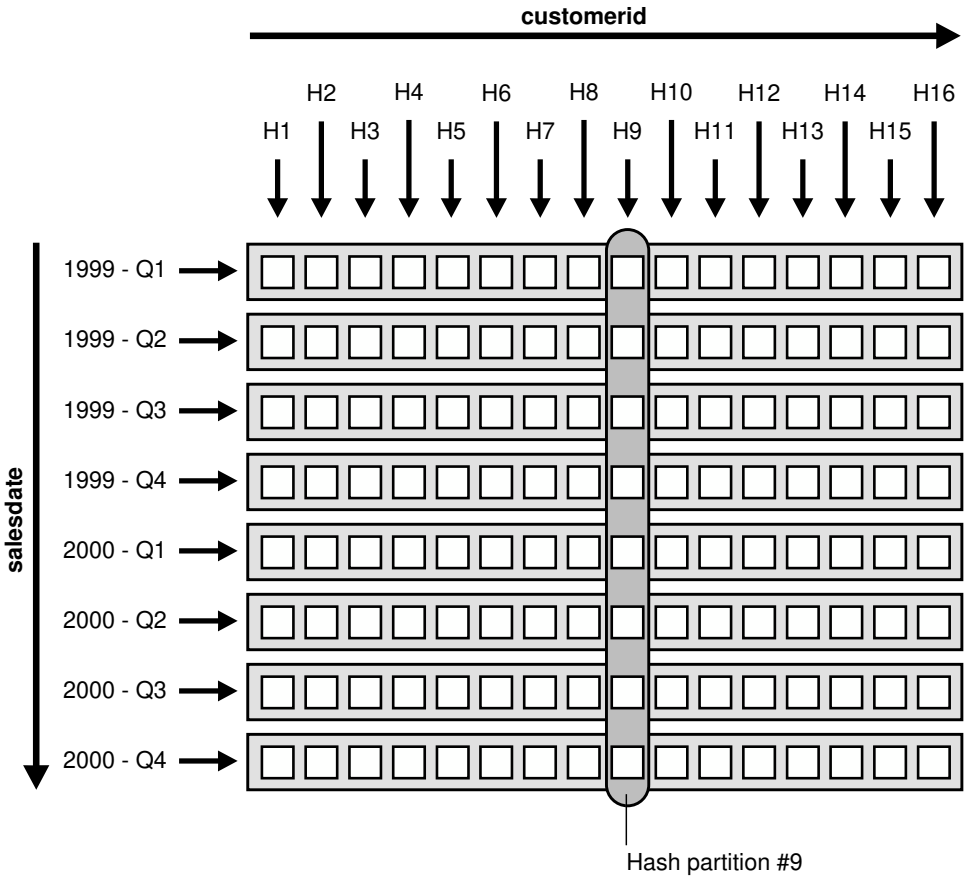
When you use the method just described, a full partition-wise join works similarly to the one created by a single-level - single-level hash-hash method. The join is still divided into 16 smaller joins between hash partition pairs from both tables. The difference is that now each hash partition in the `sales` table is composed of a set of 8 subpartitions, one from each range partition.

Figure 3-2 illustrates how the hash partitions are formed in the `sales` table. Each cell represents a subpartition. Each row corresponds to one range partition, for a total of 8 range

partitions. Each range partition has 16 subpartitions. Each column corresponds to one hash partition for a total of 16 hash partitions; each hash partition has 8 subpartitions. Hash partitions can be defined only if all partitions have the same number of subpartitions, in this case, 16.

Hash partitions are implicit in a composite table. However, Oracle does not record them in the data dictionary, and you cannot manipulate them with DDL commands as you can range or list partitions.

Figure 3-2 Range and Hash Partitions of a Composite Table



The following example shows the execution plan for the full partition-wise join with the `sales` table range partitioned by `time_id`, and subpartitioned by hash on `cust_id`.

```
-------------------------------------------------------------------------------------------------
| Id  | Operation                          | Name       | Pstart| Pstop |IN-OUT| PQ Distrib |
-------------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT                   |            |       |       |      |            |
|   1 |  PX COORDINATOR                    |            |       |       |      |            |
|   2 |   PX SEND QC (RANDOM)              | :TQ10001   |       |       | P->S | QC (RAND)  |
|*  3 |    FILTER                          |            |       |       | PCWC |            |
|   4 |     HASH GROUP BY                  |            |       |       | PCWP |            |
|   5 |      PX RECEIVE                    |            |       |       | PCWP |            |
|   6 |       PX SEND HASH                 | :TQ10000   |       |       | P->P | HASH       |
|   7 |        HASH GROUP BY               |            |       |       | PCWP |            |
|   8 |         PX PARTITION HASH ALL      |            |     1 |    16 | PCWC |            |
|*  9 |          HASH JOIN                 |            |       |       | PCWP |            |
```

```
|  10 |           TABLE ACCESS FULL          | CUSTOMERS |     1 |    16 | PCWP |            |
|  11 |           PX PARTITION RANGE ITERATOR|           |     8 |     9 | PCWC |            |
|* 12 |            TABLE ACCESS FULL         | SALES     |   113 |   144 | PCWP |            |
-------------------------------------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------

   3 - filter(COUNT(SYS_OP_CSR(SYS_OP_MSR(COUNT(*)),0))>100)
   9 - access("S"."CUST_ID"="C"."CUST_ID")
  12 - filter("S"."TIME_ID"<=TO_DATE(' 1999-10-01 00:00:00', 'syyyy-mm-dd hh24:mi:ss') AND
"S"."TIME_ID">=TO_DATE(' 1999-07-01
              00:00:00', 'syyyy-mm-dd hh24:mi:ss'))
```

> **Note:**
>
> The `Rows`, `Cost (%CPU)`, `Time`, and `TQ` columns were removed from the plan table output in this example.

Composite - single-level partitioning is effective because it enables you to combine pruning on one dimension with a full partition-wise join on another dimension. In the previous example query, pruning is achieved by scanning only the subpartitions corresponding to Q3 of 1999, in other words, row number 3 in Figure 3-2. Oracle then joins these subpartitions with the customer table, using a full partition-wise join.

All characteristics of the single-level - single-level partition-wise join apply to the composite - single-level partition-wise join. In particular, for this example, these two points are common to both methods:

- The degree of parallelism for this full partition-wise join cannot exceed 16. Even though the `sales` table has 128 subpartitions, it has only 16 *hash partitions*.
- A partition is now a collection of subpartitions. For example, in Figure 3-2, store hash partition 9 of the `sales` table shown by the eight circled subpartitions, on the same node as hash partition 9 of the `customers` table.

3.2.1.4 Full Partition-Wise Joins: Composite - Composite

You can use composite to composite full partition-wise joins for additional flexibility.

If needed, you can also partition the `customers` table by the composite method. For example, you partition it by range on a postal code column to enable pruning based on postal codes. You then subpartition it by hash on `cust_id` using the same number of partitions (16) to enable a partition-wise join on the hash dimension.

You can get full partition-wise joins on all combinations of partition and subpartition partitions: partition - partition, partition - subpartition, subpartition - partition, and subpartition - subpartition.

3.2.2 Partial Partition-Wise Joins

With partial partition-wise joins, only one table must be partitioned on the join key.

Oracle Database can perform partial partition-wise joins only in parallel. Unlike full partition-wise joins, partial partition-wise joins require you to partition only one table on the join key, not both tables. The partitioned table is referred to as the reference table. The other table may or

may not be partitioned. Partial partition-wise joins are more common than full partition-wise joins.

To execute a partial partition-wise join, the database dynamically repartitions the other table based on the partitioning of the reference table. After the other table is repartitioned, the execution is similar to a full partition-wise join.

The performance advantage that partial partition-wise joins have over joins in nonpartitioned tables is that the reference table is not moved during the join operation. Parallel joins between nonpartitioned tables require both input tables to be redistributed on the join key. This redistribution operation involves exchanging rows between parallel execution servers. This is a CPU-intensive operation that can lead to excessive interconnect traffic in Oracle RAC environments. Partitioning large tables on a join key, either a foreign or primary key, prevents this redistribution every time the table is joined on that key. Of course, if you choose a foreign key to partition the table, which is the most common scenario, then select a foreign key that is involved in many queries.

To illustrate partial partition-wise joins, consider the previous `sales/customers` example. Assume that `customers` is not partitioned or is partitioned on a column other than `cust_id`. Because `sales` is often joined with `customers` on `cust_id`, and because this join dominates our application workload, partition `sales` on `cust_id` to enable partial partition-wise joins every time `customers` and `sales` are joined. As with full partition-wise joins, you have several alternatives:

- Partial Partition-Wise Joins: Single-Level Partitioning
- Partial Partition-Wise Joins: Composite

3.2.2.1 Partial Partition-Wise Joins: Single-Level Partitioning

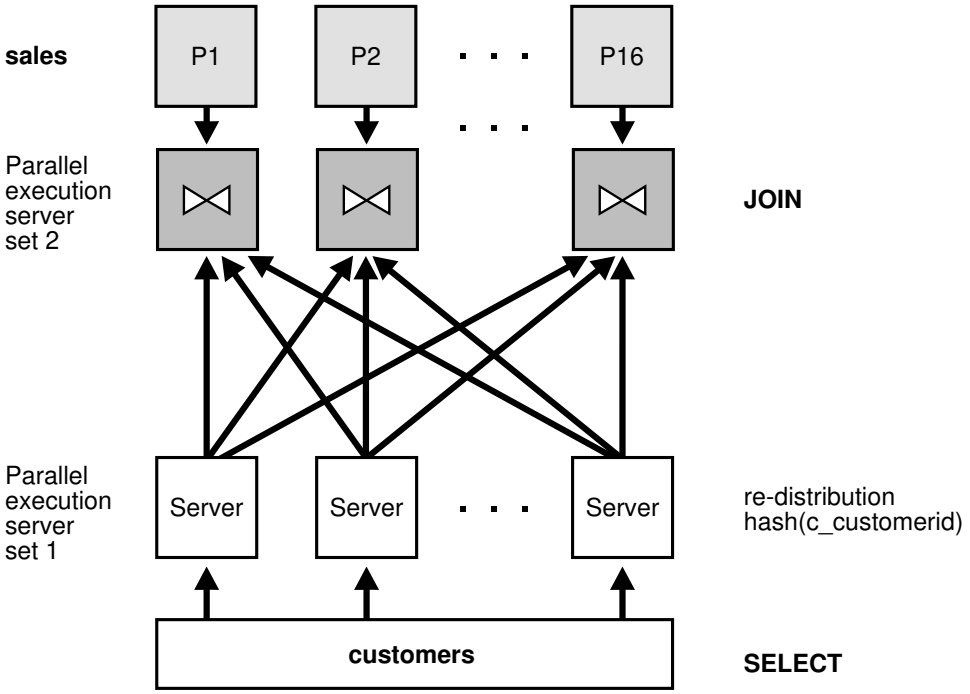
A single-level partial partition-wise join is the simplest method to enable a partial partition-wise join.

For example, you can enable a single-level partial partition-wise join to partition `sales` by hash on `cust_id`. The number of partitions determines the maximum degree of parallelism, because the partition is the smallest granule of parallelism for partial partition-wise join operations.

The parallel execution of a partial partition-wise join is illustrated in Figure 3-3, which assumes that both the degree of parallelism and the number of partitions of `sales` are 16. The execution involves two sets of query servers: one set, labeled *set 1* in Figure 3-3, scans the `customers` table in parallel. The granule of parallelism for the scan operation is a range of blocks.

Rows from `customers` that are selected by the first set, in this case all rows, are redistributed to the second set of query servers by hashing `cust_id`. For example, all rows in `customers` that could have matching rows in partition `P1` of `sales` are sent to query server 1 in the second set. Rows received by the second set of query servers are joined with the rows from the corresponding partitions in `sales`. Query server number 1 in the second set joins all `customers` rows that it receives with partition `P1` of `sales`.

Figure 3-3 Partial Partition-Wise Join



The example below shows the execution plan for the partial partition-wise join between `sales` and `customers`.

```
-------------------------------------------------------------------------------------------------
| Id  | Operation                               | Name      | Pstart| Pstop |IN-OUT| PQ Distrib |
-------------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT                        |           |       |       |      |            |
|   1 |  PX COORDINATOR                         |           |       |       |      |            |
|   2 |   PX SEND QC (RANDOM)                   | :TQ10002  |       |       | P->S | QC (RAND)  |
|*  3 |    FILTER                               |           |       |       | PCWC |            |
|   4 |     HASH GROUP BY                       |           |       |       | PCWP |            |
|   5 |      PX RECEIVE                         |           |       |       | PCWP |            |
|   6 |       PX SEND HASH                      | :TQ10001  |       |       | P->P | HASH       |
|   7 |        HASH GROUP BY                    |           |       |       | PCWP |            |
|*  8 |         HASH JOIN                       |           |       |       | PCWP |            |
|   9 |          PART JOIN FILTER CREATE        | :BF0000   |       |       | PCWP |            |
|  10 |           PX RECEIVE                    |           |       |       | PCWP |            |
|  11 |            PX SEND PARTITION (KEY)      | :TQ10000  |       |       | P->P | PART (KEY) |
|  12 |             PX BLOCK ITERATOR           |           |       |       | PCWC |            |
|  13 |              TABLE ACCESS FULL          | CUSTOMERS |       |       | PCWP |            |
|  14 |          PX PARTITION HASH JOIN-FILTER  |           |:BF0000|:BF0000| PCWC |            |
|* 15 |           TABLE ACCESS FULL             | SALES     |:BF0000|:BF0000| PCWP |            |
-------------------------------------------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------

   3 - filter(COUNT(SYS_OP_CSR(SYS_OP_MSR(COUNT(*)),0))>100)
   8 - access("S"."CUST_ID"="C"."CUST_ID")
  15 - filter("S"."TIME_ID"<=TO_DATE(' 1999-10-01 00:00:00', 'syyyy-mm-dd hh24:mi:ss') AND
"S"."TIME_ID">=TO_DATE(' 1999-07-01
              00:00:00', 'syyyy-mm-dd hh24:mi:ss'))
```

This query runs in parallel, as displayed in the plan, because there are `PX` row sources. One table is partitioned, which is the `SALES` table. You can determine this because the `PX PARTITION HASH` row source contains a nonpartitioned table `CUSTOMERS` that is distributed through `PX SEND PARTITION` to a different worker set that performs the join.

> **Note:**
>
> The `Rows`, `Cost (%CPU)`, `Time`, and `TQ` columns were removed from the plan table output in this example.

> **Note:**
>
> This discussion is based on hash partitioning, but it also applies for range, list, and interval partial partition-wise joins.

Considerations for full partition-wise joins also apply to partial partition-wise joins:

- The degree of parallelism does not need to equal the number of partitions. In Figure 3-3, the query executes with two sets of 16 query servers. In this case, Oracle assigns 1 partition to each query server of the second set. Again, the number of partitions should always be a multiple of the degree of parallelism.
- In Oracle RAC environments on MPPs, each hash partition of `sales` should preferably have affinity to only one node to avoid remote I/Os. Also, spread partitions over all nodes to avoid bottlenecks and use all CPU resources available on the system. A node can host multiple partitions when there are more partitions than nodes.

> **See Also:**
>
> *Oracle Real Application Clusters Administration and Deployment Guide* for more information about data affinity

3.2.2.2 Partial Partition-Wise Joins: Composite

You can use composite partial partition-wise joins.

As with full partition-wise joins, the prime partitioning method for the `sales` table is to use the range method on column `time_id`. This is because `sales` is a typical example of a table that stores historical data. To enable a partial partition-wise join while preserving this range partitioning, subpartition `sales` by hash on column `cust_id` using 16 subpartitions for each partition. Both pruning and partial partition-wise joins can be used if a query joins `customers` and `sales` and if the query has a selection predicate on `time_id`.

When the `sales` table is composite partitioned, the granule of parallelism for a partial partition-wise join is a hash partition and not a subpartition. Refer to Figure 3-2 for an illustration of a hash partition in a composite table. Again, the number of hash partitions should be a multiple of the degree of parallelism. Also, on an MPP system, ensure that each hash partition has affinity to a single node. In the previous example, the eight subpartitions composing a hash partition should have affinity to the same node.

> **Note:**
>
> This discussion is based on range-hash, but it also applies for all other combinations of composite partial partition-wise joins.

The following example shows the execution plan for the query between `sales` and `customers` with sales range partitioned by `time_id` and subpartitioned by hash on `cust_id`.

```
-------------------------------------------------------------------------------------------------
| Id  | Operation                           | Name      | Pstart| Pstop |IN-OUT| PQ Distrib |
-------------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT                    |           |       |       |      |            |
|   1 |  PX COORDINATOR                     |           |       |       |      |            |
|   2 |   PX SEND QC (RANDOM)               | :TQ10002  |       |       | P->S | QC (RAND)  |
|*  3 |    FILTER                           |           |       |       | PCWC |            |
|   4 |     HASH GROUP BY                   |           |       |       | PCWP |            |
|   5 |      PX RECEIVE                     |           |       |       | PCWP |            |
|   6 |       PX SEND HASH                  | :TQ10001  |       |       | P->P | HASH       |
|   7 |        HASH GROUP BY                |           |       |       | PCWP |            |
|*  8 |         HASH JOIN                   |           |       |       | PCWP |            |
|   9 |          PART JOIN FILTER CREATE    | :BF0000   |       |       | PCWP |            |
|  10 |           PX RECEIVE                |           |       |       | PCWP |            |
|  11 |            PX SEND PARTITION (KEY)  | :TQ10000  |       |       | P->P | PART (KEY) |
|  12 |             PX BLOCK ITERATOR       |           |       |       | PCWC |            |
|  13 |              TABLE ACCESS FULL      | CUSTOMERS |       |       | PCWP |            |
|  14 |          PX PARTITION RANGE ITERATOR|           |     8 |     9 | PCWC |            |
|  15 |           PX PARTITION HASH ALL     |           |     1 |    16 | PCWC |            |
|* 16 |            TABLE ACCESS FULL        | SALES     |   113 |   144 | PCWP |            |
-------------------------------------------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------

   3 - filter(COUNT(SYS_OP_CSR(SYS_OP_MSR(COUNT(*)),0))>100)
   8 - access("S"."CUST_ID"="C"."CUST_ID")
  16 - filter("S"."TIME_ID"<=TO_DATE(' 1999-10-01 00:00:00', 'syyyy-mm-dd hh24:mi:ss') AND
"S"."TIME_ID">=TO_DATE(' 1999-07-01
              00:00:00', 'syyyy-mm-dd hh24:mi:ss'))
```

> **Note:**
>
> The `Rows`, `Cost (%CPU)`, `Time`, and `TQ` columns were removed from the plan table output in this example.

3.3 Index Partitioning

Partitioning indexes has recommendations and considerations in common with partitioning tables.

The rules for partitioning indexes are similar to those for tables:

- An index can be partitioned unless:
  - The index is a cluster index.

  - The index is defined on a clustered table.
- You can mix partitioned and nonpartitioned indexes with partitioned and nonpartitioned tables:
  - A partitioned table can have partitioned or nonpartitioned indexes.
  - A nonpartitioned table can have partitioned or nonpartitioned indexes.
- Bitmap indexes on nonpartitioned tables cannot be partitioned.
- A bitmap index on a partitioned table must be a local index.

However, partitioned indexes are more complicated than partitioned tables because there are three types of partitioned indexes:

- Local prefixed
- Local nonprefixed
- Global prefixed

Oracle Database supports all three types. However, there are some restrictions. For example, a key cannot be an expression when creating a local unique index on a partitioned table.

The following topics are discussed:

- Local Partitioned Indexes
- Global Partitioned Indexes
- Summary of Partitioned Index Types
- The Importance of Nonprefixed Indexes
- Performance Implications of Prefixed and Nonprefixed Indexes
- Advanced Index Compression With Partitioned Indexes
- Guidelines for Partitioning Indexes
- Physical Attributes of Index Partitions

> **See Also:**
>
> *Oracle Database Reference* for information about the `DBA_INDEXES`, `DBA_IND_PARTITIONS`, `DBA_IND_SUBPARTITIONS`, and `DBA_PART_INDEXES` views.

3.3.1 Local Partitioned Indexes

In a local index, all keys in a particular index partition refer only to rows stored in a single underlying table partition.

A local index is created by specifying the `LOCAL` attribute. Oracle constructs the local index so that it is equipartitioned with the underlying table. Oracle partitions the index on the same columns as the underlying table, creates the same number of partitions or subpartitions, and gives them the same partition bounds as corresponding partitions of the underlying table.

Oracle also maintains the index partitioning automatically when partitions in the underlying table are added, dropped, merged, or split, or when hash partitions or subpartitions are added or coalesced. This ensures that the index remains equipartitioned with the table.

A local index can be created UNIQUE if the partitioning columns form a subset of the index columns. This restriction guarantees that rows with identical index keys always map into the same partition, where uniqueness violations can be detected.

Local indexes have the following advantages:

- Only one index partition must be rebuilt when a maintenance operation other than SPLIT PARTITION or ADD PARTITION is performed on an underlying table partition.
- The duration of a partition maintenance operation remains proportional to partition size if the partitioned table has only local indexes.
- Local indexes support partition independence.
- Local indexes support smooth roll-out of old data and roll-in of new data in historical tables.
- Oracle can take advantage of the fact that a local index is equipartitioned with the underlying table to generate better query access plans.
- Local indexes simplify the task of tablespace incomplete recovery. To recover a partition or subpartition of a table to a point in time, you must also recover the corresponding index entries to the same point in time. The only way to accomplish this is with a local index. Then you can recover the corresponding table and index partitions or subpartitions.

The following topics are discussed:

- Local Prefixed Indexes
- Local Nonprefixed Indexes

> **See Also:**
>
> *Oracle Database PL/SQL Packages and Types Reference* for a description of the DBMS_PCLXUTIL package

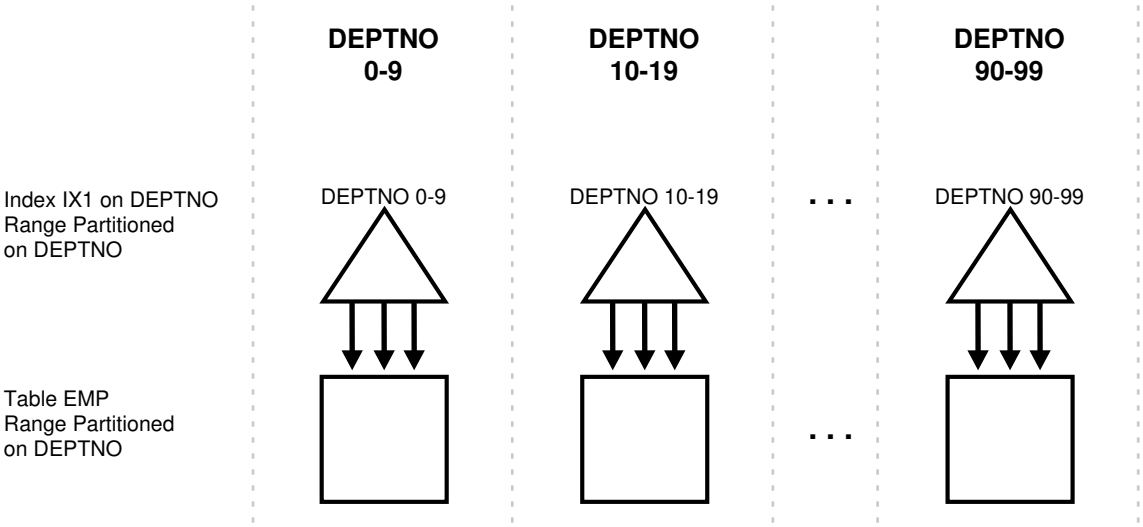
3.3.1.1 Local Prefixed Indexes

A local index is prefixed if it is partitioned on a left prefix of the index columns and the subpartioning key is included in the index key. Local prefixed indexes can be unique or nonunique.

For example, if the sales table and its local index sales_ix are partitioned on the week_num column, then index sales_ix is local prefixed if it is defined on the columns (week_num, xaction_num). On the other hand, if index sales_ix is defined on column product_num then it is not prefixed.

#unique_120/unique_120_Connect_42_I1008832 illustrates another example of a local prefixed index.

Figure 3-4 Local Prefixed Index



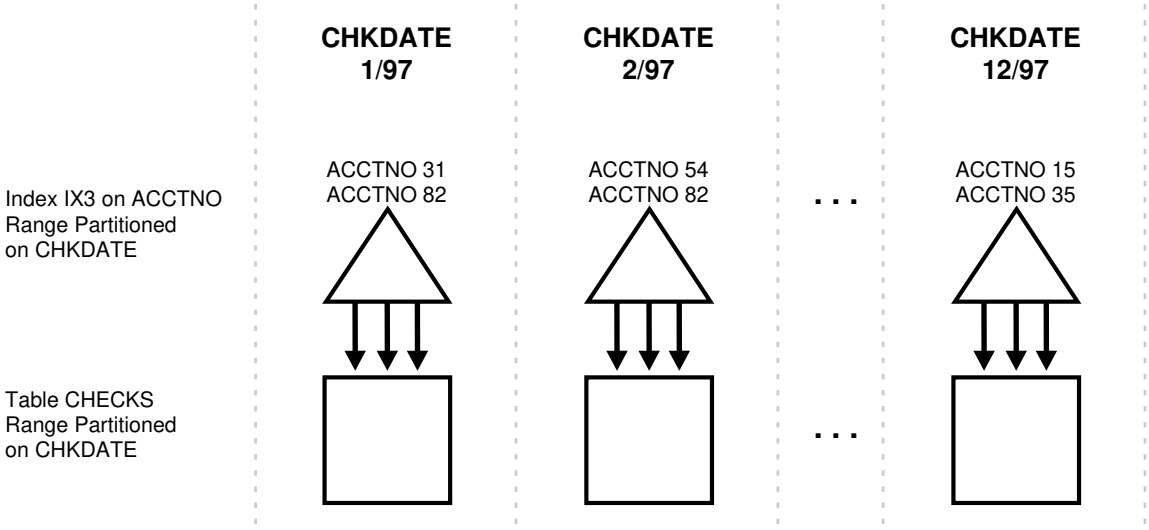
3.3.1.2 Local Nonprefixed Indexes

A local index is nonprefixed if it is not partitioned on a left prefix of the index columns or if the index key does not include the subpartitioning key.

You cannot have a unique local nonprefixed index unless the partitioning key is a subset of the index key.

Figure 3-5 illustrates an example of a local nonprefixed index.

Figure 3-5 Local Nonprefixed Index



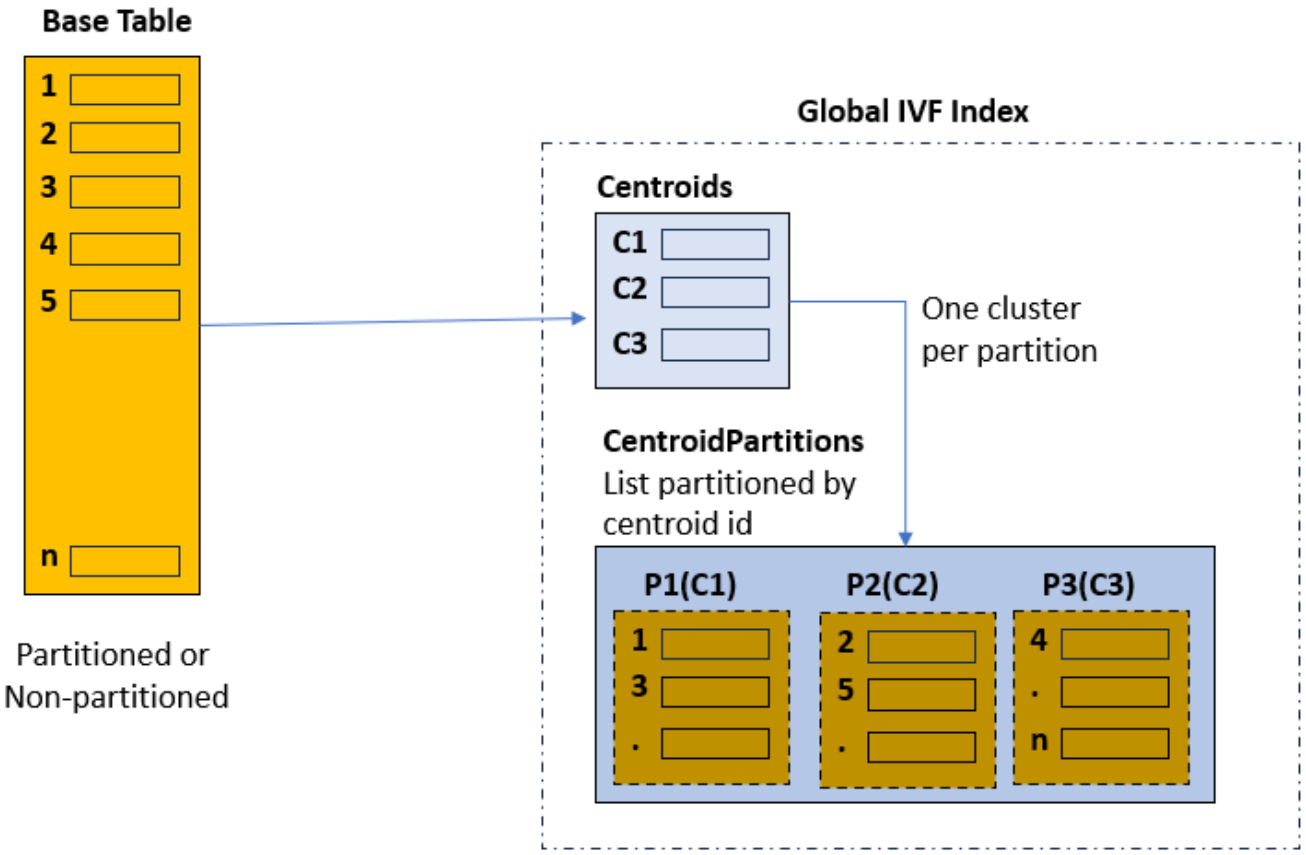
3.3.1.2.1 Inverted File Flat Vector Indexes Partitioning Schemes

Inverted File Flat vector indexes support both global and local indexes on partitioned tables. By default, IVF indexes are globally partitioned by centroid.

A **global IVF index** is composed of two tables:

- One called `VECTOR$<base table name>_IVF_IDX$<object info>$IVF_FLAT_CENTROIDS`, containing the list of identified centroid vectors and associated ids.
- The second called `VECTOR$<base table name>_IVF_IDX$<object info>$IVF_FLAT_CENTROID_PARTITIONS`, which is list-partitioned on the centroid ids. Each partition contains the base table vectors closely related (cluster) to the corresponding centroid id for that partition.

This is illustrated by the following diagram:



This structure is used to accelerate searches in the index by identifying first the centroid that is the closest to your query vector, and then use the corresponding centroid id to prune unnecessary partitions.

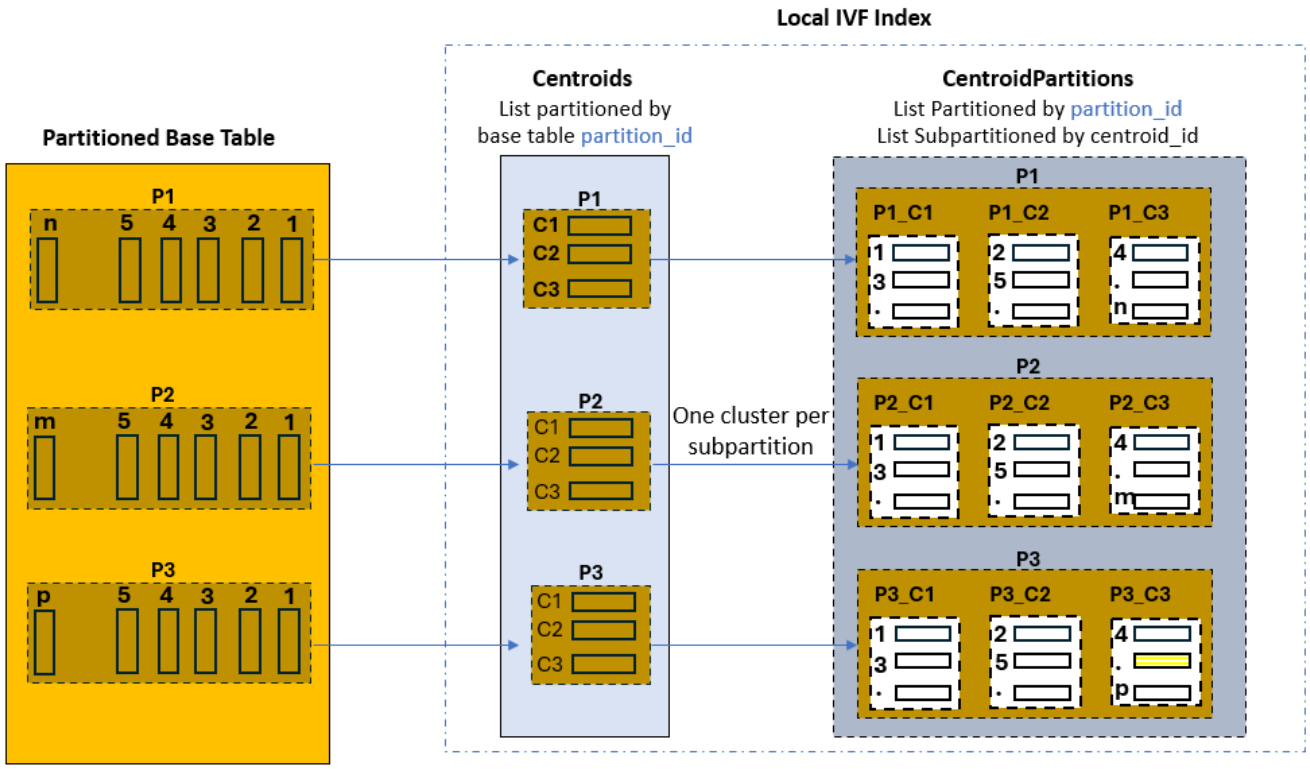
However, if the base table is partitioned on some relational data and your query is filtering on the base table partition key, then global IVF indexes are not optimal because they are completely independent of the base table partition key. For example, if you search the top-10 houses in California similar to a vectorized picture, the picture itself has most probably no relationship with the state of California. While your query benefits from the fact that your base

table is partitioned by state, so you can search only the partition corresponding to California, the query still must look at pictures that may not be in California.

To further accelerate such type of queries, you have the possibility to create a **local IVF index**. The term **local** for an index refers to a one-to-one relationship between the base table partitions or subpartitions and the index partitions.

This is illustrated by the following graphic, where the base table has three partitions. The created local IVF index is still constituted by two internal tables:

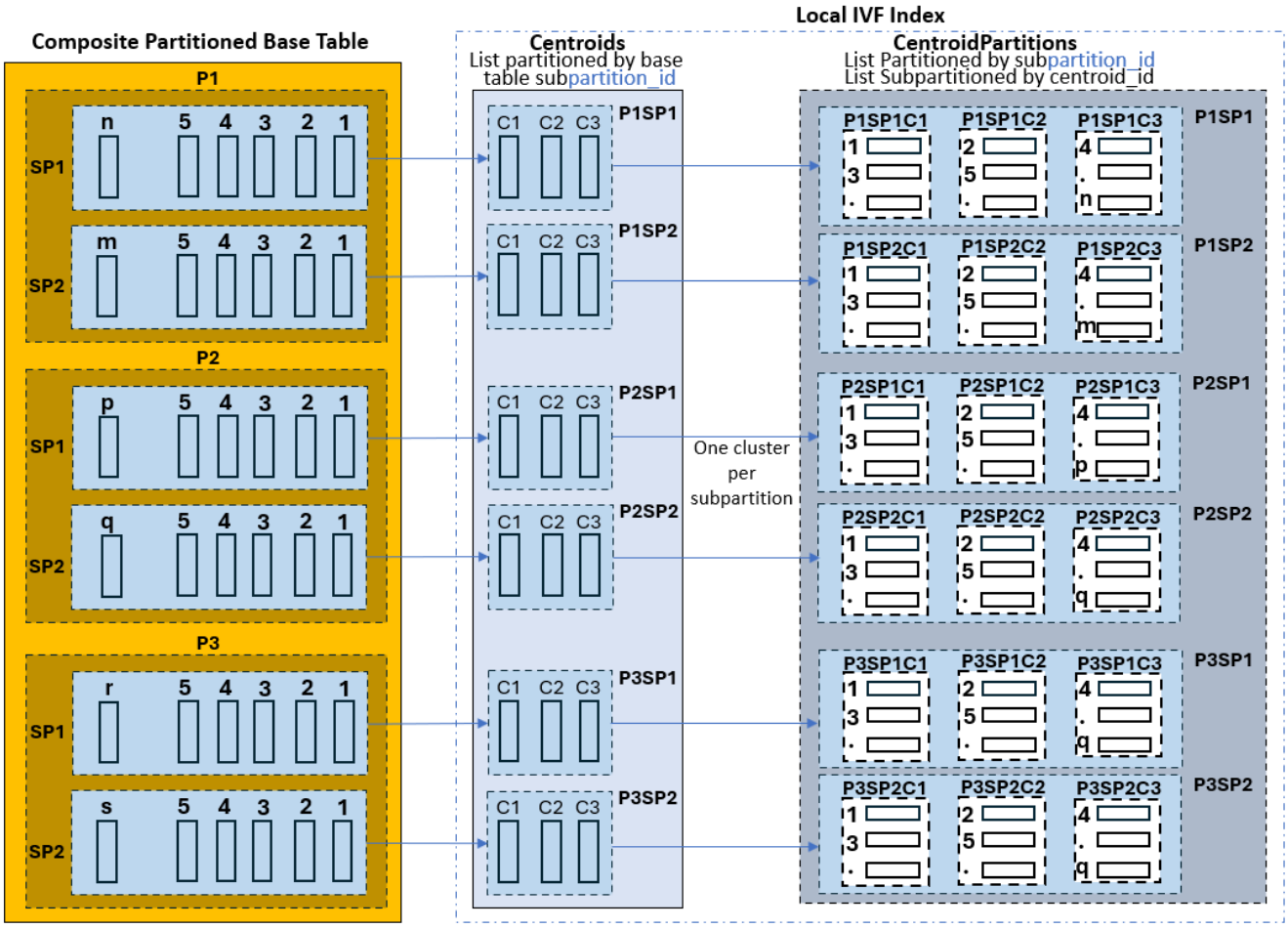
- One called `VECTOR$<base table name>_IVF_IDX$<object info>$IVF_FLAT_CENTROIDS`, which is list-partitioned by base table partition ids, and is thus equi-partioned with the base table. Each partition containing the list of corresponding identified centroid vectors and associated ids.
- The second called `VECTOR$<base table name>_IVF_IDX$<object info>$IVF_FLAT_CENTROID_PARTITIONS`, which is list-partitioned by base table partition id and list-subpartitioned by centroid id. This table is also equi-partitioned with the base table, and each subpartition contains the base table vectors closely related (cluster) to the corresponding centroid id for that subpartition.



Coming back to our initial example where you search the top-10 houses in California similar to a vectorized picture; your query benefits from partition pruning on the base table and Centroids table (California) as they are both partitioned by state. In addition, and once the closest centroid is identified in that partition, the query simply needs to scan the corresponding centroid cluster subpartition in the Centroid Partitions table without having to scan other centroid subpartitions.

Another possibility is for the base table to be composite partitioned. Here is a graphical representation corresponding to that case. The Centroids table is list-partitioned according to base table subpartitions. Each partition in the Centroids table containing all centroid vectors

found in the corresponding base table subpartition. The Centroid Partitions table is list-partitioned by base table subpartition id, and is further subpartitioned by centroid id:



> **Note:**
>
> You can create local IVF indexes only on a partitioned base table.
>
> Local IVF indexes inherit all system catalog tables and views used by regular local indexes. A flag (`idx_spare2`) in the `vecsys.vector$index` table indicates if an index is a local or global vector index.

Using local IVF indexes brings additional advantages:

- Simplified partition management operations (PMOP):

  For example, dropping a table partition just involves dropping the corresponding index partition.

- Flexible Indexing schemes:

  For example, marking certain index partitions `UNUSABLE` to avoid indexing certain table partitions through partial indexing.

> **Note:**
>
> If you want your user queries to use full potential of local IVF indexes, by taking the benefit of partition pruning, then user queries must satisfy the following conditions:
>
> - The base table is [sub]partitioned by a single column.
> - Conditions are a form of `[sub]partitition_key CMP` constant, where `CMP` can be:
>
>   `=, >, >=, <, <=, IN`
>
> - The `partition_key` condition is `AND`ed with other non-partition conditions.

Partition Management Operations (PMOP) and IVF Indexes

These are the PMOP possibilities and restrictions for local IVF indexes:

- `ALTER TABLE TRUNCATE [sub]partition <partition_name>`

  `ALTER TABLE DROP [sub]partition <partition_name>`

  These operations are supported with all partition schemes: `RANGE`, `HASH`, `LIST`. However, all corresponding IVF index partitions are marked as `UNUSABLE` after the operation.

- `ALTER TABLE ADD [sub]partition <partition_name>`

  If the base table is partitioned by `RANGE` or `LIST`, then the operation is supported. However, all corresponding IVF indexes are marked as `UNUSABLE` after the operation. If the table is partitioned by `HASH`, then the operation fails if there are any local IVF indexes on the base table.

- All other PMOP operations on the base table are not supported, such as `ALTER TABLE SPLIT/MERGE/MOVE/EXCHANGE/COALESCE`.

  These `ALTER TABLE` statements will fail if there are any local IVF indexes on the table being altered.

- `ALTER INDEX` on local IVF indexes is not supported.

Experiment with LOCAL IVF Vector Indexes

You can start experimenting with `LOCAL` IVF indexes using the following code. This is not a scenario but merely a series of SQL commands to help you get started on your own testing scenarios.

1. **Create the base table using the partition scheme of your choice:** *`RANGE` partitioning:*

```
DROP TABLE sales_data PURGE;


CREATE TABLE sales_data
(
    product_id NUMBER,
    customer_id NUMBER,
    sale_date DATE,
    amount_sold NUMBER,
    vec vector(8),
```

```
    region VARCHAR2(20)
)
PARTITION BY RANGE (product_id)
(
    PARTITION sales_1 VALUES LESS THAN (100),
    PARTITION sales_2 VALUES LESS THAN (200),
    PARTITION sales_3 VALUES LESS THAN (300),
    PARTITION sales_4 VALUES LESS THAN (400),
    PARTITION sales_5 VALUES LESS THAN (500),
    PARTITION sales_6 VALUES LESS THAN (600),
    PARTITION sales_7 VALUES LESS THAN (700),
    PARTITION sales_8 VALUES LESS THAN (800),
    PARTITION sales_9 VALUES LESS THAN (900),
    PARTITION sales_10 VALUES LESS THAN (1000),
    PARTITION sales_default VALUES LESS THAN (1000000)
);
```

LIST partitioning:

```
DROP TABLE sales_data PURGE;

CREATE TABLE sales_data
(
    product_id   NUMBER,
    customer_id  NUMBER,
    sale_date    DATE,
    amount_sold  NUMBER,
    vec          VECTOR(8),
    region       VARCHAR2(20)
)
PARTITION BY LIST (region)
(
    PARTITION RegionA_Partition VALUES
('RegionA1','RegionA2','RegionA3','RegionA4','RegionA5'),
    PARTITION RegionB_Partition VALUES
('RegionB1','RegionB2','RegionB3','RegionB4','RegionB5'),
    PARTITION RegionC_Partition VALUES
('RegionC1','RegionC2','RegionC3','RegionC4','RegionC5'),
    PARTITION RegionD_Partition VALUES
('RegionD1','RegionD2','RegionD3','RegionD4','RegionD5'),
    PARTITION RegionE_Partition VALUES
('RegionE1','RegionE2','RegionE3','RegionE4','RegionE5'),
    PARTITION RegionF_Partition VALUES
('RegionF1','RegionF2','RegionF3','RegionF4','RegionF5'),
    PARTITION RegionG_Partition VALUES
('RegionG1','RegionG2','RegionG3','RegionG4','RegionG5'),
    PARTITION RegionH_Partition VALUES
('RegionH1','RegionH2','RegionH3','RegionH4','RegionH5'),
    PARTITION RegionI_Partition VALUES
('RegionI1','RegionI2','RegionI3','RegionI4','RegionI5'),
    PARTITION RegionJ_Partition VALUES
('RegionJ1','RegionJ2','RegionJ3','RegionJ4','RegionJ5'),
    PARTITION Other_Region_Partition VALUES (DEFAULT)
);
```

HASH partitioning:

```
DROP TABLE sales_data PURGE;

CREATE TABLE sales_data
(
    product_id   NUMBER,
    customer_id  NUMBER,
    sale_date    DATE,
    amount_sold  NUMBER,
    vec          VECTOR(8),
    region       VARCHAR2(20)
)
PARTITION BY HASH (product_id)
PARTITIONS 10;
```

2. **Use this procedure to randomly insert data into the SALES_DATA table:**

```
CREATE OR REPLACE PROCEDURE insert_sales_data(numRows IN INTEGER,
maxProductId IN INTEGER) AS
  TYPE vec_array IS VARRAY(8) OF NUMBER;
BEGIN
  DBMS_RANDOM.INITIALIZE(100);
    FOR i IN 1..numRows LOOP
      INSERT INTO sales_data (product_id, customer_id, sale_date,
amount_sold, vec, region)
        VALUES (round(DBMS_RANDOM.VALUE(1, MaxProductId)),    -- Random
product_id between 1 and 1000
                round(DBMS_RANDOM.VALUE(1, 10000)),   -- Random
customer_id between 1 and 10000
                (DATE '2024-05-10' - DBMS_RANDOM.VALUE(1, 1460)), --
Random sale_date within the last 4 years
                DBMS_RANDOM.VALUE(10, 10000),  -- Random amount_sold
between 10 and 1000
                '[' ||
                    to_char(DBMS_RANDOM.VALUE(0, 1000)) || ',' ||
                    to_char(DBMS_RANDOM.VALUE(0, 1000)) || ',' ||
                    to_char(DBMS_RANDOM.VALUE(0, 1000)) || ',' ||
                    to_char(DBMS_RANDOM.VALUE(0, 1000)) || ',' ||
                    to_char(DBMS_RANDOM.VALUE(0, 1000)) || ',' ||
                    to_char(DBMS_RANDOM.VALUE(0, 1000)) || ',' ||
                    to_char(DBMS_RANDOM.VALUE(0, 1000)) || ',' ||
                    to_char(DBMS_RANDOM.VALUE(0, 1000)) ||
                ']',
                CASE MOD(i, 10)
                    WHEN 0 THEN 'RegionA' || MOD(i,5)
                    WHEN 1 THEN 'RegionB' || MOD(i,5)
                    WHEN 2 THEN 'RegionC' || MOD(i,5)
                    WHEN 3 THEN 'RegionD' || MOD(i,5)
                    WHEN 4 THEN 'RegionE' || MOD(i,5)
                    WHEN 5 THEN 'RegionF' || MOD(i,5)
                    WHEN 6 THEN 'RegionG' || MOD(i,5)
                    WHEN 7 THEN 'RegionH' || MOD(i,5)
                    WHEN 8 THEN 'RegionI' || MOD(i,5)
                    ELSE        'RegionJ' || MOD(i,5)
                END);
```

```
        IF MOD(i, 10000) = 0 THEN
            COMMIT;
        END IF;
    END LOOP;
    COMMIT;
END insert_sales_data;
/

EXEC insert_sales_data(20000, 501);
```

3. **Create a `LOCAL` IVF index on the VEC column of the `SALES_DATA` table:**

```
CREATE VECTOR INDEX vidxivf ON sales_data(vec)
ORGANIZATION NEIGHBOR PARTITIONS
WITH TARGET ACCURACY 95
DISTANCE EUCLIDEAN PARAMETERS(TYPE IVF, NEIGHBOR PARTITION 20) LOCAL;
```

4. **Check all index partitions state before and after running PMOP commands:**

```
SELECT INDEX_NAME, PARTITION_NAME, STATUS
FROM USER_IND_PARTITIONS
WHERE index_name LIKE upper('vidxivf')
ORDER BY 1, 2;
```

5. **Test the following `ALTER TABLE` commands to see what is supported and what is not:**
If the table is `RANGE` partitioned:

```
ALTER TABLE sales_data ADD PARTITION sales_1000000 VALUES LESS THAN
(2000000);

ALTER TABLE sales_data
SPLIT PARTITION sales_2
AT (150)
INTO (
      PARTITION sales_21,
      PARTITION sales_22
);

ALTER TABLE sales_data DROP PARTITION sales_3;
```

If the table is `LIST` partitioned:

```
ALTER TABLE sales_data
SPLIT PARTITION RegionE_Partition VALUES ('RegionE1', 'RegionE2',
'RegionE3')
INTO
  ( PARTITION RegionE1_Partition,
    PARTITION RegionE2_Partition
  );

ALTER TABLE sales_data DROP PARTITION RegionB_Partition;
```

If the table is `HASH` partitioned:

```
ALTER TABLE sales_data ADD PARTITION;

SELECT partition_name FROM user_tab_partitions WHERE
table_name='SALES_DATA';

ALTER TABLE sales_data COALESCE PARTITION;
```

3.3.2 Global Partitioned Indexes

In a global partitioned index, the keys in a particular index partition may refer to rows stored in multiple underlying table partitions or subpartitions.

A global index can be range or hash partitioned, though it can be defined on any type of partitioned table. A global index is created by specifying the `GLOBAL` attribute. The database administrator is responsible for defining the initial partitioning of a global index at creation and for maintaining the partitioning over time. Index partitions can be merged or split as necessary.

Normally, a global index is not equipartitioned with the underlying table. There is nothing to prevent an index from being equipartitioned with the underlying table, but Oracle does not take advantage of the equipartitioning when generating query plans or executing partition maintenance operations. So an index that is equipartitioned with the underlying table should be created as `LOCAL`.

A global partitioned index contains a single B-tree with entries for all rows in all partitions. Each index partition may contain keys that refer to many different partitions or subpartitions in the table.

The highest partition of a global index must have a partition bound that includes all values that are `MAXVALUE`. This insures that all rows in the underlying table can be represented in the index.

The following topics are discussed:

- Prefixed and Nonprefixed Global Partitioned Indexes
- Management of Global Partitioned Indexes

3.3.2.1 Prefixed and Nonprefixed Global Partitioned Indexes

A global partitioned index is prefixed if it is partitioned on a left prefix of the index columns.

A global partitioned index is nonprefixed if it is not partitioned on a left prefix of the index columns. Oracle does not support global nonprefixed partitioned indexes. See Figure 3-6 for an example.

Global prefixed partitioned indexes can be unique or nonunique. Nonpartitioned indexes are treated as global prefixed nonpartitioned indexes.

3.3.2.2 Management of Global Partitioned Indexes

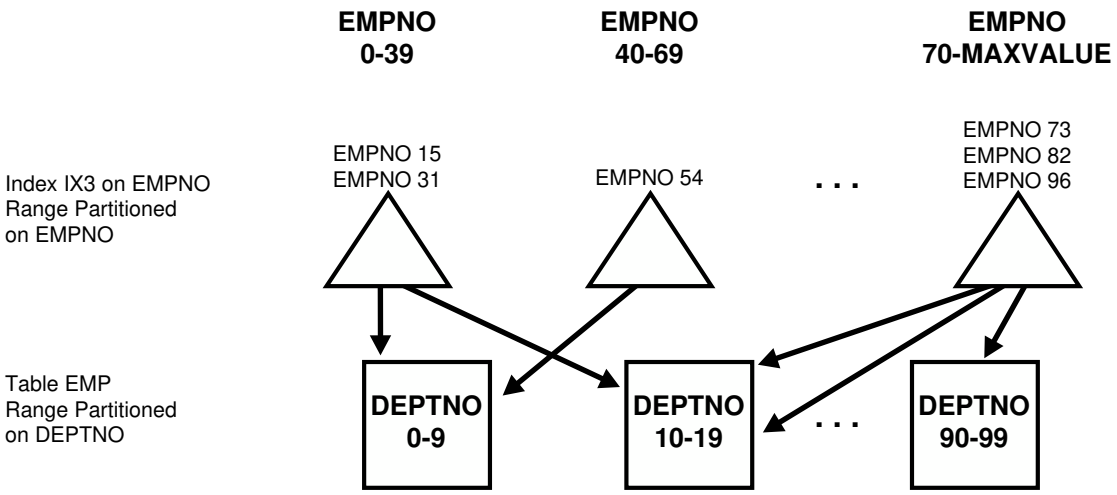
Management of global partitioned indexes presents several challenges.

Global partitioned indexes are harder to manage than local indexes because of the following:

- When the data in an underlying table partition is moved or removed (`SPLIT`, `MOVE`, `DROP`, or `TRUNCATE`), all partitions of a global index are affected. Consequently global indexes do not support partition independence.

- When an underlying table partition or subpartition is recovered to a point in time, all corresponding entries in a global index must be recovered to the same point in time. Because these entries may be scattered across all partitions or subpartitions of the index, mixed with entries for other partitions or subpartitions that are not being recovered, there is no way to accomplish this except by re-creating the entire global index.

Figure 3-6 Global Prefixed Partitioned Index



3.3.3 Summary of Partitioned Index Types

A summary of partitioned index types is provided in this topic.

Table 3-1 summarizes the types of partitioned indexes that Oracle supports. The key points are:

- If an index is local, then it is equipartitioned with the underlying table. Otherwise, it is global.
- A prefixed index is partitioned on a left prefix of the index columns. Otherwise, it is nonprefixed.

Table 3-1 Types of Partitioned Indexes

Type of Index	Index Equipartitioned with Table	Index Partitioned on Left Prefix of Index Columns	UNIQUE Attribute Allowed	Example: Table Partitioning Key	Example: Index Columns	Example: Index Partitioning Key
Local Prefixed (any partitioning method)	Yes	Yes	Yes	A	A, B	A
Local Nonprefixed (any partitioning method)	Yes	No	Yes[1]	A	B, A	A
Global Prefixed (range partitioning only)	No[2]	Yes	Yes	A	B	B

1 For a unique local nonprefixed index, the partitioning key must be a subset of the index key and cannot be a partial index.

[2] Although a global partitioned index may be equipartitioned with the underlying table, Oracle does not take advantage of the partitioning or maintain equipartitioning after partition maintenance operations such as DROP or SPLIT PARTITION.

3.3.4 The Importance of Nonprefixed Indexes

Nonprefixed indexes are important because they are particularly useful in historical databases.

In a table containing historical data, it is common for an index to be defined on one column to support the requirements of fast access by that column. However, the index can also be partitioned on another column (the same column as the underlying table) to support the time interval for rolling out old data and rolling in new data.

Consider a `sales` table partitioned by week. It contains a year's worth of data, divided into 13 partitions. It is range partitioned on `week_no`, four weeks to a partition. You might create a nonprefixed local index `sales_ix` on `sales`. The `sales_ix` index is defined on `acct_no` because there are queries that need fast access to the data by account number. However, it is partitioned on `week_no` to match the `sales` table. Every four weeks, the oldest partitions of `sales` and `sales_ix` are dropped and new ones are added.

3.3.5 Performance Implications of Prefixed and Nonprefixed Indexes

There are performance implications of prefixed and nonprefixed indexes.

With a prefixed index, the likelihood to get partition pruning is much higher than with a non-prefixed index. If a column is part of an index, then you can assume that the column is used as a filter predicate, which automatically means some level of pruning when a filtered column is a prefixed column. This result suggests that it is usually less expensive to probe into a prefixed index than to probe into a nonprefixed index. If an index is prefixed (either local or global) and Oracle is presented with a predicate involving the index columns, then partition pruning can restrict application of the predicate to a subset of the index partitions.

For example, in #unique_120/unique_120_Connect_42_I1008832, if the predicate is `deptno=15`, the optimizer knows to apply the predicate only to the second partition of the index. (If the predicate involves a bind variable, the optimizer does not know exactly which partition but it may still know there is only one partition involved, in which case at run time, only one index partition is accessed.)

When an index is nonprefixed, Oracle often has to apply a predicate involving the index columns to all `N` index partitions. This is required to look up a single key, or to do an index range scan. For a range scan, Oracle must also combine information from `N` index partitions. For example, in Figure 3-5, a local index is partitioned on `chkdate` with an index key on `acctno`. If the predicate is `acctno=31`, Oracle probes all 12 index partitions.

Of course, if there is also a predicate on the partitioning columns, then multiple index probes might not be necessary. Oracle takes advantage of the fact that a local index is equipartitioned with the underlying table to prune partitions based on the partition key. For example, if the predicate in Figure 3-5 is `chkdate<3/97`, Oracle only has to probe two partitions.

So for a nonprefixed index, if the partition key is a part of the `WHERE` clause but not of the index key, then the optimizer determines which index partitions to probe based on the underlying table partition.

When many queries and DML statements using keys of local, nonprefixed, indexes have to probe all index partitions, this effectively reduces the degree of partition independence provided by such indexes.

Table 3-2 Comparing Prefixed Local, Nonprefixed Local, and Global Indexes

Index Characteristics	Prefixed Local	Nonprefixed Local	Global
Unique possible?	Yes	Yes	Yes. Must be global if using indexes on columns other than the partitioning columns
Manageability	Easy to manage	Easy to manage	Harder to manage
OLTP	Good	Bad	Good
Long Running (DSS)	Good	Good	Not Good

3.3.6 Advanced Index Compression With Partitioned Indexes

Advanced index compression with partitioned indexes can reduce the storage requirements for indexes.

Creating an index using advanced index compression reduces the size of all supported unique and non-unique indexes. Advanced index compression improves the compression ratios significantly while still providing efficient access to the indexes. Advanced compression works well on all supported indexes, including those indexes that are not good candidates for prefix compression.

For a partitioned index, you can specify the compression type on a partition by partition basis. You can also specify advanced index compression on index partitions even when the parent index is not compressed.

The following example shows a mixture of compression attributes on the partition indexes.

```
CREATE INDEX my_test_idx ON test(a, b) COMPRESS ADVANCED HIGH LOCAL
   (PARTITION p1 COMPRESS ADVANCED LOW,
    PARTITION p2 COMPRESS,
    PARTITION p3,
    PARTITION p4 NOCOMPRESS);
```

The following example shows advanced index compression support on partitions where the parent index is not compressed.

```
CREATE INDEX my_test_idx ON test(a, b) NOCOMPRESS LOCAL
   (PARTITION p1 COMPRESS ADVANCED LOW,
    PARTITION p2 COMPRESS ADVANCED HIGH,
    PARTITION p3);
```

> **See Also:**
>
> *Oracle Database Administrator's Guide* for information about advanced index compression

3.3.7 Guidelines for Partitioning Indexes

There are several guidelines for partitioning indexes.

When deciding how to partition indexes on a table, consider the mix of applications that must access the table. There is a trade-off between performance and availability and manageability. Here are some guidelines you should consider:

- For OLTP applications:
  - Global indexes and local prefixed indexes provide better performance than local nonprefixed indexes because they minimize the number of index partition probes.
  - Local indexes support more availability when there are partition or subpartition maintenance operations on the table. Local nonprefixed indexes are very useful for historical databases.
- For DSS applications, local nonprefixed indexes can improve performance because many index partitions can be scanned in parallel by range queries on the index key.

  For example, a query using the predicate "`acctno` between 40 and 45" on the table `checks` of Figure 3-5 causes parallel scans of all the partitions of the nonprefixed index `ix3`. On the other hand, a query using the predicate `deptno BETWEEN 40 AND 45` on the table `deptno` of #unique_120/unique_120_Connect_42_I1008832 cannot be parallelized because it accesses a single partition of the prefixed index `ix1`.
- For historical tables, indexes should be local if possible. This limits the effect of regularly scheduled drop partition operations.
- Unique indexes on columns other than the partitioning columns must be global because unique local nonprefixed indexes whose keys do not contain the partitioning key are not supported.
- Unusable indexes do not consume space.

> **See Also:**
>
> *Oracle Database Administrator's Guide* for information about guidelines for managing tables

3.3.8 Physical Attributes of Index Partitions

The physical attributes of index partitions are described in this topic.

Default physical attributes are initially specified when a `CREATE INDEX` statement creates a partitioned index. Because there is no segment corresponding to the partitioned index itself, these attributes are only used in derivation of physical attributes of member partitions. Default physical attributes can later be modified using `ALTER INDEX MODIFY DEFAULT ATTRIBUTES`.

Physical attributes of partitions created by `CREATE INDEX` are determined as follows:

- Values of physical attributes specified (explicitly or by default) for the index are used whenever the value of a corresponding partition attribute is not specified. Handling of the `TABLESPACE` attribute of partitions of a `LOCAL` index constitutes an important exception to this rule in that without a user-specified `TABLESPACE` value (at both partition and index levels), the value of the physical attribute of the corresponding partition of the underlying table is used.
- Physical attributes (other than `TABLESPACE`, as explained in the preceding) of partitions of local indexes created during processing `ALTER TABLE ADD PARTITION` are set to the default physical attributes of each index.

Physical attributes (other than TABLESPACE) of index partitions created by ALTER TABLE SPLIT PARTITION are determined as follows:

- Values of physical attributes of the index partition being split are used.

Physical attributes of an existing index partition can be modified by ALTER INDEX MODIFY PARTITION and ALTER INDEX REBUILD PARTITION. Resulting attributes are determined as follows:

- Values of physical attributes of the partition before the statement was issued are used whenever a new value is not specified. The ALTER INDEX REBUILD PARTITION SQL statement can change the tablespace in which a partition resides.

Physical attributes of global index partitions created by ALTER INDEX SPLIT PARTITION are determined as follows:

- Values of physical attributes of the partition being split are used whenever a new value is not specified.
- Physical attributes of all partitions of an index (along with default values) may be modified by ALTER INDEX, for example, ALTER INDEX indexname NOLOGGING changes the logging mode of all partitions of indexname to NOLOGGING.

> **See Also:**
>
> Partition Administration for more detailed examples of adding partitions and examples of rebuilding indexes

3.4 Partitioning and Table Compression

Compression can be performed on several partitions or a complete partitioned heap-organized table.

You can do this compression by either defining a complete partitioned table as being compressed, or by defining it on a per-partition level. Partitions without a specific declaration inherit the attribute from the table definition or, if nothing is specified on table level, from the tablespace definition.

The decision whether a partition should be compressed or uncompressed adheres to the same rules as a nonpartitioned table. However, due to partitioning to separate data logically into distinct partitions, such a partitioned table is an ideal candidate for compressing parts of the data (partitions). For example, it is beneficial in all rolling window operations as a intermediate stage before aging out old data. With data segment compression, you can keep more old data online, minimizing the burden of additional storage consumption.

You can also change any existing uncompressed table partition later on, add new compressed and uncompressed partitions, or change the compression attribute as part of any partition maintenance operation that requires data movement, such as MERGE PARTITION, SPLIT PARTITION, or MOVE PARTITION. The partitions can contain data or can be empty.

The access and maintenance of a partially or fully compressed partitioned table are the same as for a fully uncompressed partitioned table. Everything that applies to fully uncompressed partitioned tables is also valid for partially or fully compressed partitioned tables.

The following topics are discussed:

- Table Compression and Bitmap Indexes

- Example of Table Compression and Partitioning

> **See Also:**
>
> - *Oracle Database Data Warehousing Guide* for a generic discussion of data warehousing optimizations and techniques
> - *Oracle Database Administrator's Guide* for information about guidelines for managing tables
> - *Oracle Database Performance Tuning Guide* for estimating the compression factor

3.4.1 Table Compression and Bitmap Indexes

There are several necessary steps before using compression on partitioned tables with bitmap indexes.

To use table compression on partitioned tables with bitmap indexes, you must do the following before you introduce the compression attribute for the first time:

1. Mark bitmap indexes unusable.
2. Set the compression attribute.
3. Rebuild the indexes.

The first time you make a compressed partition part of an existing, fully uncompressed partitioned table, you must either drop all existing bitmap indexes or mark them `UNUSABLE` before adding a compressed partition. This must be done irrespective of whether any partition contains any data. It is also independent of the operation that causes one or more compressed partitions to become part of the table. This does not apply to a partitioned table having B-tree indexes only.

This rebuilding of the bitmap index structures is necessary to accommodate the potentially higher number of rows stored for each data block with table compression enabled. Enabling table compression must be done only for the first time. All subsequent operations, whether they affect compressed or uncompressed partitions, or change the compression attribute, behave identically for uncompressed, partially compressed, or fully compressed partitioned tables.

To avoid the recreation of any bitmap index structure, Oracle recommends creating every partitioned table with at least one compressed partition whenever you plan to partially or fully compress the partitioned table in the future. This compressed partition can stay empty or even can be dropped after the partition table creation.

Having a partitioned table with compressed partitions can lead to slightly larger bitmap index structures for the uncompressed partitions. The bitmap index structures for the compressed partitions, however, are usually smaller than the appropriate bitmap index structure before table compression. This highly depends on the achieved compression rates.

> **Note:**
>
> Oracle Database raises an error if compression is introduced to an object for the first time and there are usable bitmap index segments.

3.4.2 Example of Table Compression and Partitioning

Examples of table compression with partitioned tables are described in this topic.

The following statement moves and compresses an existing partition `sales_q1_1998` of table `sales`:

```
ALTER TABLE sales
  MOVE PARTITION sales_q1_1998 TABLESPACE ts_arch_q1_1998 COMPRESS;
```

Alternatively, you could choose Hybrid Columnar Compression (HCC), as in the following:

```
ALTER TABLE sales
  MOVE PARTITION sales_q1_1998 TABLESPACE ts_arch_q1_1998
  COMPRESS FOR ARCHIVE LOW;
```

If you use the `MOVE` statement, then the local indexes for partition `sales_q1_1998` become unusable. You must rebuild them afterward, as follows:

```
ALTER TABLE sales
  MODIFY PARTITION sales_q1_1998 REBUILD UNUSABLE LOCAL INDEXES;
```

You can also include the `UPDATE INDEXES` clause in the `MOVE` statement in order for the entire operation to be completed automatically without any negative effect on users accessing the table.

The following statement merges two existing partitions into a new, compressed partition, residing in a separate tablespace. The local bitmap indexes have to be rebuilt afterward, as in the following:

```
ALTER TABLE sales MERGE PARTITIONS sales_q1_1998, sales_q2_1998
  INTO PARTITION sales_1_1998 TABLESPACE ts_arch_1_1998
  COMPRESS FOR OLTP UPDATE INDEXES;
```

> **See Also:**
>
> - Partition Administration for more details and examples about partition management operations
> - *Oracle Database Performance Tuning Guide* for details regarding how to estimate the compression ratio when using table compression
> - *Oracle Database SQL Language Reference* for the SQL syntax
> - *Oracle Database Concepts* for more information about Hybrid Columnar Compression. Hybrid Columnar Compression is a feature of certain Oracle storage systems.

3.5 Recommendations for Choosing a Partitioning Strategy

Review these recommendations based on performance considerations when choosing a partitioning strategy.

The following topics provide recommendations for choosing a partitioning strategy:

- When to Use Range or Interval Partitioning
- When to Use Hash Partitioning
- When to Use List Partitioning
- When to Use Composite Partitioning
- When to Use Interval Partitioning
- When to Use Reference Partitioning
- When to Partition on Virtual Columns
- Considerations When Using Read-Only Tablespaces

3.5.1 When to Use Range or Interval Partitioning

Range and interval partitioning are useful when organizing similar data, especially date and time data.

Range partitioning is a convenient method for partitioning historical data. The boundaries of range partitions define the ordering of the partitions in the tables or indexes.

Interval partitioning is an extension to range partitioning in which, beyond a point in time, partitions are defined by an interval. Interval partitions are automatically created by the database when data is inserted into the partition.

Range or interval partitioning is often used to organize data by time intervals on a column of type `DATE`. Thus, most SQL statements accessing range partitions focus on time frames. An example of this is a SQL statement similar to "select data from a particular period in time". In such a scenario, if each partition represents data for one month, the query "find data of month 06-DEC" must access only the December partition of year 2006. This reduces the amount of data scanned to a fraction of the total data available, an optimization method called partition pruning.

Range partitioning is also ideal when you periodically load new data and purge old data, because it is easy to add or drop partitions. For example, it is common to keep a rolling window of data, keeping the past 36 months' worth of data online. Range partitioning simplifies this process. To add data from a new month, you load it into a separate table, clean it, index it, and then add it to the range-partitioned table using the `EXCHANGE PARTITION` statement, all while the original table remains online. After you add the new partition, you can drop the trailing month with the `DROP PARTITION` statement. The alternative to using the `DROP PARTITION` statement can be to archive the partition and make it read only, but this works only when your partitions are in separate tablespaces. You can also implement a rolling window of data using inserts into the partitioned table.

Interval partitioning provides an easy way for interval partitions to be automatically created as data arrives. Interval partitions can also be used for all other partition maintenance operations.

In conclusion, consider using range or interval partitioning when:

- Very large tables are frequently scanned by a range predicate on a good partitioning column, such as `ORDER_DATE` or `PURCHASE_DATE`. Partitioning the table on that column enables partition pruning.
- You want to maintain a rolling window of data.
- You cannot complete administrative operations, such as backup and restore, on large tables in an allotted time frame, but you can divide them into smaller logical pieces based on the partition range column.

Example 3-5 creates the table `salestable` for a period of two years, 2005 and 2006, and partitions it by range according to the column `s_salesdate` to separate the data into eight quarters, each corresponding to a partition. Future partitions are created automatically through the monthly interval definition. Interval partitions are created in the provided list of tablespaces in a round-robin manner. Analysis of sales figures by a short interval can take advantage of partition pruning. The sales table also supports a rolling window approach.

Example 3-5 Creating a table with range and interval partitioning

```
CREATE TABLE salestable
  (s_productid  NUMBER,
   s_saledate   DATE,
   s_custid     NUMBER,
   s_totalprice NUMBER)
PARTITION BY RANGE(s_saledate)
INTERVAL(NUMTOYMINTERVAL(1,'MONTH')) STORE IN (tbs1,tbs2,tbs3,tbs4)
 (PARTITION sal05q1 VALUES LESS THAN (TO_DATE('01-APR-2005', 'DD-MON-YYYY')) TABLESPACE tbs1,
  PARTITION sal05q2 VALUES LESS THAN (TO_DATE('01-JUL-2005', 'DD-MON-YYYY')) TABLESPACE tbs2,
  PARTITION sal05q3 VALUES LESS THAN (TO_DATE('01-OCT-2005', 'DD-MON-YYYY')) TABLESPACE tbs3,
  PARTITION sal05q4 VALUES LESS THAN (TO_DATE('01-JAN-2006', 'DD-MON-YYYY')) TABLESPACE tbs4,
  PARTITION sal06q1 VALUES LESS THAN (TO_DATE('01-APR-2006', 'DD-MON-YYYY')) TABLESPACE tbs1,
  PARTITION sal06q2 VALUES LESS THAN (TO_DATE('01-JUL-2006', 'DD-MON-YYYY')) TABLESPACE tbs2,
  PARTITION sal06q3 VALUES LESS THAN (TO_DATE('01-OCT-2006', 'DD-MON-YYYY')) TABLESPACE tbs3,
  PARTITION sal06q4 VALUES LESS THAN (TO_DATE('01-JAN-2007', 'DD-MON-YYYY')) TABLESPACE tbs4);
```

> **See Also:**
>
> Partition Administration for more information about the partition maintenance operations on interval partitions

3.5.2 When to Use Hash Partitioning

Hash partitioning is useful for randomly distributing data across partitions based on a hashing algorithm, rather than grouping similar data.

There are times when it is not obvious in which partition data should reside, although the partitioning key can be identified. Rather than group similar data, there are times when it is desirable to distribute data such that it does not correspond to a business or a logical view of the data, as it does in range partitioning. With hash partitioning, a row is placed into a partition based on the result of passing the partitioning key into a hashing algorithm.

Using this approach, data is randomly distributed across the partitions rather than grouped. This is a good approach for some data, but may not be an effective way to manage historical data. However, hash partitions share some performance characteristics with range partitions. For example, partition pruning is limited to equality predicates. You can also use partition-wise joins, parallel index access, and parallel DML.

As a general rule, use hash partitioning for the following purposes:

- To enable partial or full parallel partition-wise joins with likely equisized partitions.
- To distribute data evenly among the nodes of an MPP platform that uses Oracle Real Application Clusters. Consequently, you can minimize interconnect traffic when processing internode parallel statements.
- To use partition pruning and partition-wise joins according to a partitioning key that is mostly constrained by a distinct value or value list.
- To randomly distribute data to avoid I/O bottlenecks if you do not use a storage management technique that stripes and mirrors across all available devices.

> **Note:**
>
> With hash partitioning, only equality or `IN`-list predicates are supported for partition pruning.

For optimal data distribution, the following requirements should be satisfied:

- Choose a column or combination of columns that is unique or almost unique.
- Create multiple partitions and subpartitions for each partition that is a power of two. For example, 2, 4, 8, 16, 32, 64, 128, and so on.

Example 3-6 creates four hash partitions for the table `sales_hash` using the column `s_productid` as the partitioning key. Parallel joins with the products table can take advantage of partial or full partition-wise joins. Queries accessing sales figures for only a single product or a list of products benefit from partition pruning.

If you do not explicitly specify partition names, but instead you specify the number of hash partitions, then Oracle automatically generates internal names for the partitions. Also, you can use the `STORE IN` clause to assign hash partitions to tablespaces in a round-robin manner.

Example 3-6 Creating a table with hash partitioning

```
CREATE TABLE sales_hash
  (s_productid  NUMBER,
   s_saledate   DATE,
   s_custid     NUMBER,
   s_totalprice NUMBER)
PARTITION BY HASH(s_productid)
( PARTITION p1 TABLESPACE tbs1
, PARTITION p2 TABLESPACE tbs2
, PARTITION p3 TABLESPACE tbs3
, PARTITION p4 TABLESPACE tbs4
);
```

> **See Also:**
>
> - Partition-Wise Operations for information about part-wise joins
> - Storage Management for VLDBs for more information about managing storage for VLDBs
> - Partition Administration for more examples on creating hash-partitioned tables
> - *Oracle Database SQL Language Reference* for partitioning syntax

3.5.3 When to Use List Partitioning

List partitioning is useful to explicitly map rows to partitions based on discrete values.

In Example 3-7, all the customers for states Oregon and Washington are stored in one partition and customers in other states are stored in different partitions. Account managers who analyze their accounts by region can take advantage of partition pruning.

Example 3-7 Creating a table with list partitioning

```
CREATE TABLE accounts
( id             NUMBER
, account_number NUMBER
, customer_id    NUMBER
, branch_id      NUMBER
, region         VARCHAR(2)
, status         VARCHAR2(1)
)
PARTITION BY LIST (region)
( PARTITION p_northwest VALUES ('OR', 'WA')
, PARTITION p_southwest VALUES ('AZ', 'UT', 'NM')
, PARTITION p_northeast VALUES ('NY', 'VM', 'NJ')
, PARTITION p_southeast VALUES ('FL', 'GA')
, PARTITION p_northcentral VALUES ('SD', 'WI')
, PARTITION p_southcentral VALUES ('OK', 'TX')
);
```

3.5.4 When to Use Composite Partitioning

Composite partitioning offers the benefits of partitioning on multiple dimensions.

From a performance perspective you can take advantage of partition pruning on one or two dimensions depending on the SQL statement, and you can take advantage of the use of full or partial partition-wise joins on either dimension.

You can take advantage of parallel backup and recovery of a single table. Composite partitioning also increases the number of partitions significantly, which may be beneficial for efficient parallel execution. From a manageability perspective, you can implement a rolling window to support historical data and still partition on another dimension if many statements can benefit from partition pruning or partition-wise joins.

You can split backups of your tables and you can decide to store data differently based on identification by a partitioning key. For example, you may decide to store data for a specific product type in a read-only, compressed format, and keep other product type data uncompressed.

The database stores every subpartition in a composite partitioned table as a separate segment. Thus, the subpartitions may have properties that differ from the properties of the table or from the partition to which the subpartitions belong.

The following topics are discussed:

- When to Use Range or Interval Partitioning
- When to Use Hash Partitioning
- When to Use List Partitioning
- When to Use Composite Partitioning
- When to Use Interval Partitioning
- When to Use Reference Partitioning
- When to Partition on Virtual Columns

> **See Also:**
>
> *Oracle Database SQL Language Reference* for details regarding syntax and restrictions

3.5.4.1 When to Use Composite Range-Hash Partitioning

Composite range-hash partitioning is particularly common for tables that store history, are very large consequently, and are frequently joined with other large tables.

For these types of tables (typical of data warehouse systems), composite range-hash partitioning provides the benefit of partition pruning at the range level with the opportunity to perform parallel full or partial partition-wise joins at the hash level. Specific cases can benefit from partition pruning on both dimensions for specific SQL statements.

Composite range-hash partitioning can also be used for tables that traditionally use hash partitioning, but also use a rolling window approach. Over time, data can be moved from one storage tier to another storage tier, compressed, stored in a read-only tablespace, and eventually purged. Information Lifecycle Management (ILM) scenarios often use range partitions to implement a tiered storage approach.

Example 3-8 is an example of a range hash partitioned `page_history` table of an Internet service provider. The table definition is optimized for historical analysis for either specific `client_ip` values (in which case queries benefit from partition pruning) or for analysis across many IP addresses, in which case queries can take advantage of full or partial partition-wise joins.

This example shows the use of interval partitioning. You can use interval partitioning in addition to range partitioning so that interval partitions are created automatically as data is inserted into the table.

Example 3-8 Creating a table with composite range-hash partitioning

```
CREATE TABLE page_history
( id                NUMBER NOT NULL
, url               VARCHAR2(300) NOT NULL
, view_date         DATE NOT NULL
, client_ip         VARCHAR2(23) NOT NULL
, from_url          VARCHAR2(300)
```

```
, to_url            VARCHAR2(300)
, timing_in_seconds NUMBER
) PARTITION BY RANGE(view_date) INTERVAL (NUMTODSINTERVAL(1,'DAY'))
SUBPARTITION BY HASH(client_ip)
SUBPARTITIONS 32
(PARTITION p0 VALUES LESS THAN (TO_DATE('01-JAN-2006','dd-MON-yyyy')))
PARALLEL 32 COMPRESS;
```

> **See Also:**
>
> Managing and Maintaining Time-Based Information for more detail on Information Lifecycle Management (ILM) and implementing tiered storage using partitioning

3.5.4.2 When to Use Composite Range-List Partitioning

Composite range-list partitioning is commonly used for large tables that store historical data and are commonly accessed on multiple dimensions.

Often the historical view of the data is one access path, but certain business cases add another categorization to the access path. For example, regional account managers are very interested in how many new customers they signed up in their region in a specific time period. ILM and its tiered storage approach is a common reason to create range-list partitioned tables so that older data can be moved and compressed, but partition pruning on the list dimension is still available.

Example 3-9 creates a range-list partitioned `call_detail_records` table. A telecommunication company can use this table to analyze specific types of calls over time. The table uses local indexes on `from_number` and `to_number`.

This example shows the use of interval partitioning. You can use interval partitioning in addition to range partitioning so that interval partitions are created automatically as data is inserted into the table.

Example 3-9 Creating a table with composite range-list partitioning

```
CREATE TABLE call_detail_records
( id NUMBER
, from_number        VARCHAR2(20)
, to_number          VARCHAR2(20)
, date_of_call       DATE
, distance           VARCHAR2(1)
, call_duration_in_s NUMBER(4)
) PARTITION BY RANGE(date_of_call)
INTERVAL (NUMTODSINTERVAL(1,'DAY'))
SUBPARTITION BY LIST(distance)
SUBPARTITION TEMPLATE
( SUBPARTITION local VALUES('L') TABLESPACE tbs1
, SUBPARTITION medium_long VALUES ('M') TABLESPACE tbs2
, SUBPARTITION long_distance VALUES ('D') TABLESPACE tbs3
, SUBPARTITION international VALUES ('I') TABLESPACE tbs4
)
(PARTITION p0 VALUES LESS THAN (TO_DATE('01-JAN-2005','dd-MON-yyyy')))
PARALLEL;

CREATE INDEX from_number_ix ON call_detail_records(from_number)
LOCAL PARALLEL NOLOGGING;
```

```
CREATE INDEX to_number_ix ON call_detail_records(to_number)
LOCAL PARALLEL NOLOGGING;
```

3.5.4.3 When to Use Composite Range-Range Partitioning

Composite range-range partitioning is useful for applications that store time-dependent data on multiple time dimensions.

Often these applications do not use one particular time dimension to access the data, but rather another time dimension, or sometimes both at the same time. For example, a web retailer wants to analyze its sales data based on when orders were placed, and when orders were shipped (handed over to the shipping company).

Other business cases for composite range-range partitioning include ILM scenarios, and applications that store historical data and want to categorize its data by range on another dimension.

Example 3-10 shows a range-range partitioned table `account_balance_history`. A bank may use access to individual subpartitions to contact its customers for low-balance reminders or specific promotions relevant to a certain category of customers.

This example shows the use of interval partitioning. You can use interval partitioning in addition to range partitioning so that interval partitions are created automatically as data is inserted into the table. In this case 7-day (weekly) intervals are created, starting Monday, January 1, 2007.

Example 3-10 Creating a table with composite range-range partitioning

```
CREATE TABLE account_balance_history
( id                 NUMBER NOT NULL
, account_number     NUMBER NOT NULL
, customer_id        NUMBER NOT NULL
, transaction_date   DATE NOT NULL
, amount_credited    NUMBER
, amount_debited     NUMBER
, end_of_day_balance NUMBER NOT NULL
) PARTITION BY RANGE(transaction_date)
INTERVAL (NUMTODSINTERVAL(7,'DAY'))
SUBPARTITION BY RANGE(end_of_day_balance)
SUBPARTITION TEMPLATE
( SUBPARTITION unacceptable VALUES LESS THAN (-1000)
, SUBPARTITION credit VALUES LESS THAN (0)
, SUBPARTITION low VALUES LESS THAN (500)
, SUBPARTITION normal VALUES LESS THAN (5000)
, SUBPARTITION high VALUES LESS THAN (20000)
, SUBPARTITION extraordinary VALUES LESS THAN (MAXVALUE)
)
(PARTITION p0 VALUES LESS THAN (TO_DATE('01-JAN-2007','dd-MON-yyyy')));
```

3.5.4.4 When to Use Composite List-Hash Partitioning

Composite list-hash partitioning is useful for large tables that are usually accessed on one dimension, but (due to their size) still must take advantage of parallel full or partial partition-wise joins on another dimension in joins with other large tables.

Example 3-11 shows a `credit_card_accounts` table. The table is list-partitioned on region in order for account managers to quickly access accounts in their region. The subpartitioning strategy is hash on `customer_id` so that queries against the transactions table, which is subpartitioned on `customer_id`, can take advantage of full partition-wise joins. Joins with the hash partitioned customers table can also benefit from full partition-wise joins. The table has a local bitmap index on the `is_active` column.

Example 3-11 Creating a table with composite list-hash partitioning

```
CREATE TABLE credit_card_accounts
( account_number  NUMBER(16) NOT NULL
, customer_id     NUMBER NOT NULL
, customer_region VARCHAR2(2) NOT NULL
, is_active       VARCHAR2(1) NOT NULL
, date_opened     DATE NOT NULL
) PARTITION BY LIST (customer_region)
SUBPARTITION BY HASH (customer_id)
SUBPARTITIONS 16
( PARTITION emea VALUES ('EU','ME','AF')
, PARTITION amer VALUES ('NA','LA')
, PARTITION apac VALUES ('SA','AU','NZ','IN','CH')
) PARALLEL;

CREATE BITMAP INDEX is_active_bix ON credit_card_accounts(is_active)
LOCAL PARALLEL NOLOGGING;
```

3.5.4.5 When to Use Composite List-List Partitioning

Composite list-list partitioning is useful for large tables that are often accessed on different dimensions.

You can specifically map rows to partitions on those dimensions based on discrete values.

Example 3-12 shows an example of a very frequently accessed `current_inventory` table. The table is constantly updated with the current inventory in the supermarket supplier's local warehouses. Potentially perishable foods are supplied from those warehouses to supermarkets, and it is important to optimize supplies and deliveries. The table has local indexes on `warehouse_id` and `product_id`.

Example 3-12 Creating a table with composite list-list partitioning

```
CREATE TABLE current_inventory
( warehouse_id      NUMBER
, warehouse_region  VARCHAR2(2)
, product_id        NUMBER
, product_category  VARCHAR2(12)
, amount_in_stock   NUMBER
, unit_of_shipping  VARCHAR2(20)
, products_per_unit NUMBER
, last_updated      DATE
) PARTITION BY LIST (warehouse_region)
SUBPARTITION BY LIST (product_category)
SUBPARTITION TEMPLATE
( SUBPARTITION perishable VALUES ('DAIRY','PRODUCE','MEAT','BREAD')
, SUBPARTITION non_perishable VALUES ('CANNED','PACKAGED')
, SUBPARTITION durable VALUES ('TOYS','KITCHENWARE')
)
( PARTITION p_northwest VALUES ('OR', 'WA')
, PARTITION p_southwest VALUES ('AZ', 'UT', 'NM')
, PARTITION p_northeast VALUES ('NY', 'VM', 'NJ')
, PARTITION p_southeast VALUES ('FL', 'GA')
, PARTITION p_northcentral VALUES ('SD', 'WI')
, PARTITION p_southcentral VALUES ('OK', 'TX')
);

CREATE INDEX warehouse_id_ix ON current_inventory(warehouse_id)
LOCAL PARALLEL NOLOGGING;
```

```
CREATE INDEX product_id_ix ON current_inventory(product_id)
LOCAL PARALLEL NOLOGGING;
```

3.5.4.6 When to Use Composite List-Range Partitioning

Composite list-range partitioning is useful for large tables that are accessed on different dimensions.

For the most commonly used dimension, you can specifically map rows to partitions on discrete values. List-range partitioning is commonly used for tables that use range values within a list partition, whereas range-list partitioning is commonly used for a discrete list values within a range partition. List-range partitioning is less commonly used to store historical data, even though equivalent scenarios are all suitable. Range-list partitioning can be implemented using interval-list partitioning, whereas list-range partitioning does not support interval partitioning.

Example 3-13 shows a `donations` table that stores donations in different currencies. The donations are categorized into small, medium, and high, depending on the amount. Due to currency differences, the ranges are different.

Example 3-13 Creating a table with composite list-range partitioning

```
CREATE TABLE donations
( id             NUMBER
, name           VARCHAR2(60)
, beneficiary    VARCHAR2(80)
, payment_method VARCHAR2(30)
, currency       VARCHAR2(3)
, amount         NUMBER
) PARTITION BY LIST (currency)
SUBPARTITION BY RANGE (amount)
( PARTITION p_eur VALUES ('EUR')
  ( SUBPARTITION p_eur_small VALUES LESS THAN (8)
  , SUBPARTITION p_eur_medium VALUES LESS THAN (80)
  , SUBPARTITION p_eur_high VALUES LESS THAN (MAXVALUE)
  )
, PARTITION p_gbp VALUES ('GBP')
  ( SUBPARTITION p_gbp_small VALUES LESS THAN (5)
  , SUBPARTITION p_gbp_medium VALUES LESS THAN (50)
  , SUBPARTITION p_gbp_high VALUES LESS THAN (MAXVALUE)
  )
, PARTITION p_aud_nzd_chf VALUES ('AUD','NZD','CHF')
  ( SUBPARTITION p_aud_nzd_chf_small VALUES LESS THAN (12)
  , SUBPARTITION p_aud_nzd_chf_medium VALUES LESS THAN (120)
  , SUBPARTITION p_aud_nzd_chf_high VALUES LESS THAN (MAXVALUE)
  )
, PARTITION p_jpy VALUES ('JPY')
  ( SUBPARTITION p_jpy_small VALUES LESS THAN (1200)
  , SUBPARTITION p_jpy_medium VALUES LESS THAN (12000)
  , SUBPARTITION p_jpy_high VALUES LESS THAN (MAXVALUE)
  )
, PARTITION p_inr VALUES ('INR')
  ( SUBPARTITION p_inr_small VALUES LESS THAN (400)
  , SUBPARTITION p_inr_medium VALUES LESS THAN (4000)
  , SUBPARTITION p_inr_high VALUES LESS THAN (MAXVALUE)
  )
, PARTITION p_zar VALUES ('ZAR')
  ( SUBPARTITION p_zar_small VALUES LESS THAN (70)
  , SUBPARTITION p_zar_medium VALUES LESS THAN (700)
  , SUBPARTITION p_zar_high VALUES LESS THAN (MAXVALUE)
  )
```

```
, PARTITION p_default VALUES (DEFAULT)
  ( SUBPARTITION p_default_small VALUES LESS THAN (10)
  , SUBPARTITION p_default_medium VALUES LESS THAN (100)
  , SUBPARTITION p_default_high VALUES LESS THAN (MAXVALUE)
  )
) ENABLE ROW MOVEMENT;
```

3.5.5 When to Use Interval Partitioning

Interval partitioning can be used for almost every table that is range partitioned and uses fixed intervals for new partitions.

The database automatically creates interval partitions as data for that partition is inserted. Until this happens, the interval partition exists but no segment is created for the partition.

The benefit of interval partitioning is that you do not need to create your range partitions explicitly. You should consider using interval partitioning unless you create range partitions with different intervals, or if you always set specific partition attributes when you create range partitions. You can specify a list of tablespaces in the interval definition. The database creates interval partitions in the provided list of tablespaces in a round-robin manner.

If you upgrade your application and you use range partitioning or composite range-* partitioning, then you can easily change your existing table definition to use interval partitioning. You cannot manually add partitions to an interval-partitioned table. If you have automated the creation of new partitions, then in the future you must change your application code to prevent the explicit creation of range partitions.

The following SQL statement initiates a change from range partitioning to using monthly interval partitioning on the `sales` table.

```
ALTER TABLE sales SET INTERVAL (NUMTOYMINTERVAL(1,'MONTH'));
```

You cannot use interval partitioning with reference partitioned tables.

Serializable transactions do not work with interval partitioning. Inserting data into a partition of an interval partitioned table that does not have a segment yet causes an error.

3.5.6 When to Use Reference Partitioning

Reference partitioning is useful in certain situations.

Reference partitioning is useful in the following scenarios:

- If you have denormalized, or would denormalize, a column from a parent table into a child table to get partition pruning benefits on both tables.

  For example, your `orders` table stores the `order_date`, but the `order_items` table, which stores one or more items for each order, does not. To get good performance for historical analysis of orders data, you would traditionally duplicate the `order_date` column in the `order_items` table to use partition pruning on the `order_items` table.

  You should consider reference partitioning in such a scenario and avoid having to duplicate the `order_date` column. Queries that join both tables and use a predicate on `order_date` automatically benefit from partition pruning on both tables.

- If two large tables are joined frequently, then the tables are not partitioned on the join key, but you want to take advantage of partition-wise joins.

  Reference partitioning implicitly enables full partition-wise joins.

- If data in multiple tables has a related life cycle, then reference partitioning can provide significant manageability benefits.

  Partition management operations against the parent table are automatically cascaded to its descendents. For example, when you add a partition to the parent table, that addition is automatically propagated to all its descendents.

  To use reference partitioning, you must enable and enforce the foreign key relationship between the parent table and the reference table in place. You can cascade reference-partitioned tables.

  The primary key-foreign key relationship must be enabled all the time and cannot be disabled. Also the relationship cannot be declared as deferred. These are mandatory requirements because the enabled primary key-foreign relationship is required to determine the data placement for the child tables.

3.5.7 When to Partition on Virtual Columns

Partitioning on virtual columns provides more flexibility to partition on a derived column.

Virtual column partitioning enables you to partition on an expression, which may use data from other columns, and perform calculations with these columns. PL/SQL function calls are not supported in virtual column definitions that are to be used as a partitioning key.

Virtual column partitioning supports all partitioning methods, plus performance and manageability features. To get partition pruning benefits, consider using virtual columns if tables are frequently accessed using a predicate that is not directly captured in a column, but can be derived. Traditionally, to get partition pruning benefits, you would have to add a separate column to capture and calculate the correct value and ensure the column is always populated correctly to ensure correct query retrieval.

Example 3-14 shows a `car_rentals` table. The customer's confirmation number contains a two-character country name as the location where the rental car is picked up. Rental car analyses usually evaluate regional patterns, so it makes sense to partition by country.

In this example, the column `country` is defined as a virtual column derived from the confirmation number. The virtual column does not require any storage. As the example illustrates, row movement is supported with virtual columns. The database migrates a row to a different partition if the virtual column evaluates to a different value in another partition.

Example 3-14 Creating a table with virtual columns for partitioning

```
CREATE TABLE car_rentals
( id                  NUMBER NOT NULL
 , customer_id         NUMBER NOT NULL
 , confirmation_number VARCHAR2(12) NOT NULL
 , car_id              NUMBER
 , car_type            VARCHAR2(10)
 , requested_car_type  VARCHAR2(10) NOT NULL
 , reservation_date    DATE NOT NULL
 , start_date          DATE NOT NULL
 , end_date            DATE
 , country as (substr(confirmation_number,9,2))
) PARTITION BY LIST (country)
SUBPARTITION BY HASH (customer_id)
SUBPARTITIONS 16
( PARTITION north_america VALUES ('US','CA','MX')
 , PARTITION south_america VALUES ('BR','AR','PE')
 , PARTITION europe VALUES ('GB','DE','NL','BE','FR','ES','IT','CH')
 , PARTITION apac VALUES ('NZ','AU','IN','CN')
) ENABLE ROW MOVEMENT;
```

3.5.8 Considerations When Using Read-Only Tablespaces

Review these considerations when using read-only tables.

When a referential integrity constraint is defined between parent and child tables, an index is defined on the foreign key, and the tablespace in which that index resides is made read-only, then the integrity check for the constraint is implemented in SQL and not through consistent read buffer access.

The implication of this is if the child is partitioned and if only some child partitions have their indexes in read-only tablespaces and if an insert is made into one nonread-only child segment, then a TM enqueue is acquired on the child table in SX mode.

SX mode is incompatible with S requests, so that if you try to insert into the parent, it is blocked because that insert attempts to acquire an S TM enqueue against the child.

4

Partition Administration

Partition administration is an important task when working with partitioned tables and indexes.

This chapter describes various aspects of creating and maintaining partitioned tables and indexes.

This chapter contains the following sections:

- Specifying Partitioning When Creating Tables and Indexes
- Specifying Composite Partitioning When Creating Tables
- Maintenance Operations Supported on Partitions
- Maintenance Operations for Partitioned Tables and Indexes
- About Dropping Partitioned Tables
- Changing a Nonpartitioned Table into a Partitioned Table
- Managing Hybrid Partitioned Tables
- Viewing Information About Partitioned Tables and Indexes

> **Note:**
>
> Before you attempt to create a partitioned table or index, or perform maintenance operations on any partitioned table, it is recommended that you review the information in Partitioning Concepts.

> **See Also:**
>
> *Oracle Database SQL Language Reference* for general restrictions on partitioning, the exact syntax of the partitioning clauses for creating and altering partitioned tables and indexes, any restrictions on their use, and specific privileges required for creating and altering tables

4.1 Specifying Partitioning When Creating Tables and Indexes

Creating a partitioned table or index is very similar to creating a nonpartitioned table or index.

When creating a partitioned table or index, you include a partitioning clause in the `CREATE TABLE` statement. The partitioning clause, and subclauses, that you include depend upon the type of partitioning you want to achieve.

Partitioning is possible on both regular (heap organized) tables and index-organized tables, except for those containing `LONG` or `LONG RAW` columns. You can create nonpartitioned global indexes, range or hash partitioned global indexes, and local indexes on partitioned tables.

When you create (or alter) a partitioned table, a row movement clause (either `ENABLE ROW MOVEMENT` or `DISABLE ROW MOVEMENT`) can be specified. This clause either enables or disables the migration of a row to a new partition if its key is updated. The default is `DISABLE ROW MOVEMENT`.

You can specify up to a total of 1024K-1 partitions for a single-level partitioned tables, or subpartitions for a composite partitioned table.

Creating automatic list composite partitioned tables and interval subpartitions can save space because these methods only create subpartitions in the presence of data. Deferring subpartition segment creation when creating new partitions on demand ensures that a subpartition segment is only created when the first matching row is inserted.

The following topics present details and examples of creating partitions for the various types of partitioned tables and indexes:

- About Creating Range-Partitioned Tables and Global Indexes
- Creating Range-Interval-Partitioned Tables
- About Creating Hash Partitioned Tables and Global Indexes
- About Creating List-Partitioned Tables
- Creating Reference-Partitioned Tables
- Creating Interval-Reference Partitioned Tables
- Creating a Table Using In-Memory Column Store With Partitioning
- Creating a Table with Read-Only Partitions or Subpartitions
- Creating a Partitioned External Table
- Specifying Partitioning on Key Columns
- Using Virtual Column-Based Partitioning
- Using Table Compression with Partitioned Tables
- Using Key Compression with Partitioned Indexes
- Specifying Partitioning with Segments
- Specifying Partitioning When Creating Index-Organized Tables
- Partitioning Restrictions for Multiple Block Sizes
- Partitioning of Collections in XMLType and Objects

See Also:

- *Oracle Database Administrator's Guide* for information about managing tables
- *Oracle Database SQL Language Reference* for the exact syntax of the partitioning clauses for creating and altering partitioned tables and indexes, any restrictions on their use, and specific privileges required for creating and altering tables
- *Oracle Database SecureFiles and Large Objects Developer's Guide* for information specific to creating partitioned tables containing columns with `LOB`s or other objects stored as `LOB`s
- *Oracle Database Object-Relational Developer's Guide* for information specific to creating tables with object types, nested tables, or VARRAYs

4.1.1 About Creating Range-Partitioned Tables and Global Indexes

The `PARTITION BY RANGE` clause of the `CREATE TABLE` statement specifies that the table or index is to be range-partitioned.

The `PARTITION` clauses identify the individual partition ranges, and the optional subclauses of a `PARTITION` clause can specify physical and other attributes specific to a partition segment. If not overridden at the partition level, partitions inherit the attributes of their underlying table.

The following topics are discussed:

- Creating a Range-Partitioned Table
- Creating a Range-Partitioned Table With More Complexity
- Creating a Range-Partitioned Global Index

4.1.1.1 Creating a Range-Partitioned Table

Use the `PARTITION BY RANGE` clause of the `CREATE TABLE` statement to create a range-partitioned table.

Example 4-1 creates a table of four partitions, one for each quarter of sales. `time_id` is the **partitioning column**, while its values constitute the **partitioning key** of a specific row. The `VALUES LESS THAN` clause determines the **partition bound**: rows with partitioning key values that compare less than the ordered list of values specified by the clause are stored in the partition. Each partition is given a name (`sales_q1_2006`, `sales_q2_2006`, `sales_q3_2006`, `sales_q4_2006`), and each partition is contained in a separate tablespace (`tsa`, `tsb`, `tsc`, `tsd`). A row with `time_id=17-MAR-2006` would be stored in partition `sales_q1_2006`.

Live SQL:

View and run a related example on Oracle Live SQL at *Oracle Live SQL: Creating a Range Partitioned Table*.

Example 4-1 Creating a range-partitioned table

```
CREATE TABLE sales
  ( prod_id       NUMBER(6)
  , cust_id       NUMBER
  , time_id       DATE
  , channel_id    CHAR(1)
  , promo_id      NUMBER(6)
  , quantity_sold NUMBER(3)
  , amount_sold   NUMBER(10,2)
  )
 PARTITION BY RANGE (time_id)
 ( PARTITION sales_q1_2006 VALUES LESS THAN (TO_DATE('01-APR-2006','dd-MON-yyyy'))
    TABLESPACE tsa
 , PARTITION sales_q2_2006 VALUES LESS THAN (TO_DATE('01-JUL-2006','dd-MON-yyyy'))
    TABLESPACE tsb
 , PARTITION sales_q3_2006 VALUES LESS THAN (TO_DATE('01-OCT-2006','dd-MON-yyyy'))
    TABLESPACE tsc
 , PARTITION sales_q4_2006 VALUES LESS THAN (TO_DATE('01-JAN-2007','dd-MON-yyyy'))
    TABLESPACE tsd
 );
```

4.1.1.2 Creating a Range-Partitioned Table With More Complexity

With attributes and storage parameters, more complexity can be added to the creation of a range-partitioned table.

In Example 4-2, storage parameters and a `LOGGING` attribute are specified at the table level. These replace the corresponding defaults inherited from the tablespace level for the table itself, and are inherited by the range partitions. However, because there was little business in the first quarter, the storage attributes for partition `sales_q1_2006` are made smaller. The `ENABLE ROW MOVEMENT` clause is specified to allow the automatic migration of a row to a new partition if an update to a key value is made that would place the row in a different partition.

Example 4-2 Creating a range-partitioned table with LOGGING and ENABLE ROW MOVEMENT

```
CREATE TABLE sales
  ( prod_id       NUMBER(6)
  , cust_id       NUMBER
  , time_id       DATE
  , channel_id    CHAR(1)
  , promo_id      NUMBER(6)
  , quantity_sold NUMBER(3)
  , amount_sold   NUMBER(10,2)
  )
 STORAGE (INITIAL 100K NEXT 50K) LOGGING
 PARTITION BY RANGE (time_id)
 ( PARTITION sales_q1_2006 VALUES LESS THAN (TO_DATE('01-APR-2006','dd-MON-yyyy'))
    TABLESPACE tsa STORAGE (INITIAL 20K NEXT 10K)
 , PARTITION sales_q2_2006 VALUES LESS THAN (TO_DATE('01-JUL-2006','dd-MON-yyyy'))
    TABLESPACE tsb
 , PARTITION sales_q3_2006 VALUES LESS THAN (TO_DATE('01-OCT-2006','dd-MON-yyyy'))
    TABLESPACE tsc
 , PARTITION sales_q4_2006 VALUES LESS THAN (TO_DATE('01-JAN-2007','dd-MON-yyyy'))
    TABLESPACE tsd
 )
 ENABLE ROW MOVEMENT;
```

4.1.1.3 Creating a Range-Partitioned Global Index

The rules for creating range-partitioned global indexes are similar to those for creating range-partitioned tables.

Example 4-3 creates a range-partitioned global index on `sale_month` for the tables created in the previous examples. Each index partition is named but is stored in the default tablespace for the index.

Example 4-3 Creating a range-partitioned global index table

```
CREATE INDEX amount_sold_ix ON sales(amount_sold)
   GLOBAL PARTITION BY RANGE(sale_month)
      ( PARTITION p_100 VALUES LESS THAN (100)
      , PARTITION p_1000 VALUES LESS THAN (1000)
      , PARTITION p_10000 VALUES LESS THAN (10000)
      , PARTITION p_100000 VALUES LESS THAN (100000)
      , PARTITION p_1000000 VALUES LESS THAN (1000000)
      , PARTITION p_greater_than_1000000 VALUES LESS THAN (maxvalue)
      );
```

> **Note:**
>
> If your enterprise has databases using different character sets, use caution when partitioning on character columns, because the sort sequence of characters is not identical in all character sets. For more information, refer to *Oracle Database Globalization Support Guide*

4.1.2 Creating Range-Interval-Partitioned Tables

The `INTERVAL` clause of the `CREATE TABLE` statement establishes interval partitioning for the table.

You must specify at least one range partition using the `PARTITION` clause. The range partitioning key value determines the high value of the range partitions, which is called the transition point, and the database automatically creates interval partitions for data beyond that transition point. The lower boundary of every interval partition is the non-inclusive upper boundary of the previous range or interval partition.

For example, if you create an interval partitioned table with monthly intervals and the transition point is at January 1, 2010, then the lower boundary for the January 2010 interval is January 1, 2010. The lower boundary for the July 2010 interval is July 1, 2010, regardless of whether the June 2010 partition was previously created. Note, however, that using a date where the high or low bound of the partition would be out of the range set for storage causes an error. For example, `TO_DATE('9999-12-01', 'YYYY-MM-DD')` causes the high bound to be 10000-01-01, which would not be storable if 10000 is out of the legal range.

The optional `STORE IN` clause lets you specify one or more tablespaces into which the database stores interval partition data using a round-robin algorithm for subsequently created interval partitions.

For interval partitioning, you can specify only one partitioning key column and the datatype is restricted.

The following example specifies four partitions with varying interval widths. It also specifies that above the transition point of January 1, 2010, partitions are created with an interval width of one month. The high bound of partition `p3` represents the transition point. `p3` and all partitions below it (`p0`, `p1`, and `p2` in this example) are in the range section while all partitions above it fall into the interval section.

```
CREATE TABLE interval_sales
    ( prod_id        NUMBER(6)
    , cust_id        NUMBER
    , time_id        DATE
    , channel_id     CHAR(1)
    , promo_id       NUMBER(6)
    , quantity_sold  NUMBER(3)
    , amount_sold    NUMBER(10,2)
    )
  PARTITION BY RANGE (time_id)
  INTERVAL(NUMTOYMINTERVAL(1, 'MONTH'))
    ( PARTITION p0 VALUES LESS THAN (TO_DATE('1-1-2008', 'DD-MM-YYYY')),
      PARTITION p1 VALUES LESS THAN (TO_DATE('1-1-2009', 'DD-MM-YYYY')),
      PARTITION p2 VALUES LESS THAN (TO_DATE('1-7-2009', 'DD-MM-YYYY')),
      PARTITION p3 VALUES LESS THAN (TO_DATE('1-1-2010', 'DD-MM-YYYY')) );
```

See Also:

Oracle Database SQL Language Reference for restrictions on partitioning keys, the exact syntax of the partitioning clauses for creating and altering partitioned tables and indexes, any restrictions on their use, and specific privileges required for creating and altering tables.

4.1.3 About Creating Hash Partitioned Tables and Global Indexes

The `PARTITION BY HASH` clause of the `CREATE TABLE` statement identifies that the table is to be hash partitioned.

The `PARTITIONS` clause can then be used to specify the number of partitions to create, and optionally, the tablespaces to store them in. Alternatively, you can use `PARTITION` clauses to name the individual partitions and their tablespaces.

The only attribute you can specify for hash partitions is `TABLESPACE`. All of the hash partitions of a table must share the same segment attributes (except `TABLESPACE`), which are inherited from the table level.

The following topics are discussed:

- Creating a Hash Partitioned Table
- Creating a Hash Partitioned Global Index

4.1.3.1 Creating a Hash Partitioned Table

The example in this topic shows how to create a hash partitioned table.

The partitioning column is `id`, four partitions are created and assigned system generated names, and they are placed in four named tablespaces (`gear1`, `gear2`, `gear3`, `gear4`).

```
CREATE TABLE scubagear
     (id NUMBER,
```

```
    name VARCHAR2 (60))
  PARTITION BY HASH (id)
  PARTITIONS 4
  STORE IN (gear1, gear2, gear3, gear4);
```

In the following example, the number of partitions is specified when creating a hash partitioned table, but system generated names are assigned to them and they are stored in the default tablespace of the table.

```
CREATE TABLE departments_hash (department_id NUMBER(4) NOT NULL,
            department_name VARCHAR2(30))
    PARTITION BY HASH(department_id) PARTITIONS 16;
```

In the following example, names of individual partitions, and tablespaces in which they are to reside, are specified. The initial extent size for each hash partition (segment) is also explicitly stated at the table level, and all partitions inherit this attribute.

```
CREATE TABLE departments_hash (department_id NUMBER(4) NOT NULL,
            department_name VARCHAR2(30))
    STORAGE (INITIAL 10K)
    PARTITION BY HASH(department_id)
      (PARTITION p1 TABLESPACE ts1, PARTITION p2 TABLESPACE ts2,
       PARTITION p3 TABLESPACE ts1, PARTITION p4 TABLESPACE ts3);
```

If you create a local index for this table, the database constructs the index so that it is equipartitioned with the underlying table. The database also ensures that the index is maintained automatically when maintenance operations are performed on the underlying table. The following is an example of creating a local index on a table:

```
CREATE INDEX loc_dept_ix ON departments_hash(department_id) LOCAL;
```

You can optionally name the hash partitions and tablespaces into which the local index partitions are to be stored, but if you do not do so, then the database uses the name of the corresponding base partition as the index partition name, and stores the index partition in the same tablespace as the table partition.

> **See Also:**
>
> Specifying Partitioning on Key Columns for more information about partitioning on key columns

4.1.3.2 Creating a Hash Partitioned Global Index

Hash partitioned global indexes can improve the performance of indexes where a small number of leaf blocks in the index have high contention in multiuser OLTP environments.

Hash partitioned global indexes can also limit the impact of index skew on monotonously increasing column values. Queries involving the equality and `IN` predicates on the index partitioning key can efficiently use hash partitioned global indexes.

The syntax for creating a hash partitioned global index is similar to that used for a hash partitioned table. For example, the statement in Example 4-4 creates a hash partitioned global index:

Example 4-4 Creating a hash partitioned global index

```
CREATE INDEX hgidx ON tab (c1,c2,c3) GLOBAL
     PARTITION BY HASH (c1,c2)
     (PARTITION p1  TABLESPACE tbs_1,
      PARTITION p2  TABLESPACE tbs_2,
      PARTITION p3  TABLESPACE tbs_3,
      PARTITION p4  TABLESPACE tbs_4);
```

4.1.4 About Creating List-Partitioned Tables

The semantics for creating list partitions are very similar to those for creating range partitions.

However, to create list partitions, you specify a `PARTITION BY LIST` clause in the `CREATE TABLE` statement, and the `PARTITION` clauses specify lists of literal values, which are the discrete values of the partitioning columns that qualify rows to be included in the partition. For list partitioning, the partitioning key can be one or multiple column names from the table.

Available only with list partitioning, you can use the keyword `DEFAULT` to describe the value list for a partition. This identifies a partition that accommodates rows that do not map into any of the other partitions.

As with range partitions, optional subclauses of a `PARTITION` clause can specify physical and other attributes specific to a partition segment. If not overridden at the partition level, partitions inherit the attributes of their parent table.

The following topics are discussed:

- Creating a List-Partitioned Table
- Creating a List-Partitioned Table With a Default Partition
- Creating an Automatic List-Partitioned Table
- Creating a Multi-column List-Partitioned Table

4.1.4.1 Creating a List-Partitioned Table

The example in this topic show how to create a list-partitioned table.

Example 4-5creates table `q1_sales_by_region` which is partitioned by regions consisting of groups of US states. A row is mapped to a partition by checking whether the value of the partitioning column for a row matches a value in the value list that describes the partition. For example, the following list describes how some sample rows are inserted into the table.

- (10, 'accounting', 100, 'WA') maps to partition `q1_northwest`
- (20, 'R&D', 150, 'OR') maps to partition `q1_northwest`
- (30, 'sales', 100, 'FL') maps to partition `q1_southeast`
- (40, 'HR', 10, 'TX') maps to partition `q1_southwest`
- (50, 'systems engineering', 10, 'CA') does not map to any partition in the table and raises an error

> **Live SQL:**
>
> View and run a related example on Oracle Live SQL at *Oracle Live SQL: Creating a List Partitioned Table*.

Example 4-5 Creating a list-partitioned table

```
CREATE TABLE q1_sales_by_region
      (deptno number,
       deptname varchar2(20),
       quarterly_sales number(10, 2),
       state varchar2(2))
   PARTITION BY LIST (state)
      (PARTITION q1_northwest VALUES ('OR', 'WA'),
       PARTITION q1_southwest VALUES ('AZ', 'UT', 'NM'),
       PARTITION q1_northeast VALUES  ('NY', 'VM', 'NJ'),
       PARTITION q1_southeast VALUES ('FL', 'GA'),
       PARTITION q1_northcentral VALUES ('SD', 'WI'),
       PARTITION q1_southcentral VALUES ('OK', 'TX'));
```

4.1.4.2 Creating a List-Partitioned Table With a Default Partition

Unlike range partitioning, with list partitioning, there is no apparent sense of order between partitions.

You can also specify a **default partition** into which rows that do not map to any other partition are mapped. If a default partition were specified in the preceding example, the state CA would map to that partition.

Example 4-6 creates table `sales_by_region` and partitions it using the list method. The first two `PARTITION` clauses specify physical attributes, which override the table-level defaults. The remaining `PARTITION` clauses do not specify attributes and those partitions inherit their physical attributes from table-level defaults. A default partition is also specified.

Example 4-6 Creating a list-partitioned table with a default partition

```
CREATE TABLE sales_by_region (item# INTEGER, qty INTEGER,
             store_name VARCHAR(30), state_code VARCHAR(2),
             sale_date DATE)
     STORAGE(INITIAL 10K NEXT 20K) TABLESPACE tbs5
     PARTITION BY LIST (state_code)
     (
     PARTITION region_east
        VALUES ('MA','NY','CT','NH','ME','MD','VA','PA','NJ')
        STORAGE (INITIAL 8M)
        TABLESPACE tbs8,
     PARTITION region_west
        VALUES ('CA','AZ','NM','OR','WA','UT','NV','CO')
        NOLOGGING,
     PARTITION region_south
        VALUES ('TX','KY','TN','LA','MS','AR','AL','GA'),
     PARTITION region_central
        VALUES ('OH','ND','SD','MO','IL','MI','IA'),
     PARTITION region_null
        VALUES (NULL),
     PARTITION region_unknown
        VALUES (DEFAULT)
     );
```

4.1.4.3 Creating an Automatic List-Partitioned Table

The automatic list partitioning method enables list partition creation on demand.

An auto-list partitioned table is similar to a regular list partitioned table, except that this partitioned table is easier to manage. You can create an auto-list partitioned table using only the partitioning key values that are known. As data is loaded into the table, the database automatically creates a new partition if the loaded partitioning key value does not correspond to any of the existing partitions. Because partitions are automatically created on demand, the auto-list partitioning method is conceptually similar to the existing interval partitioning method.

Automatic list partitioning on data types whose value changes very frequently are less suitable for this method unless you can adjust the data. For example, a `SALES_DATE` field with a date value, when the format is not stripped, would increase every second. Each of the `SALES_DATE` values, such as `05-22-2016 08:00:00`, `05-22-2016 08:00:01`, and so on, would generate its own partition. To avoid the creation of a very large number of partitions, you must be aware of the data that would be entered and adjust accordingly. As an example, you can truncate the `SALES_DATE` date value to a day or some other time period, depending on the number of partitions required.

The `CREATE` and `ALTER TABLE` SQL statements are updated with an additional clause to specify `AUTOMATIC` or `MANUAL` list partitioning. An automatic list-partitioned table must have at least one partition when created. Because new partitions are automatically created for new, and unknown, partition key values, an automatic list partition cannot have a `DEFAULT` partition.

You can check the `AUTOLIST` column of the `*_PART_TABLES` view to determine whether a table is automatic list-partitioned.

> **Live SQL:**
>
> View and run a related example on Oracle Live SQL at *Oracle Live SQL: Creating an Automatic List-Partitioned Table*.

Example 4-7 is an example of the `CREATE TABLE` statement using the `AUTOMATIC` keyword for auto-list partitioning on the `sales_state` field. The CREATE TABLE SQL statement creates at least one partition as required. As additional rows are inserted, the number of partitions increases when a new `sales_state` value is added.

Example 4-7 Creating an automatic list partitioned table

```
CREATE TABLE sales_auto_list
(
   salesperson_id   NUMBER(5)    NOT NULL,
   salesperson_name VARCHAR2(30),
   sales_state   VARCHAR2(20) NOT NULL,
   sales_amount  NUMBER(10),
   sales_date    DATE         NOT NULL
)
  PARTITION BY LIST (sales_state) AUTOMATIC
 (PARTITION P_CAL VALUES ('CALIFORNIA')
);

SELECT TABLE_NAME, PARTITIONING_TYPE, AUTOLIST, PARTITION_COUNT FROM USER_PART_TABLES WHERE TABLE_NAME
='SALES_AUTO_LIST';
TABLE_NAME        PARTITIONING_TYPE  AUTOLIST  PARTITION_COUNT
```

```
---------------  -----------------  --------  ---------------
SALES_AUTO_LIST   LIST               YES                     1

SELECT TABLE_NAME, PARTITION_NAME, HIGH_VALUE FROM USER_TAB_PARTITIONS WHERE TABLE_NAME
='SALES_AUTO_LIST';

TABLE_NAME        PARTITION_NAME    HIGH_VALUE
---------------  ---------------   ----------------
SALES_AUTO_LIST     P_CAL           'CALIFORNIA'

INSERT INTO SALES_AUTO_LIST VALUES(021, 'Mary Smith', 'FLORIDA', 41000, TO_DATE ('21-DEC-2018','DD-MON-
YYYY'));
1 row inserted.

INSERT INTO SALES_AUTO_LIST VALUES(032, 'Luis Vargas', 'MICHIGAN', 42000, TO_DATE ('31-DEC-2018','DD-MON-
YYYY'));
1 row inserted.

SELECT TABLE_NAME, PARTITIONING_TYPE, AUTOLIST, PARTITION_COUNT FROM USER_PART_TABLES WHERE TABLE_NAME
='SALES_AUTO_LIST';
TABLE_NAME        PARTITIONING_TYPE  AUTOLIST  PARTITION_COUNT
---------------  -----------------  --------  ---------------
SALES_AUTO_LIST   LIST               YES                     3

INSERT INTO SALES_AUTO_LIST VALUES(015, 'Simone Blair', 'CALIFORNIA', 45000, TO_DATE ('11-JAN-2019','DD-
MON-YYYY'));
1 row inserted.

INSERT INTO SALES_AUTO_LIST VALUES(015, 'Simone Blair', 'OREGON', 38000, TO_DATE ('18-JAN-2019','DD-MON-
YYYY'));
1 row inserted.

SELECT TABLE_NAME, PARTITIONING_TYPE, AUTOLIST,PARTITION_COUNT FROM USER_PART_TABLES WHERE TABLE_NAME
='SALES_AUTO_LIST';
TABLE_NAME        PARTITIONING_TYPE  AUTOLIST  PARTITION_COUNT
---------------  -----------------  --------  ---------------
SALES_AUTO_LIST   LIST               YES                     4


SELECT TABLE_NAME, PARTITION_NAME, HIGH_VALUE FROM USER_TAB_PARTITIONS WHERE TABLE_NAME
='SALES_AUTO_LIST';
TABLE_NAME        PARTITION_NAME    HIGH_VALUE
---------------  ---------------   ----------------
SALES_AUTO_LIST     P_CAL           'CALIFORNIA'
SALES_AUTO_LIST     SYS_P478        'FLORIDA'
SALES_AUTO_LIST     SYS_P479        'MICHIGAN'
SALES_AUTO_LIST     SYS_P480        'OREGON'
```

> **See Also:**
>
> *Oracle Database Reference* for information about `*_PART_TABLES` view

4.1.4.4 Creating a Multi-column List-Partitioned Table

Multi-column list partitioning enables you to partition a table based on list values of multiple columns.

Similar to single-column list partitioning, individual partitions can contain sets containing lists of values.

Multi-column list partitioning is supported on a table using the `PARTITION BY LIST` clause on multiple columns of a table. For example:

```
PARTITION BY LIST (column1,column2)
```

A multi-column list-partitioned table can only have one `DEFAULT` partition.

> **Live SQL:**
>
> View and run a related example on Oracle Live SQL at *Oracle Live SQL: Creating a Multicolumn List-Partitioned Table*.

The following is an example of the `CREATE TABLE` statement using multi-column partitioning on the `state` and `channel` columns.

Example 4-8 Creating a multicolumn list-partitioned table

```
CREATE TABLE sales_by_region_and_channel
 (dept_number      NUMBER NOT NULL,
  dept_name        VARCHAR2(20),
  quarterly_sales  NUMBER(10,2),
  state            VARCHAR2(2),
  channel          VARCHAR2(1)
 )
 PARTITION BY LIST (state, channel)
 (
  PARTITION yearly_west_direct VALUES (('OR','D'),('UT','D'),('WA','D')),
  PARTITION yearly_west_indirect VALUES (('OR','I'),('UT','I'),('WA','I')),
  PARTITION yearly_south_direct VALUES (('AZ','D'),('TX','D'),('GA','D')),
  PARTITION yearly_south_indirect VALUES (('AZ','I'),('TX','I'),('GA','I')),
  PARTITION yearly_east_direct VALUES (('PA','D'), ('NC','D'), ('MA','D')),
  PARTITION yearly_east_indirect VALUES (('PA','I'), ('NC','I'), ('MA','I')),
  PARTITION yearly_north_direct VALUES (('MN','D'),('WI','D'),('MI','D')),
  PARTITION yearly_north_indirect VALUES (('MN','I'),('WI','I'),('MI','I')),
  PARTITION yearly_ny_direct VALUES ('NY','D'),
  PARTITION yearly_ny_indirect VALUES ('NY','I'),
  PARTITION yearly_ca_direct VALUES ('CA','D'),
  PARTITION yearly_ca_indirect VALUES ('CA','I'),
  PARTITION rest VALUES (DEFAULT)
 );

SELECT PARTITION_NAME, HIGH_VALUE FROM USER_TAB_PARTITIONS WHERE TABLE_NAME
='SALES_BY_REGION_AND_CHANNEL';
PARTITION_NAME            HIGH_VALUE
--------------------      ------------------------------------------
REST                      DEFAULT
YEARLY_CA_DIRECT          ( 'CA', 'D' )
YEARLY_CA_INDIRECT        ( 'CA', 'I' )
YEARLY_EAST_DIRECT        ( 'PA', 'D' ), ( 'NC', 'D' ), ( 'MA', 'D' )
YEARLY_EAST_INDIRECT      ( 'PA', 'I' ), ( 'NC', 'I' ), ( 'MA', 'I' )
YEARLY_NORTH_DIRECT       ( 'MN', 'D' ), ( 'WI', 'D' ), ( 'MI', 'D' )
YEARLY_NORTH_INDIRECT     ( 'MN', 'I' ), ( 'WI', 'I' ), ( 'MI', 'I' )
YEARLY_NY_DIRECT          ( 'NY', 'D' )
YEARLY_NY_INDIRECT        ( 'NY', 'I' )
YEARLY_SOUTH_DIRECT       ( 'AZ', 'D' ), ( 'TX', 'D' ), ( 'GA', 'D' )
YEARLY_SOUTH_INDIRECT     ( 'AZ', 'I' ), ( 'TX', 'I' ), ( 'GA', 'I' )
```

```
YEARLY_WEST_DIRECT        ( 'OR', 'D' ), ( 'UT', 'D' ), ( 'WA', 'D' )
YEARLY_WEST_INDIRECT      ( 'OR', 'I' ), ( 'UT', 'I' ), ( 'WA', 'I' )
13 rows selected.

INSERT INTO SALES_BY_REGION_AND_CHANNEL VALUES (005, 'AUTO DIRECT', 701000, 'OR', 'D' );
INSERT INTO SALES_BY_REGION_AND_CHANNEL VALUES (006, 'AUTO INDIRECT', 1201000, 'OR', 'I' );
INSERT INTO SALES_BY_REGION_AND_CHANNEL VALUES (005, 'AUTO DIRECT', 625000, 'WA', 'D' );
INSERT INTO SALES_BY_REGION_AND_CHANNEL VALUES (006, 'AUTO INDIRECT', 945000, 'WA', 'I' );
INSERT INTO SALES_BY_REGION_AND_CHANNEL VALUES (005, 'AUTO DIRECT', 595000, 'UT', 'D' );
INSERT INTO SALES_BY_REGION_AND_CHANNEL VALUES (006, 'AUTO INDIRECT', 825000, 'UT', 'I' );
INSERT INTO SALES_BY_REGION_AND_CHANNEL VALUES (003, 'AUTO DIRECT', 1950000, 'CA', 'D' );
INSERT INTO SALES_BY_REGION_AND_CHANNEL VALUES (004, 'AUTO INDIRECT', 5725000, 'CA', 'I' );
INSERT INTO SALES_BY_REGION_AND_CHANNEL VALUES (010, 'AUTO DIRECT', 925000, 'IL', 'D' );
INSERT INTO SALES_BY_REGION_AND_CHANNEL VALUES (010, 'AUTO INDIRECT', 3250000, 'IL', 'I' );

SELECT DEPT_NUMBER, DEPT_NAME, QUARTERLY_SALES, STATE, CHANNEL FROM SALES_BY_REGION_AND_CHANNEL
PARTITION(yearly_west_direct);
DEPT_NUMBER DEPT_NAME            QUARTERLY_SALES ST C
----------- -------------------- --------------- -- -
          5 AUTO DIRECT                  701000 OR D
          5 AUTO DIRECT                  625000 WA D
          5 AUTO DIRECT                  595000 UT D

SELECT DEPT_NUMBER, DEPT_NAME, QUARTERLY_SALES, STATE, CHANNEL FROM SALES_BY_REGION_AND_CHANNEL
PARTITION(yearly_west_indirect);
DEPT_NUMBER DEPT_NAME            QUARTERLY_SALES ST C
----------- -------------------- --------------- -- -
          6 AUTO INDIRECT               1201000 OR I
          6 AUTO INDIRECT                945000 WA I
          6 AUTO INDIRECT                825000 UT I

SELECT DEPT_NUMBER, DEPT_NAME, QUARTERLY_SALES, STATE, CHANNEL FROM SALES_BY_REGION_AND_CHANNEL
PARTITION(yearly_ca_direct);
DEPT_NUMBER DEPT_NAME            QUARTERLY_SALES ST C
----------- -------------------- --------------- -- -
          3 AUTO DIRECT                 1950000 CA D

SELECT DEPT_NUMBER, DEPT_NAME, QUARTERLY_SALES, STATE, CHANNEL FROM SALES_BY_REGION_AND_CHANNEL
PARTITION(yearly_ca_indirect);
DEPT_NUMBER DEPT_NAME            QUARTERLY_SALES ST C
----------- -------------------- --------------- -- -
          4 AUTO INDIRECT               5725000 CA I

SELECT DEPT_NUMBER, DEPT_NAME, QUARTERLY_SALES, STATE, CHANNEL FROM SALES_BY_REGION_AND_CHANNEL
PARTITION(rest);
DEPT_NUMBER DEPT_NAME            QUARTERLY_SALES ST C
----------- -------------------- --------------- -- -
         10 AUTO DIRECT                  925000 IL D
         10 AUTO INDIRECT               3250000 IL I
```

4.1.5 Creating Reference-Partitioned Tables

To create a reference-partitioned table, you specify a `PARTITION BY REFERENCE` clause in the `CREATE TABLE` statement.

The `PARTITION BY REFERENCE` clause specifies the name of a referential constraint and this constraint becomes the partitioning referential constraint that is used as the basis for reference partitioning in the table. The referential constraint must be enabled and enforced.

As with other partitioned tables, you can specify object-level default attributes, and you can optionally specify partition descriptors that override the object-level defaults on a per-partition basis.

Example 4-9 creates a parent table `orders` which is range-partitioned on `order_date`. The reference-partitioned child table `order_items` is created with four partitions, `Q1_2005`, `Q2_2005`, `Q3_2005`, and `Q4_2005`, where each partition contains the `order_items` rows corresponding to orders in the respective parent partition.

If partition descriptors are provided, then the number of partitions described must exactly equal the number of partitions or subpartitions in the referenced table. If the parent table is a composite partitioned table, then the table has one partition for each subpartition of its parent; otherwise the table has one partition for each partition of its parent.

Partition bounds cannot be specified for the partitions of a reference-partitioned table.

The partitions of a reference-partitioned table can be named. If a partition is not explicitly named, then it inherits its name from the corresponding partition in the parent table, unless this inherited name conflicts with an existing explicit name. In this case, the partition has a system-generated name.

Partitions of a reference-partitioned table collocate with the corresponding partition of the parent table, if no explicit tablespace is specified for the reference-partitioned table's partition.

Example 4-9 Creating reference-partitioned tables

```
CREATE TABLE orders
    ( order_id           NUMBER(12),
      order_date         DATE,
      order_mode         VARCHAR2(8),
      customer_id        NUMBER(6),
      order_status       NUMBER(2),
      order_total        NUMBER(8,2),
      sales_rep_id       NUMBER(6),
      promotion_id       NUMBER(6),
      CONSTRAINT orders_pk PRIMARY KEY(order_id)
    )
  PARTITION BY RANGE(order_date)
    ( PARTITION Q1_2005 VALUES LESS THAN (TO_DATE('01-APR-2005','DD-MON-YYYY')),
      PARTITION Q2_2005 VALUES LESS THAN (TO_DATE('01-JUL-2005','DD-MON-YYYY')),
      PARTITION Q3_2005 VALUES LESS THAN (TO_DATE('01-OCT-2005','DD-MON-YYYY')),
      PARTITION Q4_2005 VALUES LESS THAN (TO_DATE('01-JAN-2006','DD-MON-YYYY'))
    );

CREATE TABLE order_items
    ( order_id           NUMBER(12) NOT NULL,
      line_item_id       NUMBER(3)  NOT NULL,
      product_id         NUMBER(6)  NOT NULL,
      unit_price         NUMBER(8,2),
      quantity           NUMBER(8),
      CONSTRAINT order_items_fk
      FOREIGN KEY(order_id) REFERENCES orders(order_id)
    )
    PARTITION BY REFERENCE(order_items_fk);
```

4.1.6 Creating Interval-Reference Partitioned Tables

You can use interval partitioned tables as parent tables for reference partitioning. Partitions in a reference-partitioned table corresponding to interval partitions in the parent table are created when inserting records into the reference partitioned table.

When creating an interval partition in a child table, the partition name is inherited from the associated parent table fragment. If the child table has a table-level default tablespace, then it is used as tablespace for the new interval partition; otherwise, the tablespace is inherited from the parent table fragment.

The SQL `ALTER TABLE SET INTERVAL` statement is not allowed for reference-partitioned tables, but can be run on tables that have reference-partitioned children. In particular, `ALTER TABLE SET INTERVAL` removes the interval property from the targeted table and converts any interval-reference children to ordinary reference-partitioned tables. Also, the SQL `ALTER TABLE SET STORE IN` statement is not allowed for reference-partitioned tables, but can be run on tables that have reference-partitioned children.

Operations that transform interval partitions to conventional partitions in the parent table, such as `ALTER TABLE SPLIT PARTITION` on an interval partition, construct the corresponding transformation in the child table, creating partitions in the child table as necessary.

For example, the following SQL statements provides three interval partitions in the parent table and none in the child table:

```
CREATE TABLE par(pk INT CONSTRAINT par_pk PRIMARY KEY, i INT)
 PARTITION BY RANGE(i) INTERVAL (10)
 (PARTITION p1 VALUES LESS THAN (10));

CREATE TABLE chi(fk INT NOT NULL, i INT,
 CONSTRAINT chi_fk FOREIGN KEY(fk) REFERENCES par(pk))
 PARTITION BY REFERENCE(chi_fk);

INSERT INTO par VALUES(15, 15);
INSERT INTO par VALUES(25, 25);
INSERT INTO par VALUES(35, 35);
```

You can display information about partitions with the `USER_TAB_PARTITIONS` view:

```
SELECT table_name, partition_position, high_value, interval
   FROM USER_TAB_PARTITIONS WHERE table_name IN ('PAR', 'CHI')
   ORDER BY 1, 2;

TABLE_NAME       PARTITION_POSITION HIGH_VALUE INT
---------------- ------------------ ---------- ---
CHI                               1            NO
PAR                               1         10 NO
PAR                               2         20 YES
PAR                               3         30 YES
PAR                               4         40 YES
```

If the interval partition is split in the parent table, then some interval partitions are converted to conventional partitions for all tables in the hierarchy, creating conventional partitions in the child table in the process. For example:

```
ALTER TABLE par SPLIT PARTITION FOR (25) AT (25)
   INTO (partition x, partition y);

SELECT table_name, partition_position, high_value, interval
```

```
   FROM USER_TAB_PARTITIONS WHERE table_name IN ('PAR', 'CHI')
   ORDER BY 1, 2;

TABLE_NAME       PARTITION_POSITION HIGH_VALUE INT
---------------- ------------------ ---------- ---
CHI                               1            NO
CHI                               2            NO
CHI                               3            NO
CHI                               4            NO
PAR                               1         10 NO
PAR                               2         20 NO
PAR                               3         25 NO
PAR                               4         30 NO
PAR                               5         40 YES
```

Interval-reference functionality requires that the database compatibility level (Oracle Database `COMPATIBLE` initialization parameter setting) be set to greater than or equal to `12.0.0.0`.

4.1.7 Creating a Table Using In-Memory Column Store With Partitioning

You can create a partitioned table using the In-Memory Column Store with the `INMEMORY` clause.

The following example specifies that individual partitions are loaded into the In-Memory Column Store using the `INMEMORY` clause with the partitioning clauses of the `CREATE TABLE` SQL statements.

```
CREATE TABLE list_customers
   ( customer_id            NUMBER(6)
   , cust_first_name        VARCHAR2(20)
   , cust_last_name         VARCHAR2(20)
   , cust_address           CUST_ADDRESS_TYP
   , nls_territory          VARCHAR2(30)
   , cust_email             VARCHAR2(40))
   PARTITION BY LIST (nls_territory) (
   PARTITION asia VALUES ('CHINA', 'THAILAND')
         INMEMORY MEMCOMPRESS FOR CAPACITY HIGH,
   PARTITION europe VALUES ('GERMANY', 'ITALY', 'SWITZERLAND')
         INMEMORY MEMCOMPRESS FOR CAPACITY LOW,
   PARTITION west VALUES ('AMERICA')
         INMEMORY MEMCOMPRESS FOR CAPACITY LOW,
   PARTITION east VALUES ('INDIA')
         INMEMORY MEMCOMPRESS FOR CAPACITY HIGH,
   PARTITION rest VALUES (DEFAULT);
```

See Also:

- *Oracle Database In-Memory Guide* for overview information about In-Memory Column Store
- *Oracle Database In-Memory Guide* for information about enabling objects for population in the In-Memory Column Store and ADO support
- *Oracle Database SQL Language Reference* for information about SQL syntax related to In-Memory Column Store

4.1.8 Creating a Table with Read-Only Partitions or Subpartitions

You can set tables, partitions, and subpartitions to read-only status to protect data from unintentional DML operations by any user or trigger.

Any attempt to update data in a partition or subpartition that is set to read only results in an error, while updating data in partitions or subpartitions that are set to read write succeeds.

The `CREATE TABLE` and `ALTER TABLE` SQL statements provide a read-only clause for partitions and subpartitions. The values of the read-only clause can be `READ ONLY` or `READ WRITE`. `READ WRITE` is the default value. A higher level setting of the read-only clause is applied to partitions and subpartitions unless the read-only clause has been explicitly set for a partition or subpartition.

The following is an example of a creating a composite range-list partitioned table with both read-only and read-write status. The `orders_read_write_only` is explicitly specified as `READ WRITE`, so the default attribute of the table is read write. The default attribute of partition `order_p1` is specified as read only, so the subpartitions `ord_p1_northwest` and `order_p1_southwest` inherit read only status from partition `order_p1`. Subpartitions `ord_p2_southwest` and `order_p3_northwest` are explicitly specified as read only, overriding the default read write status.

Example 4-10 Creating a table with read-only and read-write partitions

```
CREATE TABLE orders_read_write_only (
   order_id NUMBER (12),
   order_date DATE CONSTRAINT order_date_nn NOT NULL,
   state VARCHAR2(2)
   ) READ WRITE
     PARTITION BY RANGE (order_date)
     SUBPARTITION BY LIST (state)
     ( PARTITION order_p1 VALUES LESS THAN (TO_DATE ('01-DEC-2015','DD-MON-YYYY')) READ
ONLY
     ( SUBPARTITION order_p1_northwest VALUES ('OR', 'WA'),
       SUBPARTITION order_p1_southwest VALUES ('AZ', 'UT', 'NM')
      ),
     PARTITION order_p2 VALUES LESS THAN (TO_DATE ('01-MAR-2016','DD-MON-YYYY'))
     ( SUBPARTITION order_p2_northwest VALUES ('OR', 'WA'),
       SUBPARTITION order_p2_southwest VALUES ('AZ', 'UT', 'NM') READ ONLY
      ),
     PARTITION order_p3 VALUES LESS THAN (TO_DATE ('01-JUL-2016','DD-MON-YYYY'))
     (
     SUBPARTITION order_p3_northwest VALUES ('OR', 'WA') READ ONLY,
     SUBPARTITION order_p3_southwest VALUES ('AZ', 'UT', 'NM')
     )
 );
```

You can check the read-only status with the `DEF_READ_ONLY` column of the `*_PART_TABLES` view, the `READ_ONLY` column of the `*_TAB_PARTITIONS` view, and the `READ_ONLY` column of the `*_TAB_SUBPARTITIONS` view. Note that only physical segments, partitions for single-level partitioning and subpartitions for composite partitioning, have a status. All other levels are logical and only have a default status.

```
SQL> SELECT PARTITION_NAME, READ_ONLY FROM USER_TAB_PARTITIONS WHERE TABLE_NAME
='ORDERS_READ_WRITE_ONLY';
PARTITION_NAME                 READ
------------------------------ ----
ORDER_P1                       YES
ORDER_P2                       NONE
```

```
ORDER_P3                        NONE

SQL> SELECT PARTITION_NAME, SUBPARTITION_NAME, READ_ONLY FROM USER_TAB_SUBPARTITIONS
WHERE TABLE_NAME ='ORDERS_READ_WRITE_ONLY';
PARTITION_NAME                 SUBPARTITION_NAME             REA
------------------------------ ------------------------------ ---
ORDER_P1                       ORDER_P1_NORTHWEST            YES
ORDER_P1                       ORDER_P1_SOUTHWEST            YES
ORDER_P2                       ORDER_P2_NORTHWEST            NO
ORDER_P2                       ORDER_P2_SOUTHWEST            YES
ORDER_P3                       ORDER_P3_NORTHWEST            YES
ORDER_P3                       ORDER_P3_SOUTHWEST            NO
```

See Also:

Oracle Database Reference for information about `*_PART_TABLES`, `*_TAB_PARTITIONS`, and `*_TAB_SUBPARTITIONS` views

4.1.9 Creating a Partitioned External Table

You can create partitions for an external table.

The organization external clause identifies the table as external table, followed by the specification and access parameters of the external table. While parameters, such as the default directory; can be overridden on a partition or subpartition level, the external table type and its access parameters are table-level attributes and applicable to all partitions or subpartitions.

The table created in Example 4-11 has three partitions for external data accessed from different locations. Partition `p1` stores customer data for California, located in the default directory of the table. Partition `p2` points to a file storing data for Washington. Partition `p3` does not have a file descriptor and is empty.

Example 4-11 Creating a Partitioned External Table

```
CREATE TABLE sales (loc_id number, prod_id number, cust_id number, amount_sold number,
quantity_sold number)
 ORGANIZATION EXTERNAL
 (TYPE oracle_loader
  DEFAULT DIRECTORY load_d1
  ACCESS PARAMETERS
  ( RECORDS DELIMITED BY NEWLINE CHARACTERSET US7ASCII
    NOBADFILE
    LOGFILE log_dir:'sales.log'
    FIELDS TERMINATED BY ","
   )
 )
  REJECT LIMIT UNLIMITED
 PARTITION BY RANGE (loc_id)
 (PARTITION p1 VALUES LESS THAN (1000) LOCATION ('california.txt'),
  PARTITION p2 VALUES LESS THAN (2000) DEFAULT DIRECTORY load_d2 LOCATION
('washington.txt'),
  PARTITION p3 VALUES LESS THAN (3000))
;
```

> **Note:**
>
> Hive column names should be limited to `[a-zA-Z0-9_]+` on external partitioned tables. If you use any Oracle column names that do not follow this rule, then use the two access parameters `com.oracle.bigdata.fields` and `com.oracle.bigdata.colmap` together to remap the “extended” Oracle column name to a simple Hive identifer.

> **See Also:**
>
> *Oracle Database Administrator’s Guide* for information about partitioning external tables

4.1.10 Specifying Partitioning on Key Columns

For range-partitioned and hash partitioned tables, you can specify up to 16 partitioning key columns.

Use multicolumn partitioning when the partitioning key is composed of several columns and subsequent columns define a higher granularity than the preceding ones. The most common scenario is a decomposed `DATE` or `TIMESTAMP` key, consisting of separated columns, for year, month, and day.

In evaluating multicolumn partitioning keys, the database uses the second value only if the first value cannot uniquely identify a single target partition, and uses the third value only if the first and second do not determine the correct partition, and so forth. A value cannot determine the correct partition only when a partition bound exactly matches that value and the same bound is defined for the next partition. The n^{th} column is investigated only when all previous (n-1) values of the multicolumn key exactly match the (n-1) bounds of a partition. A second column, for example, is evaluated only if the first column exactly matches the partition boundary value. If all column values exactly match all of the bound values for a partition, then the database determines that the row does not fit in this partition and considers the next partition for a match.

For nondeterministic boundary definitions (successive partitions with identical values for at least one column), the partition boundary value becomes an inclusive value, representing a "less than or equal to" boundary. This is in contrast to deterministic boundaries, where the values are always regarded as "less than" boundaries.

The following topics are discussed:

- Creating a Multicolumn Range-Partitioned Table By Date
- Creating a Multicolumn Range-Partitioned Table to Enforce Equal-Sized Partitions

4.1.10.1 Creating a Multicolumn Range-Partitioned Table By Date

The example in this topic shows how to create a multicolumn range-partitioned table by date.

Example 4-12 illustrates the column evaluation for a multicolumn range-partitioned table, storing the actual `DATE` information in three separate columns: `year`, `month`, and `day`. The partitioning granularity is a calendar quarter. The partitioned table being evaluated is created as follows:

The year value for 12-DEC-2000 satisfied the first partition, `before2001`, so no further evaluation is needed:

```
SELECT * FROM sales_demo PARTITION(before2001);

      YEAR      MONTH        DAY AMOUNT_SOLD
---------- ---------- ---------- -----------
      2000         12         12        1000
```

The information for 17-MAR-2001 is stored in partition `q1_2001`. The first partitioning key column, `year`, does not by itself determine the correct partition, so the second partitioning key column, `month`, must be evaluated.

```
SELECT * FROM sales_demo PARTITION(q1_2001);

      YEAR      MONTH        DAY AMOUNT_SOLD
---------- ---------- ---------- -----------
      2001          3         17        2000
```

Following the same determination rule as for the previous record, the second column, `month`, determines partition `q4_2001` as correct partition for 1-NOV-2001:

```
SELECT * FROM sales_demo PARTITION(q4_2001);

      YEAR      MONTH        DAY AMOUNT_SOLD
---------- ---------- ---------- -----------
      2001         11          1        5000
```

The partition for 01-JAN-2002 is determined by evaluating only the `year` column, which indicates the `future` partition:

```
SELECT * FROM sales_demo PARTITION(future);

      YEAR      MONTH        DAY AMOUNT_SOLD
---------- ---------- ---------- -----------
      2002          1          1        4000
```

If the database encounters `MAXVALUE` in a partitioning key column, then all other values of subsequent columns become irrelevant. That is, a definition of partition `future` in the preceding example, having a bound of (`MAXVALUE`,0) is equivalent to a bound of (`MAXVALUE`,100) or a bound of (`MAXVALUE`,`MAXVALUE`).

Example 4-12 Creating a multicolumn range-partitioned table

```
CREATE TABLE sales_demo (
   year          NUMBER,
   month         NUMBER,
   day           NUMBER,
   amount_sold   NUMBER)
PARTITION BY RANGE (year,month)
  (PARTITION before2001 VALUES LESS THAN (2001,1),
   PARTITION q1_2001    VALUES LESS THAN (2001,4),
   PARTITION q2_2001    VALUES LESS THAN (2001,7),
   PARTITION q3_2001    VALUES LESS THAN (2001,10),
   PARTITION q4_2001    VALUES LESS THAN (2002,1),
   PARTITION future     VALUES LESS THAN (MAXVALUE,0));

REM  12-DEC-2000
INSERT INTO sales_demo VALUES(2000,12,12, 1000);
REM  17-MAR-2001
INSERT INTO sales_demo VALUES(2001,3,17, 2000);
```

```
REM  1-NOV-2001
INSERT INTO sales_demo VALUES(2001,11,1, 5000);
REM  1-JAN-2002
INSERT INTO sales_demo VALUES(2002,1,1, 4000);
```

4.1.10.2 Creating a Multicolumn Range-Partitioned Table to Enforce Equal-Sized Partitions

The example in this topic shows how to create a multicolumn range-partitioned table to enforce equal-sized partitions.

The following example illustrates the use of a multicolumn partitioned approach for table `supplier_parts`, storing the information about which suppliers deliver which parts. To distribute the data in equal-sized partitions, it is not sufficient to partition the table based on the `supplier_id`, because some suppliers might provide hundreds of thousands of parts, while others provide only a few specialty parts. Instead, you partition the table on (`supplier_id`, `partnum`) to manually enforce equal-sized partitions.

Every row with `supplier_id` < 10 is stored in partition `p1`, regardless of the `partnum` value. The column `partnum` is evaluated only if `supplier_id` =10, and the corresponding rows are inserted into partition `p1`, `p2`, or even into `p3` when `partnum` >=200. To achieve equal-sized partitions for ranges of `supplier_parts`, you could choose a composite range-hash partitioned table, range partitioned by `supplier_id`, hash subpartitioned by `partnum`.

Defining the partition boundaries for multicolumn partitioned tables must obey some rules. For example, consider a table that is range partitioned on three columns `a`, `b`, and `c`. The individual partitions have range values represented as follows:

```
P0(a0, b0, c0)
P1(a1, b1, c1)
P2(a2, b2, c2)
...
Pn(an, bn, cn)
```

The range values you provide for each partition must follow these rules:

- `a0` must be less than or equal to `a1`, and `a1` must be less than or equal to `a2`, and so on.
- If `a0`=`a1`, then `b0` must be less than or equal to `b1`. If `a0` < `a1`, then `b0` and `b1` can have any values. If `a0`=`a1` and `b0`=`b1`, then `c0` must be less than or equal to `c1`. If `b0`<`b1`, then `c0` and `c1` can have any values, and so on.
- If `a1`=`a2`, then `b1` must be less than or equal to `b2`. If `a1`<`a2`, then `b1` and `b2` can have any values. If `a1`=`a2` and `b1`=`b2`, then `c1` must be less than or equal to `c2`. If `b1`<`b2`, then `c1` and `c2` can have any values, and so on.

```
CREATE TABLE supplier_parts (
   supplier_id      NUMBER,
   partnum          NUMBER,
   price            NUMBER)
PARTITION BY RANGE (supplier_id, partnum)
  (PARTITION p1 VALUES LESS THAN  (10,100),
   PARTITION p2 VALUES LESS THAN (10,200),
   PARTITION p3 VALUES LESS THAN (MAXVALUE,MAXVALUE));
```

The following three records are inserted into the table:

```
INSERT INTO supplier_parts VALUES (5,5, 1000);
INSERT INTO supplier_parts VALUES (5,150, 1000);
INSERT INTO supplier_parts VALUES (10,100, 1000);
```

The first two records are inserted into partition `p1`, uniquely identified by `supplier_id`. However, the third record is inserted into partition `p2`; it matches all range boundary values of partition `p1` exactly and the database therefore considers the following partition for a match. The value of `partnum` satisfies the criteria < 200, so it is inserted into partition `p2`.

```
SELECT * FROM supplier_parts PARTITION (p1);

SUPPLIER_ID    PARTNUM      PRICE
----------- ---------- ----------
          5          5       1000
          5        150       1000

SELECT * FROM supplier_parts PARTITION (p2);

SUPPLIER_ID    PARTNUM      PRICE
----------- ---------- ----------
          10       100       1000
```

4.1.11 Using Virtual Column-Based Partitioning

With partitioning, a virtual column can be used as any regular column.

All partition methods are supported when using virtual columns, including interval partitioning and all different combinations of composite partitioning. A virtual column used as the partitioning column cannot use calls to a PL/SQL function.

The following example shows the `sales` table partitioned by range-range using a virtual column for the subpartitioning key. The virtual column calculates the total value of a sale by multiplying `amount_sold` and `quantity_sold`. As the example shows, row movement is also supported with virtual columns. If row movement is enabled, then a row migrates from one partition to another partition if the virtual column evaluates to a value that belongs to another partition.

```
CREATE TABLE sales
  ( prod_id       NUMBER(6) NOT NULL
  , cust_id       NUMBER NOT NULL
  , time_id       DATE NOT NULL
  , channel_id    CHAR(1) NOT NULL
  , promo_id      NUMBER(6) NOT NULL
  , quantity_sold NUMBER(3) NOT NULL
  , amount_sold   NUMBER(10,2) NOT NULL
  , total_amount AS (quantity_sold * amount_sold)
  )
 PARTITION BY RANGE (time_id) INTERVAL (NUMTOYMINTERVAL(1,'MONTH'))
 SUBPARTITION BY RANGE(total_amount)
 SUBPARTITION TEMPLATE
   ( SUBPARTITION p_small VALUES LESS THAN (1000)
   , SUBPARTITION p_medium VALUES LESS THAN (5000)
   , SUBPARTITION p_large VALUES LESS THAN (10000)
   , SUBPARTITION p_extreme VALUES LESS THAN (MAXVALUE)
   )
 (PARTITION sales_before_2007 VALUES LESS THAN
        (TO_DATE('01-JAN-2007','dd-MON-yyyy'))
)
ENABLE ROW MOVEMENT
PARALLEL NOLOGGING;
```

> **See Also:**
>
> *Oracle Database SQL Language Reference* for the syntax on how to create a virtual column

4.1.12 Using Table Compression with Partitioned Tables

For heap-organized partitioned tables, you can compress some or all partitions using table compression.

The compression attribute can be declared for a tablespace, a table, or a partition of a table. Whenever the compress attribute is not specified, it is inherited like any other storage attribute.

Example 4-13 creates a range-partitioned table with one compressed partition `costs_old`. The compression attribute for the table and all other partitions is inherited from the tablespace level.

Example 4-13 Creating a range-partitioned table with a compressed partition

```
CREATE TABLE costs_demo (
   prod_id     NUMBER(6),    time_id     DATE,
   unit_cost   NUMBER(10,2), unit_price  NUMBER(10,2))
PARTITION BY RANGE (time_id)
   (PARTITION costs_old
      VALUES LESS THAN (TO_DATE('01-JAN-2003', 'DD-MON-YYYY')) COMPRESS,
    PARTITION costs_q1_2003
      VALUES LESS THAN (TO_DATE('01-APR-2003', 'DD-MON-YYYY')),
    PARTITION costs_q2_2003
      VALUES LESS THAN (TO_DATE('01-JUN-2003', 'DD-MON-YYYY')),
    PARTITION costs_recent VALUES LESS THAN (MAXVALUE));
```

4.1.13 Using Key Compression with Partitioned Indexes

You can compress some or all partitions of a B-tree index using key compression.

Key compression is applicable only to B-tree indexes. Bitmap indexes are stored in a compressed manner by default. An index using key compression eliminates repeated occurrences of key column prefix values, thus saving space and I/O.

The following example creates a local partitioned index with all partitions except the most recent one compressed:

```
CREATE INDEX i_cost1 ON costs_demo (prod_id) COMPRESS LOCAL
   (PARTITION costs_old, PARTITION costs_q1_2003,
    PARTITION costs_q2_2003, PARTITION costs_recent NOCOMPRESS);
```

You cannot specify `COMPRESS` (or `NOCOMPRESS`) explicitly for an index subpartition. All index subpartitions of a given partition inherit the key compression setting from the parent partition.

To modify the key compression attribute for all subpartitions of a given partition, you must first issue an `ALTER INDEX...MODIFY PARTITION` statement and then rebuild all subpartitions. The `MODIFY PARTITION` clause marks all index subpartitions as `UNUSABLE`.

4.1.14 Specifying Partitioning with Segments

Partitioning with segments is introduced in this topic.

These topics discuss the functionality when using partitioning with segments.

- Deferred Segment Creation for Partitioning
- Truncating Segments That Are Empty
- Maintenance Procedures for Segment Creation on Demand

4.1.14.1 Deferred Segment Creation for Partitioning

You can defer the creation of segments when creating a partitioned table until the first row is inserted into a partition.

When the first row is inserted, segments are created for the base table partition, LOB columns, all global indexes, and local index partitions. Deferred segment creation can be controlled by the following:

- Setting the `DEFERRED_SEGMENT_CREATION` initialization parameter to `TRUE` or `FALSE` in the initialization parameter file.
- Setting the initialization parameter `DEFERRED_SEGMENT_CREATION` to `TRUE` or `FALSE` with the `ALTER SESSION` or `ALTER SYSTEM` SQL statements.
- Specifying the keywords `SEGMENT CREATION IMMEDIATE` or `SEGMENT CREATION DEFERRED` with the partition clause when issuing the `CREATE TABLE` SQL statement.

You can force the creation of segments for an existing created partition with the `ALTER TABLE MODIFY PARTITION ALLOCATE EXTENT` SQL statement. This statement allocates one extent more than the initial number of extents specified during the `CREATE TABLE`.

Serializable transactions are not supported with deferred segment creation. Inserting data into an empty table with no segment created, or into a partition of an interval partitioned table that does not have a segment yet, can cause an error.

See Also:

- *Oracle Database Reference* for more information about the `DEFERRED_SEGMENT_CREATION` initialization parameter
- *Oracle Database SQL Language Reference* for more information about the `ALTER SESSION` and `ALTER SYSTEM` SQL statements
- *Oracle Database SQL Language Reference* for more information about the keywords `SEGMENT CREATION IMMEDIATE` and `SEGMENT CREATION DEFERRED` of the `CREATE TABLE` SQL statement

4.1.14.2 Truncating Segments That Are Empty

You can drop empty segments in tables and table fragments with the `DBMS_SPACE_ADMIN.DROP_EMPTY_SEGMENTS` procedure.

In addition, if a partition or subpartition has a segment, then the truncate feature drops the segment if the `DROP ALL STORAGE` clause is specified with the `ALTER TABLE TRUNCATE PARTITION` SQL statement.

> **See Also:**
>
> - *Oracle Database PL/SQL Packages and Types Reference* for more information about the `DBMS_SPACE_ADMIN` package
> - *Oracle Database SQL Language Reference* for more information about the `DROP ALL STORAGE` clause of `ALTER TABLE`

4.1.14.3 Maintenance Procedures for Segment Creation on Demand

You can use the `MATERIALIZE_DEFERRED_SEGMENTS` procedure in the `DBMS_SPACE_ADMIN` package to create segments for tables and dependent objects for tables with the deferred segment property.

You can also force the creation of segments for an existing created table and table fragment with the `DBMS_SPACE_ADMIN.MATERIALIZE_DEFERRED_SEGMENTS` procedure. The `MATERIALIZE_DEFERRED_SEGMENTS` procedure differs from the `ALTER TABLE MODIFY PARTITION ALLOCATE EXTENT` SQL statement because it does not allocate one additional extent for the table or table fragment.

> **See Also:**
>
> - *Oracle Database PL/SQL Packages and Types Reference* for more information about the `DBMS_SPACE_ADMIN` package

4.1.15 Specifying Partitioning When Creating Index-Organized Tables

For index-organized tables, you can use the range, list, or hash partitioning method.

The semantics for creating partitioned index-organized tables are similar to that for regular tables with these differences:

- When you create the table, you specify the `ORGANIZATION INDEX` clause, and `INCLUDING` and `OVERFLOW` clauses as necessary.
- The `PARTITION` clause can have `OVERFLOW` subclauses that allow you to specify attributes of the overflow segments at the partition level.

Specifying an `OVERFLOW` clause results in the overflow data segments themselves being equipartitioned with the primary key index segments. Thus, for partitioned index-organized tables with overflow, each partition has an index segment and an overflow data segment.

For index-organized tables, the set of partitioning columns must be a subset of the primary key columns. Because rows of an index-organized table are stored in the primary key index for the table, the partitioning criterion affects the availability. By choosing the partitioning key to be a subset of the primary key, an insert operation must only verify uniqueness of the primary key in a single partition, thereby maintaining partition independence.

Support for secondary indexes on index-organized tables is similar to the support for regular tables. Because of the logical nature of the secondary indexes, global indexes on index-organized tables remain usable for certain operations where they would be marked `UNUSABLE` for regular tables.

The following topics are discussed:

- Creating Range-Partitioned Index-Organized Tables
- Creating Hash Partitioned Index-Organized Tables
- Creating List-Partitioned Index-Organized Tables

> **See Also:**
>
> - Maintenance Operations for Partitioned Tables and Indexes for information about maintenance operations on index-organized tables
> - *Oracle Database Administrator's Guide* for more information about managing index-organized tables
> - *Oracle Database Concepts* for more information about index-organized tables

4.1.15.1 Creating Range-Partitioned Index-Organized Tables

You can partition index-organized tables, and their secondary indexes, by the range method.

In Example 4-14, a range-partitioned index-organized table `sales` is created. The `INCLUDING` clause specifies that all columns after `week_no` are to be stored in an overflow segment. There is one overflow segment for each partition, all stored in the same tablespace (`overflow_here`). Optionally, `OVERFLOW TABLESPACE` could be specified at the individual partition level, in which case some or all of the overflow segments could have separate `TABLESPACE` attributes.

Example 4-14 Creating a range-partitioned index-organized table

```
CREATE TABLE sales(acct_no NUMBER(5),
                   acct_name CHAR(30),
                   amount_of_sale NUMBER(6),
                   week_no INTEGER,
                   sale_details VARCHAR2(1000),
              PRIMARY KEY (acct_no, acct_name, week_no))
     ORGANIZATION INDEX
             INCLUDING week_no
             OVERFLOW TABLESPACE overflow_here
     PARTITION BY RANGE (week_no)
            (PARTITION VALUES LESS THAN (5)
                   TABLESPACE ts1,
             PARTITION VALUES LESS THAN (9)
                   TABLESPACE ts2 OVERFLOW TABLESPACE overflow_ts2,
             ...
             PARTITION VALUES LESS THAN (MAXVALUE)
                   TABLESPACE ts13);
```

4.1.15.2 Creating Hash Partitioned Index-Organized Tables

Another option for partitioning index-organized tables is to use the hash method.

In Example 4-15, the `sales` index-organized table is partitioned by the hash method.

> **Note:**
>
> A well-designed hash function is intended to distribute rows in a well-balanced fashion among the partitions. Therefore, updating the primary key column(s) of a row is very likely to move that row to a different partition. Oracle recommends that you explicitly specify the `ENABLE ROW MOVEMENT` clause when creating a hash partitioned index-organized table with a changeable partitioning key. The default is that `ENABLE ROW MOVEMENT` is disabled.

Example 4-15 Creating a hash partitioned index-organized table

```
CREATE TABLE sales(acct_no NUMBER(5),
                   acct_name CHAR(30),
                   amount_of_sale NUMBER(6),
                   week_no INTEGER,
                   sale_details VARCHAR2(1000),
              PRIMARY KEY (acct_no, acct_name, week_no))
     ORGANIZATION INDEX
             INCLUDING week_no
     OVERFLOW
          PARTITION BY HASH (week_no)
             PARTITIONS 16
             STORE IN (ts1, ts2, ts3, ts4)
             OVERFLOW STORE IN (ts3, ts6, ts9);
```

4.1.15.3 Creating List-Partitioned Index-Organized Tables

The other option for partitioning index-organized tables is to use the list method.

In Example 4-16, the `sales` index-organized table is partitioned by the list method.

Example 4-16 Creating a list-partitioned index-organized table

```
CREATE TABLE sales(acct_no NUMBER(5),
                   acct_name CHAR(30),
                   amount_of_sale NUMBER(6),
                   week_no INTEGER,
                   sale_details VARCHAR2(1000),
              PRIMARY KEY (acct_no, acct_name, week_no))
     ORGANIZATION INDEX
             INCLUDING week_no
             OVERFLOW TABLESPACE ts1
     PARTITION BY LIST (week_no)
            (PARTITION VALUES (1, 2, 3, 4)
                   TABLESPACE ts2,
             PARTITION VALUES (5, 6, 7, 8)
                   TABLESPACE ts3 OVERFLOW TABLESPACE ts4,
             PARTITION VALUES (DEFAULT)
                   TABLESPACE ts5);
```

4.1.16 Partitioning Restrictions for Multiple Block Sizes

Use caution when creating partitioned objects in a database with tablespaces of different block sizes.

The storage of partitioned objects in such tablespaces is subject to some restrictions. Specifically, all partitions of the following entities must reside in tablespaces of the same block size:

- Conventional tables
- Indexes
- Primary key index segments of index-organized tables
- Overflow segments of index-organized tables
- `LOB` columns stored out of line

Therefore:

- For each conventional table, all partitions of that table must be stored in tablespaces with the same block size.
- For each index-organized table, all primary key index partitions must reside in tablespaces of the same block size, and all overflow partitions of that table must reside in tablespaces of the same block size. However, index partitions and overflow partitions can reside in tablespaces of different block size.
- For each index (global or local), each partition of that index must reside in tablespaces of the same block size. However, partitions of different indexes defined on the same object can reside in tablespaces of different block sizes.
- For each `LOB` column, each partition of that column must be stored in tablespaces of equal block sizes. However, different `LOB` columns can be stored in tablespaces of different block sizes.

When you create or alter a partitioned table or index, all tablespaces you *explicitly specify* for the partitions and subpartitions of each entity must be of the same block size. If you *do not explicitly specify* tablespace storage for an entity, then the tablespaces the database uses by default must be of the same block size. Therefore, you must be aware of the default tablespaces at each level of the partitioned object.

4.1.17 Partitioning of Collections in XMLType and Objects

Partitioning when using `XMLType` or object tables and columns follows the basic rules for partitioning.

For the purposes of this discussion, the term *Collection Tables* is used for the following two categories: (1) ordered collection tables inside `XMLType` tables or columns, and (2) nested tables inside object tables or columns.

When you partition Collection Tables, Oracle Database uses the partitioning scheme of the base table. Also, Collection Tables are automatically partitioned when the base table is partitioned. DML against a partitioned nested table behaves in a similar manner to that of a reference partitioned table.

Oracle Database provides a `LOCAL` keyword to equipartition a Collection Table with a partitioned base table. This is the default behavior in this release. The default in earlier releases was not to equipartition the Collection Table with the partitioned base table. Now you must specify the `GLOBAL` keyword to store an unpartitioned Collection Table with a partitioned base table.

Out-of-line (OOL) table partitioning is supported. However, you cannot create two tables of the same XML schema that has out-of-line tables. This restriction means that exchange

partitioning cannot be performed for schemas with OOL tables because it is not possible to have two tables of the same schema.

The statement in the following example creates a nested table partition.

```
CREATE TABLE print_media_part (
   product_id NUMBER(6),
   ad_id NUMBER(6),
   ad_composite BLOB,
   ad_sourcetext CLOB,
   ad_finaltext CLOB,
   ad_fltextn NCLOB,
   ad_textdocs_ntab TEXTDOC_TAB,
   ad_photo BLOB,
   ad_graphic BFILE,
   ad_header ADHEADER_TYP)
NESTED TABLE ad_textdocs_ntab STORE AS textdoc_nt
PARTITION BY RANGE (product_id)
  (PARTITION p1 VALUES LESS THAN (100),
   PARTITION p2 VALUES LESS THAN (200));
```

See Also:

- Performing PMOs on Partitions that Contain Collection Tables and Partitioning of XMLIndex for Binary XML Tables for additional related examples
- Collection Tables for an example of issuing a query against a partitioned nested table and using the `EXPLAIN PLAN` to improve performance
- Changing a Nonpartitioned Table into a Partitioned Table for information about using online redefinition to convert your existing nonpartitioned collection tables to partitioned tables
- *Oracle Database SQL Language Reference* for details about `CREATE TABLE` syntax

4.1.17.1 Performing PMOs on Partitions that Contain Collection Tables

Whether a partition contains Collection Tables or not does not significantly affect your ability to perform partition maintenance operations (PMOs).

Usually, maintenance operations on Collection Tables are carried out on the base table. The following example illustrates a typical `ADD PARTITION` operation based on the preceding nested table partition:

```
ALTER TABLE print_media_part
   ADD PARTITION p4 VALUES LESS THAN (400)
   LOB(ad_photo, ad_composite) STORE AS (TABLESPACE omf_ts1)
   LOB(ad_sourcetext, ad_finaltext) STORE AS (TABLESPACE omf_ts1)
   NESTED TABLE ad_textdocs_ntab STORE AS nt_p3;
```

The storage table for nested table storage column `ad_textdocs_ntab` is named `nt_p3` and inherits all other attributes from the table-level defaults and then from the tablespace defaults.

You must directly invoke the following partition maintenance operations on the storage table corresponding to the collection column:

- modify partition

- move partition
- rename partition
- modify the default attributes of a partition

> **See Also:**
>
> - *Oracle Database SQL Language Reference* for `ADD PARTITION` syntax
> - Maintenance Operations Supported on Partitions for a list of partition maintenance operations that can be performed on partitioned tables and composite partitioned tables

4.1.17.2 Partitioning of XMLIndex for Binary XML Tables

For binary XML tables, XMLIndex is equipartitioned with the base table for range, hash, list, interval, and reference partitions.

In the following example, an XMLIndex is created on a range-partitioned table.

```
CREATE TABLE purchase_order
   (id NUMBER, doc XMLTYPE)
    PARTITION BY RANGE (id)
    (PARTITION p1 VALUES LESS THAN (10),
     PARTITION p2 VALUES LESS THAN (MAXVALUE));

CREATE INDEX purchase_order_idx ON purchase_order(doc)
       INDEXTYPE IS XDB.XMLINDEX LOCAL;
```

> **See Also:**
>
> - *Oracle Database Data Cartridge Developer's Guide* for information about Oracle XML DB and partitioning of XMLIndex for binary XML tables
> - *Oracle XML DB Developer's Guide* for information about XMLIndex
> - *Oracle XML DB Developer's Guide* for information about partitioning XMLType tables and columns

4.2 Specifying Composite Partitioning When Creating Tables

When creating a composite partitioned table, you use the `PARTITION` and `SUBPARTITION` clauses of the `CREATE TABLE` SQL statement.

To create a composite partitioned table, you start by using the `PARTITION BY` {`HASH` | `RANGE` [`INTERVAL`]| `LIST`} clause of a `CREATE TABLE` statement. Next, you specify a `SUBPARTITION BY` clause that follows similar syntax and rules as the `PARTITION BY` clause.

The following topics are discussed:

- Creating Composite Hash-* Partitioned Tables

- Creating Composite Interval-* Partitioned Tables
- Creating Composite List-* Partitioned Tables
- Creating Composite Range-* Partitioned Tables
- Specifying Subpartition Templates to Describe Composite Partitioned Tables

4.2.1 Creating Composite Hash-* Partitioned Tables

Composite hash-* partitioning enables hash partitioning along two dimensions.

The composite hash-hash partitioning strategy has the most business value of the composite hash-* partitioned tables. This technique is beneficial to enable partition-wise joins along two dimensions.

In the following example, the number of subpartitions is specified when creating a composite hash-hash partitioned table; however, names are not specified. System generated names are assigned to partitions and subpartitions, which are stored in the default tablespace of the table.

> **Live SQL:**
>
> View and run a related example on Oracle Live SQL at *Oracle Live SQL: Creating a Composite Hash-Hash Partition Table*.

Example 4-17 Creating a composite hash-hash partitioned table

```
CREATE TABLE departments_courses_hash (
             department_id NUMBER(4) NOT NULL,
             department_name VARCHAR2(30),
             course_id NUMBER(4) NOT NULL)
     PARTITION BY HASH(department_id)
     SUBPARTITION BY HASH (course_id) SUBPARTITIONS 32 PARTITIONS 16;
```

> **See Also:**
>
> Specifying Subpartition Templates to Describe Composite Partitioned Tables to learn how using a subpartition template can simplify the specification of a composite partitioned table

4.2.2 Creating Composite Interval-* Partitioned Tables

The concepts of interval-* composite partitioning are similar to the concepts for range-* partitioning.

However, you extend the `PARTITION BY RANGE` clause to include the `INTERVAL` definition. You must specify at least one range partition using the `PARTITION` clause. The range partitioning key value determines the high value of the range partitions, which is called the transition point, and the database automatically creates interval partitions for data beyond that transition point.

The subpartitions for intervals in an interval-* partitioned table are created when the database creates the interval. You can specify the definition of future subpartitions only with a subpartition template.

The following topics show examples for the different interval-* composite partitioning methods.

- Creating Composite Interval-Hash Partitioned Tables
- Creating Composite Interval-List Partitioned Tables
- Creating Composite Interval-Range Partitioned Tables

> **See Also:**
>
> Specifying Subpartition Templates to Describe Composite Partitioned Tables to learn how using a subpartition template can simplify the specification of a composite partitioned table

4.2.2.1 Creating Composite Interval-Hash Partitioned Tables

You can create an interval-hash partitioned table with multiple hash partitions by specifying multiple hash partitions in the `PARTITION` clause or by using a subpartition template.

If you do not use either of these methods, then future interval partitions get only a single hash subpartition.

The following example shows the `sales` table, interval partitioned using monthly intervals on `time_id`, with hash subpartitions by `cust_id`. This example specifies multiple hash partitions, without any specific tablespace assignment to the individual hash partitions.

> **Live SQL:**
>
> View and run a related example on Oracle Live SQL at *Oracle Live SQL: Creating a Composite Interval-Hash Partitioned Table*.

```
CREATE TABLE sales
  ( prod_id       NUMBER(6)
  , cust_id       NUMBER
  , time_id       DATE
  , channel_id    CHAR(1)
  , promo_id      NUMBER(6)
  , quantity_sold NUMBER(3)
  , amount_sold   NUMBER(10,2)
  )
 PARTITION BY RANGE (time_id) INTERVAL (NUMTOYMINTERVAL(1,'MONTH'))
 SUBPARTITION BY HASH (cust_id) SUBPARTITIONS 4
 (PARTITION before_2000 VALUES LESS THAN (TO_DATE('01-JAN-2000','dd-MON-yyyy'))
  )
PARALLEL;
```

This next example shows the same `sales` table, interval partitioned using monthly intervals on `time_id`, again with hash subpartitions by `cust_id`. This time, however, individual hash partitions are stored in separate tablespaces. The subpartition template is used to define the tablespace assignment for future hash subpartitions.

```
CREATE TABLE sales
  ( prod_id       NUMBER(6)
  , cust_id       NUMBER
```

```
  , time_id       DATE
  , channel_id    CHAR(1)
  , promo_id      NUMBER(6)
  , quantity_sold NUMBER(3)
  , amount_sold   NUMBER(10,2)
  )
 PARTITION BY RANGE (time_id) INTERVAL (NUMTOYMINTERVAL(1,'MONTH'))
 SUBPARTITION BY hash(cust_id)
   SUBPARTITION template
   ( SUBPARTITION p1 TABLESPACE ts1
   , SUBPARTITION p2 TABLESPACE ts2
   , SUBPARTITION p3 TABLESPACE ts3
   , SUBPARTITION P4 TABLESPACE ts4
   )
 (PARTITION before_2000 VALUES LESS THAN (TO_DATE('01-JAN-2000','dd-MON-yyyy'))
  )
PARALLEL;
```

4.2.2.2 Creating Composite Interval-List Partitioned Tables

To define list subpartitions for future interval-list partitions, you must use the subpartition template.

If you do not use the subpartitioning template, then the only subpartition that are created for every interval partition is a `DEFAULT` subpartition.

Example 4-18 shows the `sales_interval_list` table, interval partitioned using monthly intervals on `sales_date`, with list subpartitions by `channel_id`.

Example 4-18 Creating a composite interval-list partitioned table

```
CREATE TABLE sales_interval_list
  ( product_id       NUMBER(6)
  , customer_id      NUMBER
  , channel_id       CHAR(1)
  , promo_id         NUMBER(6)
  , sales_date       DATE
  , quantity_sold    INTEGER
  , amount_sold      NUMBER(10,2)
  )
 PARTITION BY RANGE (sales_date) INTERVAL (NUMTOYMINTERVAL(1,'MONTH'))
 SUBPARTITION BY LIST (channel_id)
   SUBPARTITION TEMPLATE
   ( SUBPARTITION p_catalog VALUES ('C')
   , SUBPARTITION p_internet VALUES ('I')
   , SUBPARTITION p_partners VALUES ('P')
   , SUBPARTITION p_direct_sales VALUES ('S')
   , SUBPARTITION p_tele_sales VALUES ('T')
   )
 (PARTITION before_2017 VALUES LESS THAN (TO_DATE('01-JAN-2017','dd-MON-yyyy'))
  )
PARALLEL;

SELECT TABLE_NAME, PARTITION_NAME, SUBPARTITION_NAME FROM USER_TAB_SUBPARTITIONS WHERE
TABLE_NAME ='SALES_INTERVAL_LIST';
```

4.2.2.3 Creating Composite Interval-Range Partitioned Tables

To define range subpartitions for future interval-range partitions, you must use the subpartition template.

If you do not use the subpartition template, then the only subpartition that is created for every interval partition is a range subpartition with the `MAXVALUE` upper boundary.

Example 4-19 shows the `sales` table, interval partitioned using daily intervals on `time_id`, with range subpartitions by `amount_sold`.

Example 4-19 Creating a composite interval-range partitioned table

```
CREATE TABLE sales
  ( prod_id       NUMBER(6)
  , cust_id       NUMBER
  , time_id       DATE
  , channel_id    CHAR(1)
  , promo_id      NUMBER(6)
  , quantity_sold NUMBER(3)
  , amount_sold   NUMBER(10,2)
  )
 PARTITION BY RANGE (time_id) INTERVAL (NUMTODSINTERVAL(1,'DAY'))
SUBPARTITION BY RANGE(amount_sold)
   SUBPARTITION TEMPLATE
   ( SUBPARTITION p_low VALUES LESS THAN (1000)
   , SUBPARTITION p_medium VALUES LESS THAN (4000)
   , SUBPARTITION p_high VALUES LESS THAN (8000)
   , SUBPARTITION p_ultimate VALUES LESS THAN (maxvalue)
   )
 (PARTITION before_2000 VALUES LESS THAN (TO_DATE('01-JAN-2000','dd-MON-yyyy'))
  )
PARALLEL;
```

4.2.3 Creating Composite List-* Partitioned Tables

The concepts of list-hash, list-list, and list-range composite partitioning are similar to the concepts for range-hash, range-list, and range-range partitioning.

However, for list-* composite partitioning you specify `PARTITION BY LIST` to define the partitioning strategy.

The list partitions of a list-* composite partitioned table are similar to non-composite range partitioned tables. This organization enables optional subclauses of a `PARTITION` clause to specify physical and other attributes, including tablespace, specific to a partition segment. If not overridden at the partition level, then partitions inherit the attributes of their underlying table.

The subpartition descriptions, in the `SUBPARTITION` or `SUBPARTITIONS` clauses, are similar to range-* composite partitioning methods.

The following topics show examples for the different list-* composite partitioning methods.

- Creating Composite List-Hash Partitioned Tables
- Creating Composite List-List Partitioned Tables
- Creating Composite List-Range Partitioned Tables

> **See Also:**
>
> - Specifying Subpartition Templates to Describe Composite Partitioned Tables to learn how using a subpartition template can simplify the specification of a composite partitioned table
> - About Creating Composite Range-Hash Partitioned Tables for more information about the subpartition definition of a list-hash composite partitioning method
> - About Creating Composite Range-List Partitioned Tables for more information about the subpartition definition of a list-list composite partitioning method
> - Creating Composite Range-Range Partitioned Tables for more information about the subpartition definition of a list-range composite partitioning method

4.2.3.1 Creating Composite List-Hash Partitioned Tables

The example in this topic shows how to create a composite list-hash partitioned table.

Example 4-20 shows an `accounts` table that is list partitioned by region and subpartitioned using hash by customer identifier.

Example 4-20 Creating a composite list-hash partitioned table

```
CREATE TABLE accounts
( id             NUMBER
 , account_number NUMBER
 , customer_id    NUMBER
 , balance        NUMBER
 , branch_id      NUMBER
 , region         VARCHAR(2)
 , status         VARCHAR2(1)
)
PARTITION BY LIST (region)
SUBPARTITION BY HASH (customer_id) SUBPARTITIONS 8
( PARTITION p_northwest VALUES ('OR', 'WA')
 , PARTITION p_southwest VALUES ('AZ', 'UT', 'NM')
 , PARTITION p_northeast VALUES ('NY', 'VM', 'NJ')
 , PARTITION p_southeast VALUES ('FL', 'GA')
 , PARTITION p_northcentral VALUES ('SD', 'WI')
 , PARTITION p_southcentral VALUES ('OK', 'TX')
);
```

4.2.3.2 Creating Composite List-List Partitioned Tables

The example in this topic shows how to create a composite list-list partitioned table.

Example 4-21 shows an `accounts` table that is list partitioned by region and subpartitioned using list by account status.

> **Live SQL:**
>
> View and run a related example on Oracle Live SQL at *Oracle Live SQL: Creating a Composite List-List Partitioned Table*.

Example 4-21 Creating a composite list-list partitioned table

```
CREATE TABLE accounts
( id             NUMBER
, account_number NUMBER
, customer_id    NUMBER
, balance        NUMBER
, branch_id      NUMBER
, region         VARCHAR(2)
, status         VARCHAR2(1)
)
PARTITION BY LIST (region)
SUBPARTITION BY LIST (status)
( PARTITION p_northwest VALUES ('OR', 'WA')
  ( SUBPARTITION p_nw_bad VALUES ('B')
  , SUBPARTITION p_nw_average VALUES ('A')
  , SUBPARTITION p_nw_good VALUES ('G')
  )
, PARTITION p_southwest VALUES ('AZ', 'UT', 'NM')
  ( SUBPARTITION p_sw_bad VALUES ('B')
  , SUBPARTITION p_sw_average VALUES ('A')
  , SUBPARTITION p_sw_good VALUES ('G')
  )
, PARTITION p_northeast VALUES ('NY', 'VM', 'NJ')
  ( SUBPARTITION p_ne_bad VALUES ('B')
  , SUBPARTITION p_ne_average VALUES ('A')
  , SUBPARTITION p_ne_good VALUES ('G')
  )
, PARTITION p_southeast VALUES ('FL', 'GA')
  ( SUBPARTITION p_se_bad VALUES ('B')
  , SUBPARTITION p_se_average VALUES ('A')
  , SUBPARTITION p_se_good VALUES ('G')
  )
, PARTITION p_northcentral VALUES ('SD', 'WI')
  ( SUBPARTITION p_nc_bad VALUES ('B')
  , SUBPARTITION p_nc_average VALUES ('A')
  , SUBPARTITION p_nc_good VALUES ('G')
  )
, PARTITION p_southcentral VALUES ('OK', 'TX')
  ( SUBPARTITION p_sc_bad VALUES ('B')
  , SUBPARTITION p_sc_average VALUES ('A')
  , SUBPARTITION p_sc_good VALUES ('G')
  )
);
```

4.2.3.3 Creating Composite List-Range Partitioned Tables

The example in this topic shows how to create a composite list-range partitioned table.

Example 4-22 shows an `accounts` table that is list partitioned by region and subpartitioned using range by account balance, and row movement is enabled. Subpartitions for different list partitions could have different ranges specified.

Example 4-22 Creating a composite list-range partitioned table

```
CREATE TABLE accounts
( id             NUMBER
, account_number NUMBER
, customer_id    NUMBER
, balance        NUMBER
, branch_id      NUMBER
, region         VARCHAR(2)
```

```
, status         VARCHAR2(1)
)
PARTITION BY LIST (region)
SUBPARTITION BY RANGE (balance)
( PARTITION p_northwest VALUES ('OR', 'WA')
  ( SUBPARTITION p_nw_low VALUES LESS THAN (1000)
  , SUBPARTITION p_nw_average VALUES LESS THAN (10000)
  , SUBPARTITION p_nw_high VALUES LESS THAN (100000)
  , SUBPARTITION p_nw_extraordinary VALUES LESS THAN (MAXVALUE)
  )
, PARTITION p_southwest VALUES ('AZ', 'UT', 'NM')
  ( SUBPARTITION p_sw_low VALUES LESS THAN (1000)
  , SUBPARTITION p_sw_average VALUES LESS THAN (10000)
  , SUBPARTITION p_sw_high VALUES LESS THAN (100000)
  , SUBPARTITION p_sw_extraordinary VALUES LESS THAN (MAXVALUE)
  )
, PARTITION p_northeast VALUES ('NY', 'VM', 'NJ')
  ( SUBPARTITION p_ne_low VALUES LESS THAN (1000)
  , SUBPARTITION p_ne_average VALUES LESS THAN (10000)
  , SUBPARTITION p_ne_high VALUES LESS THAN (100000)
  , SUBPARTITION p_ne_extraordinary VALUES LESS THAN (MAXVALUE)
  )
, PARTITION p_southeast VALUES ('FL', 'GA')
  ( SUBPARTITION p_se_low VALUES LESS THAN (1000)
  , SUBPARTITION p_se_average VALUES LESS THAN (10000)
  , SUBPARTITION p_se_high VALUES LESS THAN (100000)
  , SUBPARTITION p_se_extraordinary VALUES LESS THAN (MAXVALUE)
  )
, PARTITION p_northcentral VALUES ('SD', 'WI')
  ( SUBPARTITION p_nc_low VALUES LESS THAN (1000)
  , SUBPARTITION p_nc_average VALUES LESS THAN (10000)
  , SUBPARTITION p_nc_high VALUES LESS THAN (100000)
  , SUBPARTITION p_nc_extraordinary VALUES LESS THAN (MAXVALUE)
  )
, PARTITION p_southcentral VALUES ('OK', 'TX')
  ( SUBPARTITION p_sc_low VALUES LESS THAN (1000)
  , SUBPARTITION p_sc_average VALUES LESS THAN (10000)
  , SUBPARTITION p_sc_high VALUES LESS THAN (100000)
  , SUBPARTITION p_sc_extraordinary VALUES LESS THAN (MAXVALUE)
  )
) ENABLE ROW MOVEMENT;
```

4.2.4 Creating Composite Range-* Partitioned Tables

The methods for creating composite range-* partitioned tables are introduced in this topic.

The following topics show examples of the different range-* composite partitioning methods.

- About Creating Composite Range-Hash Partitioned Tables
- About Creating Composite Range-List Partitioned Tables
- Creating Composite Range-Range Partitioned Tables

> **See Also:**
>
> Specifying Subpartition Templates to Describe Composite Partitioned Tables to learn how using a subpartition template can simplify the specification of a composite partitioned table

4.2.4.1 About Creating Composite Range-Hash Partitioned Tables

The partitions of a range-hash partitioned table are logical structures only, because their data is stored in the segments of their subpartitions.

As with partitions, these subpartitions share the same logical attributes. Unlike range partitions in a range-partitioned table, the subpartitions cannot have different physical attributes from the owning partition, although they are not required to reside in the same tablespace.

The following topics are discussed:

- Creating a Composite Range-Hash Partitioned Table With the Same Tablespaces
- Creating a Composite Range-Hash Partitioned Table With Varying Tablespaces
- Creating a Local Index Across Multiple Tablespaces

> **See Also:**
>
> Specifying Subpartition Templates to Describe Composite Partitioned Tables to learn how using a subpartition template can simplify the specification of a composite partitioned table

4.2.4.1.1 Creating a Composite Range-Hash Partitioned Table With the Same Tablespaces

The example in this topic shows how to create a composite range-hash partitioned table using the same tablespaces.

The statement in Example 4-23 creates a range-hash partitioned table. Four range partitions are created, each containing eight subpartitions. Because the subpartitions are not named, system generated names are assigned, but the `STORE IN` clause distributes them across the 4 specified tablespaces (`ts1`, `ts2`, `ts3`,`ts4`).

Example 4-23 Creating a composite range-hash partitioned table using one STORE IN clause

```
CREATE TABLE sales
  ( prod_id       NUMBER(6)
  , cust_id       NUMBER
  , time_id       DATE
  , channel_id    CHAR(1)
  , promo_id      NUMBER(6)
  , quantity_sold NUMBER(3)
  , amount_sold   NUMBER(10,2)
  )
 PARTITION BY RANGE (time_id) SUBPARTITION BY HASH (cust_id)
  SUBPARTITIONS 8 STORE IN (ts1, ts2, ts3, ts4)
 ( PARTITION sales_q1_2006 VALUES LESS THAN (TO_DATE('01-APR-2006','dd-MON-yyyy'))
  , PARTITION sales_q2_2006 VALUES LESS THAN (TO_DATE('01-JUL-2006','dd-MON-yyyy'))
  , PARTITION sales_q3_2006 VALUES LESS THAN (TO_DATE('01-OCT-2006','dd-MON-yyyy'))
  , PARTITION sales_q4_2006 VALUES LESS THAN (TO_DATE('01-JAN-2007','dd-MON-yyyy'))
 );
```

4.2.4.1.2 Creating a Composite Range-Hash Partitioned Table With Varying Tablespaces

The example in this topic shows how to create a composite range-hash partitioned table using varying tablespaces.

Attributes specified for a range partition apply to all subpartitions of that partition. You can specify different attributes for each range partition, and you can specify a `STORE IN` clause at the partition level if the list of tablespaces across which the subpartitions of that partition should be spread is different from those of other partitions. This is illustrated in the following example.

```
CREATE TABLE employees_range_hash
         (department_id NUMBER(4) NOT NULL,
          last_name VARCHAR2(25),
          job_id VARCHAR2(10))
     PARTITION BY RANGE(department_id) SUBPARTITION BY HASH(last_name)
        SUBPARTITIONS 8 STORE IN (ts1, ts3, ts5, ts7)
    (PARTITION p1 VALUES LESS THAN (1000),
     PARTITION p2 VALUES LESS THAN (2000)
        STORE IN (ts2, ts4, ts6, ts8),
     PARTITION p3 VALUES LESS THAN (MAXVALUE)
       (SUBPARTITION p3_s1 TABLESPACE ts4,
        SUBPARTITION p3_s2 TABLESPACE ts5));
```

4.2.4.1.3 Creating a Local Index Across Multiple Tablespaces

The example in this topic shows how to create a local index across multiple tablespaces.

The following statement is an example of creating a local index on a table where the index segments are spread across tablespaces `ts7`, `ts8`, and `ts9`.

```
CREATE INDEX employee_ix ON employees_range_hash(department_id)
     LOCAL STORE IN (ts7, ts8, ts9);
```

This local index is equipartitioned with the base table so that it consists of as many partitions as the base table. Each index partition consists of as many subpartitions as the corresponding base table partition. Index entries for rows in a given subpartition of the base table are stored in the corresponding subpartition of the index.

4.2.4.2 About Creating Composite Range-List Partitioned Tables

The range partitions of a range-list composite partitioned table are described as the same for non-composite range partitioned tables.

This organization enables optional subclauses of a `PARTITION` clause to specify physical and other attributes, including tablespace, specific to a partition segment. If not overridden at the partition level, partitions inherit the attributes of their underlying table.

The list subpartition descriptions, in the `SUBPARTITION` clauses, are described as for non-composite list partitions, except the only physical attribute that can be specified is a tablespace (optional). Subpartitions inherit all other physical attributes from the partition description.

The following topics are discussed:

- Creating a Composite Range-List Partitioned Table
- Creating a Composite Range-List Partitioned Table Specifying Tablespaces

> **See Also:**
>
> Specifying Subpartition Templates to Describe Composite Partitioned Tables to learn how using a subpartition template can simplify the specification of a composite partitioned table

4.2.4.2.1 Creating a Composite Range-List Partitioned Table

The example in this topic shows how to create a composite range-list partitioned table.

Example 4-24 illustrates how range-list partitioning might be used. The example tracks sales data of products by quarters and within each quarter, groups it by specified states.

A row is mapped to a partition by checking whether the value of the partitioning column for a row falls within a specific partition range. The row is then mapped to a subpartition within that partition by identifying the subpartition whose descriptor value list contains a value matching the subpartition column value. For example, the following list describes how some sample rows are inserted.

- (10, 4532130, '23-Jan-1999', 8934.10, 'WA') maps to subpartition `q1_1999_northwest`
- (20, 5671621, '15-May-1999', 49021.21, 'OR') maps to subpartition `q2_1999_northwest`
- (30, 9977612, '07-Sep-1999', 30987.90, 'FL') maps to subpartition `q3_1999_southeast`
- (40, 9977612, '29-Nov-1999', 67891.45, 'TX') maps to subpartition `q4_1999_southcentral`
- (40, 4532130, '5-Jan-2000', 897231.55, 'TX') does not map to any partition in the table and displays an error
- (50, 5671621, '17-Dec-1999', 76123.35, 'CA') does not map to any subpartition in the table and displays an error

> **Live SQL:**
>
> View and run a related example on Oracle Live SQL at *Oracle Live SQL: Creating a Composite Range-List Partitioned Table*.

Example 4-24 Creating a composite range-list partitioned table

```
CREATE TABLE quarterly_regional_sales
      (deptno number, item_no varchar2(20),
       txn_date date, txn_amount number, state varchar2(2))
  TABLESPACE ts4
  PARTITION BY RANGE (txn_date)
    SUBPARTITION BY LIST (state)
      (PARTITION q1_1999 VALUES LESS THAN (TO_DATE('1-APR-1999','DD-MON-YYYY'))
         (SUBPARTITION q1_1999_northwest VALUES ('OR', 'WA'),
          SUBPARTITION q1_1999_southwest VALUES ('AZ', 'UT', 'NM'),
          SUBPARTITION q1_1999_northeast VALUES ('NY', 'VM', 'NJ'),
          SUBPARTITION q1_1999_southeast VALUES ('FL', 'GA'),
          SUBPARTITION q1_1999_northcentral VALUES ('SD', 'WI'),
          SUBPARTITION q1_1999_southcentral VALUES ('OK', 'TX')
         ),
       PARTITION q2_1999 VALUES LESS THAN ( TO_DATE('1-JUL-1999','DD-MON-YYYY'))
         (SUBPARTITION q2_1999_northwest VALUES ('OR', 'WA'),
```

```
     SUBPARTITION q2_1999_southwest VALUES ('AZ', 'UT', 'NM'),
     SUBPARTITION q2_1999_northeast VALUES ('NY', 'VM', 'NJ'),
     SUBPARTITION q2_1999_southeast VALUES ('FL', 'GA'),
     SUBPARTITION q2_1999_northcentral VALUES ('SD', 'WI'),
     SUBPARTITION q2_1999_southcentral VALUES ('OK', 'TX')
    ),
  PARTITION q3_1999 VALUES LESS THAN (TO_DATE('1-OCT-1999','DD-MON-YYYY'))
    (SUBPARTITION q3_1999_northwest VALUES ('OR', 'WA'),
     SUBPARTITION q3_1999_southwest VALUES ('AZ', 'UT', 'NM'),
     SUBPARTITION q3_1999_northeast VALUES ('NY', 'VM', 'NJ'),
     SUBPARTITION q3_1999_southeast VALUES ('FL', 'GA'),
     SUBPARTITION q3_1999_northcentral VALUES ('SD', 'WI'),
     SUBPARTITION q3_1999_southcentral VALUES ('OK', 'TX')
    ),
  PARTITION q4_1999 VALUES LESS THAN ( TO_DATE('1-JAN-2000','DD-MON-YYYY'))
    (SUBPARTITION q4_1999_northwest VALUES ('OR', 'WA'),
     SUBPARTITION q4_1999_southwest VALUES ('AZ', 'UT', 'NM'),
     SUBPARTITION q4_1999_northeast VALUES ('NY', 'VM', 'NJ'),
     SUBPARTITION q4_1999_southeast VALUES ('FL', 'GA'),
     SUBPARTITION q4_1999_northcentral VALUES ('SD', 'WI'),
     SUBPARTITION q4_1999_southcentral VALUES ('OK', 'TX')
    )
);
```

4.2.4.2.2 Creating a Composite Range-List Partitioned Table Specifying Tablespaces

The example in this topic shows how to create a composite range-list partitioned table while specifying tablespaces.

The partitions of a range-list partitioned table are logical structures only, because their data is stored in the segments of their subpartitions. The list subpartitions have the same characteristics as list partitions. You can specify a default subpartition, just as you specify a default partition for list partitioning.

The following example creates a table that specifies a tablespace at the partition and subpartition levels. The number of subpartitions within each partition varies, and default subpartitions are specified. This example results in the following subpartition descriptions:

- All subpartitions inherit their physical attributes, other than tablespace, from tablespace level defaults. This is because the only physical attribute that has been specified for partitions or subpartitions is tablespace. There are no table level physical attributes specified, thus tablespace level defaults are inherited at all levels.
- The first 4 subpartitions of partition `q1_1999` are all contained in `tbs_1`, except for the subpartition `q1_others`, which is stored in `tbs_4` and contains all rows that do not map to any of the other partitions.
- The 6 subpartitions of partition `q2_1999` are all stored in `tbs_2`.
- The first 2 subpartitions of partition `q3_1999` are all contained in `tbs_3`, except for the subpartition `q3_others`, which is stored in `tbs_4` and contains all rows that do not map to any of the other partitions.
- There is no subpartition description for partition `q4_1999`. This results in one default subpartition being created and stored in `tbs_4`. The subpartition name is system generated in the form `SYS_SUBP`*n*.

```
CREATE TABLE sample_regional_sales
      (deptno number, item_no varchar2(20),
       txn_date date, txn_amount number, state varchar2(2))
  PARTITION BY RANGE (txn_date)
    SUBPARTITION BY LIST (state)
```

```
(PARTITION q1_1999 VALUES LESS THAN (TO_DATE('1-APR-1999','DD-MON-YYYY'))
    TABLESPACE tbs_1
   (SUBPARTITION q1_1999_northwest VALUES ('OR', 'WA'),
    SUBPARTITION q1_1999_southwest VALUES ('AZ', 'UT', 'NM'),
    SUBPARTITION q1_1999_northeast VALUES ('NY', 'VM', 'NJ'),
    SUBPARTITION q1_1999_southeast VALUES ('FL', 'GA'),
    SUBPARTITION q1_others VALUES (DEFAULT) TABLESPACE tbs_4
   ),
 PARTITION q2_1999 VALUES LESS THAN ( TO_DATE('1-JUL-1999','DD-MON-YYYY'))
    TABLESPACE tbs_2
   (SUBPARTITION q2_1999_northwest VALUES ('OR', 'WA'),
    SUBPARTITION q2_1999_southwest VALUES ('AZ', 'UT', 'NM'),
    SUBPARTITION q2_1999_northeast VALUES ('NY', 'VM', 'NJ'),
    SUBPARTITION q2_1999_southeast VALUES ('FL', 'GA'),
    SUBPARTITION q2_1999_northcentral VALUES ('SD', 'WI'),
    SUBPARTITION q2_1999_southcentral VALUES ('OK', 'TX')
   ),
 PARTITION q3_1999 VALUES LESS THAN (TO_DATE('1-OCT-1999','DD-MON-YYYY'))
    TABLESPACE tbs_3
   (SUBPARTITION q3_1999_northwest VALUES ('OR', 'WA'),
    SUBPARTITION q3_1999_southwest VALUES ('AZ', 'UT', 'NM'),
    SUBPARTITION q3_others VALUES (DEFAULT) TABLESPACE tbs_4
   ),
 PARTITION q4_1999 VALUES LESS THAN ( TO_DATE('1-JAN-2000','DD-MON-YYYY'))
    TABLESPACE tbs_4
);
```

4.2.4.3 Creating Composite Range-Range Partitioned Tables

The range partitions of a range-range composite partitioned table are similar to non-composite range partitioned tables.

This organization enables optional subclauses of a `PARTITION` clause to specify physical and other attributes, including tablespace, specific to a partition segment. If not overridden at the partition level, then partitions inherit the attributes of their underlying table.

The range subpartition descriptions, in the `SUBPARTITION` clauses, are similar to non-composite range partitions, except the only physical attribute that can be specified is an optional tablespace. Subpartitions inherit all other physical attributes from the partition description.

The following example illustrates how range-range partitioning might be used. The example tracks shipments. The service level agreement with the customer states that every order is delivered in the calendar month after the order was placed. The following types of orders are identified:

A row is mapped to a partition by checking whether the value of the partitioning column for a row falls within a specific partition range. The row is then mapped to a subpartition within that partition by identifying whether the value of the subpartitioning column falls within a specific range. For example, a shipment with an order date in September 2006 and a delivery date of October 28, 2006 falls in partition `p06_oct_a`.

- E (EARLY): orders that are delivered before the middle of the next month after the order was placed. These orders likely exceed customers' expectations.
- A (AGREED): orders that are delivered in the calendar month after the order was placed (but not early orders).
- L (LATE): orders that were only delivered starting the second calendar month after the order was placed.

```
CREATE TABLE shipments
( order_id      NUMBER NOT NULL
, order_date    DATE NOT NULL
, delivery_date DATE NOT NULL
, customer_id   NUMBER NOT NULL
, sales_amount  NUMBER NOT NULL
)
PARTITION BY RANGE (order_date)
SUBPARTITION BY RANGE (delivery_date)
( PARTITION p_2006_jul VALUES LESS THAN (TO_DATE('01-AUG-2006','dd-MON-yyyy'))
  ( SUBPARTITION p06_jul_e VALUES LESS THAN (TO_DATE('15-AUG-2006','dd-MON-yyyy'))
  , SUBPARTITION p06_jul_a VALUES LESS THAN (TO_DATE('01-SEP-2006','dd-MON-yyyy'))
  , SUBPARTITION p06_jul_l VALUES LESS THAN (MAXVALUE)
  )
, PARTITION p_2006_aug VALUES LESS THAN (TO_DATE('01-SEP-2006','dd-MON-yyyy'))
  ( SUBPARTITION p06_aug_e VALUES LESS THAN (TO_DATE('15-SEP-2006','dd-MON-yyyy'))
  , SUBPARTITION p06_aug_a VALUES LESS THAN (TO_DATE('01-OCT-2006','dd-MON-yyyy'))
  , SUBPARTITION p06_aug_l VALUES LESS THAN (MAXVALUE)
  )
, PARTITION p_2006_sep VALUES LESS THAN (TO_DATE('01-OCT-2006','dd-MON-yyyy'))
  ( SUBPARTITION p06_sep_e VALUES LESS THAN (TO_DATE('15-OCT-2006','dd-MON-yyyy'))
  , SUBPARTITION p06_sep_a VALUES LESS THAN (TO_DATE('01-NOV-2006','dd-MON-yyyy'))
  , SUBPARTITION p06_sep_l VALUES LESS THAN (MAXVALUE)
  )
, PARTITION p_2006_oct VALUES LESS THAN (TO_DATE('01-NOV-2006','dd-MON-yyyy'))
  ( SUBPARTITION p06_oct_e VALUES LESS THAN (TO_DATE('15-NOV-2006','dd-MON-yyyy'))
  , SUBPARTITION p06_oct_a VALUES LESS THAN (TO_DATE('01-DEC-2006','dd-MON-yyyy'))
  , SUBPARTITION p06_oct_l VALUES LESS THAN (MAXVALUE)
  )
, PARTITION p_2006_nov VALUES LESS THAN (TO_DATE('01-DEC-2006','dd-MON-yyyy'))
  ( SUBPARTITION p06_nov_e VALUES LESS THAN (TO_DATE('15-DEC-2006','dd-MON-yyyy'))
  , SUBPARTITION p06_nov_a VALUES LESS THAN (TO_DATE('01-JAN-2007','dd-MON-yyyy'))
  , SUBPARTITION p06_nov_l VALUES LESS THAN (MAXVALUE)
  )
, PARTITION p_2006_dec VALUES LESS THAN (TO_DATE('01-JAN-2007','dd-MON-yyyy'))
  ( SUBPARTITION p06_dec_e VALUES LESS THAN (TO_DATE('15-JAN-2007','dd-MON-yyyy'))
  , SUBPARTITION p06_dec_a VALUES LESS THAN (TO_DATE('01-FEB-2007','dd-MON-yyyy'))
  , SUBPARTITION p06_dec_l VALUES LESS THAN (MAXVALUE)
  )
);
```

> **See Also:**
>
> Specifying Subpartition Templates to Describe Composite Partitioned Tables to learn how using a subpartition template can simplify the specification of a composite partitioned table

4.2.5 Specifying Subpartition Templates to Describe Composite Partitioned Tables

You can create subpartitions in a composite partitioned table using a subpartition template.

A subpartition template simplifies the specification of subpartitions by not requiring that a subpartition descriptor be specified for every partition in the table. Instead, you describe subpartitions only one time in a template, then apply that subpartition template to every

partition in the table. For interval-* composite partitioned tables, the subpartition template is the only way to define subpartitions for interval partitions.

The subpartition template is used whenever a subpartition descriptor is not specified for a partition. If a subpartition descriptor is specified, then it is used instead of the subpartition template for that partition. If no subpartition template is specified, and no subpartition descriptor is supplied for a partition, then a single default subpartition is created.

The following topics are discussed:

- Specifying a Subpartition Template for a *-Hash Partitioned Table
- Specifying a Subpartition Template for a *-List Partitioned Table

4.2.5.1 Specifying a Subpartition Template for a *-Hash Partitioned Table

For range-hash, interval-hash, and list-hash partitioned tables, the subpartition template can describe the subpartitions in detail, or it can specify just the number of hash subpartitions.

Example 4-25 creates a range-hash partitioned table using a subpartition template and displays the subpartition names and tablespaces.

The example produces a table with the following description.

- Every partition has four subpartitions as described in the subpartition template.
- Each subpartition has a tablespace specified. It is required that if a tablespace is specified for one subpartition in a subpartition template, then one must be specified for all.
- The names of the subpartitions, unless you use interval-* subpartitioning, are generated by concatenating the partition name with the subpartition name in the form:

  partition name_subpartition name

  For interval-* subpartitioning, the subpartition names are system-generated in the form:

  `SYS_SUBP`*n*

Example 4-25 Creating a range-hash partitioned table with a subpartition template

```
CREATE TABLE employees_sub_template (department_id NUMBER(4) NOT NULL,
             last_name VARCHAR2(25), job_id VARCHAR2(10))
     PARTITION BY RANGE(department_id) SUBPARTITION BY HASH(last_name)
     SUBPARTITION TEMPLATE
         (SUBPARTITION a TABLESPACE ts1,
          SUBPARTITION b TABLESPACE ts2,
          SUBPARTITION c TABLESPACE ts3,
          SUBPARTITION d TABLESPACE ts4
         )
    (PARTITION p1 VALUES LESS THAN (1000),
     PARTITION p2 VALUES LESS THAN (2000),
     PARTITION p3 VALUES LESS THAN (MAXVALUE)
    );

SQL> SELECT TABLESPACE_NAME, PARTITION_NAME, SUBPARTITION_NAME
  2  FROM DBA_TAB_SUBPARTITIONS WHERE TABLE_NAME='EMPLOYEEES_SUB_TEMPLATE'
  3  ORDER BY TABLESPACE_NAME;

TABLESPACE_NAME PARTITION_NAME  SUBPARTITION_NAME
--------------- --------------- ------------------
TS1             P1              P1_A
TS1             P2              P2_A
TS1             P3              P3_A
TS2             P1              P1_B
```

```
TS2             P2              P2_B
TS2             P3              P3_B
TS3             P1              P1_C
TS3             P2              P2_C
TS3             P3              P3_C
TS4             P1              P1_D
TS4             P2              P2_D
TS4             P3              P3_D

12 rows selected.
```

4.2.5.2 Specifying a Subpartition Template for a *-List Partitioned Table

For -list partitioned tables, the subpartition template can describe the subpartitions in detail.

Example 4-26, for a range-list partitioned table, illustrates how using a subpartition template can help you stripe data across tablespaces. In this example, a table is created where the table subpartitions are vertically striped, meaning that subpartition *n* from every partition is in the same tablespace.

If you specified the tablespaces at the partition level (for example, `tbs_1` for partition `q1_1999`, `tbs_2` for partition `q2_1999`, `tbs_3` for partition `q3_1999`, and `tbs_4` for partition `q4_1999`) and not in the subpartition template, then the table would be horizontally striped. All subpartitions would be in the tablespace of the owning partition.

Example 4-26 Creating a range-list partitioned table with a subpartition template

```
CREATE TABLE stripe_regional_sales
            ( deptno number, item_no varchar2(20),
              txn_date date, txn_amount number, state varchar2(2))
   PARTITION BY RANGE (txn_date)
   SUBPARTITION BY LIST (state)
   SUBPARTITION TEMPLATE
      (SUBPARTITION northwest VALUES ('OR', 'WA') TABLESPACE tbs_1,
       SUBPARTITION southwest VALUES ('AZ', 'UT', 'NM') TABLESPACE tbs_2,
       SUBPARTITION northeast VALUES ('NY', 'VM', 'NJ') TABLESPACE tbs_3,
       SUBPARTITION southeast VALUES ('FL', 'GA') TABLESPACE tbs_4,
       SUBPARTITION midwest VALUES ('SD', 'WI') TABLESPACE tbs_5,
       SUBPARTITION south VALUES ('AL', 'AK') TABLESPACE tbs_6,
       SUBPARTITION others VALUES (DEFAULT ) TABLESPACE tbs_7
      )
  (PARTITION q1_1999 VALUES LESS THAN ( TO_DATE('01-APR-1999','DD-MON-YYYY')),
   PARTITION q2_1999 VALUES LESS THAN ( TO_DATE('01-JUL-1999','DD-MON-YYYY')),
   PARTITION q3_1999 VALUES LESS THAN ( TO_DATE('01-OCT-1999','DD-MON-YYYY')),
   PARTITION q4_1999 VALUES LESS THAN ( TO_DATE('1-JAN-2000','DD-MON-YYYY'))
  );
```

4.3 Maintenance Operations Supported on Partitions

There are various maintenance operations that can be performed on partitions, subpartitions, and index partitions.

The maintenance operations that are supported on partitions, subpartitions, and index partitions are described in the following tables and topics.

- Table 4-1 lists partition maintenance operations that can be performed on partitioned tables and composite partitioned tables
- Table 4-2 lists subpartition maintenance operations that can be performed on composite partitioned tables

- Table 4-3 lists maintenance operations that can be performed on index partitions, and on which type of index (global or local) they can be performed
- Updating Indexes Automatically
- Asynchronous Global Index Maintenance for Dropping and Truncating Partitions
- Modifying a Subpartition Template
- Filtering Maintenance Operations

For each type of partitioning and subpartitioning in Table 4-1 and Table 4-2, the specific clause of the `ALTER TABLE` statement that is used to perform that maintenance operation is listed.

> **Note:**
>
> Partition maintenance operations on multiple partitions are not supported on tables with domain indexes.

Table 4-1 ALTER TABLE Maintenance Operations for Table Partitions

Maintenance Operation	Range Composite Range-*	Interval Composite Interval-*	Hash	List Composite List-*	Reference
Adding Partitions, refer to About Adding Partitions and Subpartitions	`ADD PARTITION`, single and multiple partitions	N/A	`ADD PARTITION`	`ADD PARTITION`, single and multiple partitions	N/A. (These operations cannot be performed on reference-partitioned tables. If performed on a parent table, then these operations cascade to all descendant tables.)
Coalescing Partitions, refer to About Coalescing Partitions and Subpartitions	N/A	N/A	`COALESCE PARTITION`	N/A	N/A (These operations cannot be performed on reference-partitioned tables. If performed on a parent table, then these operations cascade to all descendant tables.)
Dropping Partitions, refer to About Dropping Partitions and Subpartitions	`DROP PARTITION`, single and multiple partitions	`DROP PARTITION`, single and multiple partitions	N/A	`DROP PARTITION`, single and multiple partitions	N/A (These operations cannot be performed on reference-partitioned tables. If performed on a parent table, then these operations cascade to all descendant tables.)

Table 4-1 (Cont.) ALTER TABLE Maintenance Operations for Table Partitions

Maintenance Operation	Range Composite Range-*	Interval Composite Interval-*	Hash	List Composite List-*	Reference
Exchanging Partitions, refer to About Exchanging Partitions and Subpartitions	`EXCHANGE PARTITION`	`EXCHANGE PARTITION`	`EXCHANGE PARTITION`	`EXCHANGE PARTITION`	`EXCHANGE PARTITION`
Merging Partitions, refer to About Merging Partitions and Subpartitions	`MERGE PARTITIONS`, single and multiple partitions	`MERGE PARTITIONS`, single and multiple partitions	N/A	`MERGE PARTITIONS`, single and multiple partitions	N/A (These operations cannot be performed on reference-partitioned tables. If performed on a parent table, then these operations cascade to all descendant tables.)
About Modifying Default Attributes	`MODIFY DEFAULT ATTRIBUTES`	`MODIFY DEFAULT ATTRIBUTES`	`MODIFY DEFAULT ATTRIBUTES`	`MODIFY DEFAULT ATTRIBUTES`	`MODIFY DEFAULT ATTRIBUTES`
About Modifying Real Attributes of Partitions	`MODIFY PARTITION`	`MODIFY PARTITION`	`MODIFY PARTITION`	`MODIFY PARTITION`	`MODIFY PARTITION`
About Modifying List Partitions: Adding Values	N/A	N/A	N/A	`MODIFY PARTITION ADD VALUES`	N/A
About Modifying List Partitions: Dropping Values	N/A	N/A	N/A	`MODIFY PARTITION DROP VALUES`	N/A
Moving Partitions, refer to About Moving Partitions and Subpartitions	`MOVE SUBPARTITION`	`MOVE SUBPARTITION`	`MOVE PARTITION`	`MOVE SUBPARTITION`	`MOVE PARTITION`
Renaming Partitions, refer to About Renaming Partitions and Subpartitions	`RENAME PARTITION`	`RENAME PARTITION`	`RENAME PARTITION`	`RENAME PARTITION`	`RENAME PARTITION`
Splitting Partitions, refer to About Splitting Partitions and Subpartitions	`SPLIT PARTITION`, single and multiple partitions	`SPLIT PARTITION`, single and multiple partitions	N/A	`SPLIT PARTITION`, single and multiple partitions	N/A (These operations cannot be performed on reference-partitioned tables. If performed on a parent table, then these operations cascade to all descendant tables.)
Truncating Partitions, refer to About Truncating Partitions and Subpartitions	`TRUNCATE PARTITION`, single and multiple partitions	`TRUNCATE PARTITION`, single and multiple partitions	`TRUNCATE PARTITION`, single and multiple partitions	`TRUNCATE PARTITION`, single and multiple partitions	`TRUNCATE PARTITION`, single and multiple partitions

Table 4-2 ALTER TABLE Maintenance Operations for Table Subpartitions

Maintenance Operation	Composite *-Range	Composite *-Hash	Composite *-List
Adding Subpartitions, refer to About Adding Partitions and Subpartitions	`MODIFY PARTITION ADD SUBPARTITION`, single and multiple subpartitions	`MODIFY PARTITION ADD SUBPARTITION`	`MODIFY PARTITION ADD SUBPARTITION`, single and multiple subpartitions
Coalescing Subpartitions, refer to About Coalescing Partitions and Subpartitions	N/A	`MODIFY PARTITION COALESCE SUBPARTITION`	N/A
Dropping Subpartitions, refer to About Dropping Partitions and Subpartitions	`DROP SUBPARTITION`, single and multiple subpartitions	N/A	`DROP SUBPARTITION`, single and multiple subpartitions
Exchanging Subpartitions, refer to About Exchanging Partitions and Subpartitions	`EXCHANGE SUBPARTITION`	N/A	`EXCHANGE SUBPARTITION`
Merging Subpartitions, refer to About Merging Partitions and Subpartitions	`MERGE SUBPARTITIONS`, single and multiple subpartitions	N/A	`MERGE SUBPARTITIONS`, single and multiple subpartitions
About Modifying Default Attributes	`MODIFY DEFAULT ATTRIBUTES FOR PARTITION`	`MODIFY DEFAULT ATTRIBUTES FOR PARTITION`	`MODIFY DEFAULT ATTRIBUTES FOR PARTITION`
Modifying Real Attributes of Subpartitions, refer to About Modifying Real Attributes of Partitions	`MODIFY SUBPARTITION`	`MODIFY SUBPARTITION`	`MODIFY SUBPARTITION`
Modifying List Subpartitions, refer to About Modifying List Partitions: Adding Values	N/A	N/A	`MODIFY SUBPARTITION ADD VALUES`
Modifying List Subpartitions, refer to About Modifying List Partitions: Dropping Values	N/A	N/A	`MODIFY SUBPARTITION DROP VALUES`
Modifying a Subpartition Template	`SET SUBPARTITION TEMPLATE`	`SET SUBPARTITION TEMPLATE`	`SET SUBPARTITION TEMPLATE`
Moving Subpartitions, refer to About Moving Partitions and Subpartitions	`MOVE SUBPARTITION`	`MOVE SUBPARTITION`	`MOVE SUBPARTITION`
Renaming Subpartitions, refer to About Renaming Partitions and Subpartitions	`RENAME SUBPARTITION`	`RENAME SUBPARTITION`	`RENAME SUBPARTITION`
Splitting Subpartitions, refer to About Splitting Partitions and Subpartitions	`SPLIT SUBPARTITION`, single and multiple subpartitions	N/A	`SPLIT SUBPARTITION`, single and multiple subpartitions
Truncating Subpartitions, refer to About Truncating Partitions and Subpartitions	`TRUNCATE SUBPARTITION`, single and multiple subpartitions	`TRUNCATE SUBPARTITION`, single and multiple subpartitions	`TRUNCATE SUBPARTITION`, single and multiple subpartitions

> **Note:**
>
> The first time you use table compression to introduce a compressed partition into a partitioned table that has bitmap indexes and that currently contains only uncompressed partitions, you must do the following:
>
> - Either drop all existing bitmap indexes and bitmap index partitions, or mark them `UNUSABLE`.
> - Set the table compression attribute.
> - Rebuild the indexes.
>
> These actions are independent of whether any partitions contain data and of the operation that introduces the compressed partition.
>
> This does not apply to partitioned tables with B-tree indexes or to partitioned index-organized tables.

Table 4-3 lists maintenance operations that can be performed on index partitions, and indicates on which type of index (global or local) they can be performed. The `ALTER INDEX` clause used for the maintenance operation is shown.

Global indexes do not reflect the structure of the underlying table. If partitioned, they can be partitioned by range or hash.

Because local indexes reflect the underlying structure of the table, partitioning is maintained automatically when table partitions and subpartitions are affected by maintenance activity. Therefore, partition maintenance on local indexes is less necessary and there are fewer options.

Table 4-3 ALTER INDEX Maintenance Operations for Index Partitions

Maintenance Operation	Type of Index	Type of Index Partitioning		
		Range	**Hash and List**	**Composite**
Adding Index Partitions	Global	–	`ADD PARTITION` (hash only)	-
Adding Index Partitions	Local	N/A	N/A	N/A
Dropping Index Partitions	Global	`DROP PARTITION`	-	-
Dropping Index Partitions	Local	N/A	N/A	N/A
Modifying Default Attributes of Index Partitions	Global	`MODIFY DEFAULT ATTRIBUTES`	-	-
Modifying Default Attributes of Index Partitions	Local	`MODIFY DEFAULT ATTRIBUTES`	`MODIFY DEFAULT ATTRIBUTES`	`MODIFY DEFAULT ATTRIBUTES` `MODIFY DEFAULT ATTRIBUTES FOR PARTITION`
Modifying Real Attributes of Index Partitions	Global	`MODIFY PARTITION`	-	-
Modifying Real Attributes of Index Partitions	Local	`MODIFY PARTITION`	`MODIFY PARTITION`	`MODIFY PARTITION` `MODIFY SUBPARTITION`

Table 4-3 (Cont.) ALTER INDEX Maintenance Operations for Index Partitions

Maintenance Operation	Type of Index	Type of Index Partitioning		
		Range	Hash and List	Composite
About Rebuilding Index Partitions	Global	`REBUILD PARTITION`	-	-
About Rebuilding Index Partitions	Local	`REBUILD PARTITION`	`REBUILD PARTITION`	`REBUILD SUBPARTITION`
About Renaming Index Partitions	Global	`RENAME PARTITION`	-	-
About Renaming Index Partitions	Local	`RENAME PARTITION`	`RENAME PARTITION`	`RENAME PARTITION` `RENAME SUBPARTITION`
Splitting Index Partitions	Global	`SPLIT PARTITION`	-	-
Splitting Index Partitions	Local	N/A	N/A	N/A

4.3.1 Updating Indexes Automatically

Before discussing the individual maintenance operations for partitioned tables and indexes, it is important to discuss the effects of the `UPDATE INDEXES` clause that can be specified in the `ALTER TABLE` statement.

By default, many table maintenance operations on partitioned tables invalidate (mark `UNUSABLE`) the corresponding indexes or index partitions. You must then rebuild the entire index or, for a global index, each of its partitions. The database lets you override this default behavior if you specify `UPDATE INDEXES` in your `ALTER TABLE` statement for the maintenance operation. Specifying this clause tells the database to update the indexes at the time it executes the maintenance operation DDL statement. This provides the following benefits:

- The indexes are updated with the base table operation. You are not required to update later and independently rebuild the indexes.
- The global indexes are more highly available, because they are not marked `UNUSABLE`. These indexes remain available even while the partition DDL is executing and can access unaffected partitions in the table.
- You need not look up the names of all invalid indexes to rebuild them.

Optional clauses for local indexes let you specify physical and storage characteristics for updated local indexes and their partitions.

- You can specify physical attributes, tablespace storage, and logging for each partition of each local index. Alternatively, you can specify only the `PARTITION` keyword and let the database update the partition attributes as follows:
  - For operations on a single table partition (such as `MOVE PARTITION` and `SPLIT PARTITION`), the corresponding index partition inherits the attributes of the affected table partition. The database does not generate names for new index partitions, so any new index partitions resulting from this operation inherit their names from the corresponding new table partition.
  - For `MERGE PARTITION` operations, the resulting local index partition inherits its name from the resulting table partition and inherits its attributes from the local index.

- For a composite-partitioned index, you can specify tablespace storage for each subpartition.

The following operations support the `UPDATE INDEXES` clause:

- `ADD PARTITION | SUBPARTITION`
- `COALESCE PARTITION | SUBPARTITION`
- `DROP PARTITION | SUBPARTITION`
- `EXCHANGE PARTITION | SUBPARTITION`
- `MERGE PARTITION | SUBPARTITION`
- `MOVE PARTITION | SUBPARTITION`
- `SPLIT PARTITION | SUBPARTITION`
- `TRUNCATE PARTITION | SUBPARTITION`

SKIP_UNUSABLE_INDEXES Initialization Parameter

`SKIP_UNUSABLE_INDEXES` is an initialization parameter with a default value of `TRUE`. This setting disables error reporting of indexes and index partitions marked `UNUSABLE`. If you do not want the database to choose an alternative execution plan to avoid the unusable elements, then you should set this parameter to `FALSE`.

Considerations when Updating Indexes Automatically

The following implications are worth noting when you specify `UPDATE INDEXES`:

- The partition DDL statement can take longer to execute, because indexes that were previously marked `UNUSABLE` are updated. However, you must compare this increase with the time it takes to execute DDL without updating indexes, and then rebuild all indexes. A rule of thumb is that it is faster to update indexes if the size of the partition is less that 5% of the size of the table.
- The `EXCHANGE` operation is no longer a fast operation. Again, you must compare the time it takes to do the DDL and then rebuild all indexes.
- When you update a table with a global index:
  - The index is updated in place. The updates to the index are logged, and redo and undo records are generated. In contrast, if you rebuild an entire global index, you can do so in `NOLOGGING` mode.
  - Rebuilding the entire index manually creates a more efficient index, because it is more compact with better space utilization.
- The `UPDATE INDEXES` clause is not supported for index-organized tables. However, the `UPDATE GLOBAL INDEXES` clause may be used with `DROP PARTITION`, `TRUNCATE PARTITION`, and `EXCHANGE PARTITION` operations to keep the global indexes on index-organized tables usable. For the remaining operations in the above list, global indexes on index-organized tables remain usable. In addition, local index partitions on index-organized tables remain usable after a `MOVE PARTITION` operation.

> **See Also:**
>
> *Oracle Database SQL Language Reference* for information about the *update_all_indexes_clause* of ALTER TABLE in for the syntax for updating indexes

4.3.2 Asynchronous Global Index Maintenance for Dropping and Truncating Partitions

The partition maintenance operations DROP PARTITION and TRUNCATE PARTITION are optimized by making the index maintenance for metadata only.

Asynchronous global index maintenance for DROP and TRUNCATE is performed by default; however, the UPDATE INDEXES clause is still required for backward compatibility.

The following list summarizes the limitations of asynchronous global index maintenance:

- Only performed on heap tables
- No support for tables with object types
- No support for tables with domain indexes
- Not performed for the user SYS

Maintenance operations on indexes can be performed with the automatic scheduler job SYS.PMO_DEFERRED_GIDX_MAINT_JOB to clean up all global indexes. This job is scheduled to run on a regular basis by default. You can run this job at any time using DBMS_SCHEDULER.RUN_JOB if you want to proactively clean up the indexes. You can also modify the job to run with a schedule based on your specific requirements. Oracle recommends that you do not drop the job.

You can also force cleanup of an index needing maintenance using one of the following options:

- DBMS_PART.CLEANUP_GIDX - This PL/SQL procedure gathers the list of global indexes in the system that may require cleanup and runs the operations necessary to restore the indexes to a clean state.
- ALTER INDEX REBUILD [PARTITION] – This SQL statement rebuilds the entire index or index partition as is done in releases previous to Oracle Database 12*c* Release 1 (12.1). The resulting index (partition) does not contain any stale entries.
- ALTER INDEX [PARTITION] COALESCE CLEANUP – This SQL statement cleans up any orphaned entries in index blocks.

> **See Also:**
>
> *Oracle Database Administrator's Guide* for information about managing jobs with Oracle Scheduler

4.3.3 Modifying a Subpartition Template

You can modify a subpartition template of a composite partitioned table by replacing it with a new subpartition template.

Any subsequent operations that use the subpartition template (such as `ADD PARTITION` or `MERGE PARTITIONS`) now use the new subpartition template. Existing subpartitions remain unchanged.

If you modify a subpartition template of an interval-* composite partitioned table, then interval partitions that have not yet been created use the new subpartition template.

Use the `ALTER TABLE SET SUBPARTITION TEMPLATE` statement to specify a new subpartition template. For example:

```
ALTER TABLE employees_sub_template
   SET SUBPARTITION TEMPLATE
         (SUBPARTITION e TABLESPACE ts1,
          SUBPARTITION f TABLESPACE ts2,
          SUBPARTITION g TABLESPACE ts3,
          SUBPARTITION h TABLESPACE ts4
         );
```

You can drop a subpartition template by specifying an empty list:

```
ALTER TABLE employees_sub_template
   SET SUBPARTITION TEMPLATE ( );
```

4.3.4 Filtering Maintenance Operations

Partition maintenance operations support the addition of data filtering, enabling the combination of partition and data maintenance.

A filtered partition maintenance operation only preserves the data satisfying the data filtering as part of the partition maintenance. The capability of data filtering applies to `MOVE PARTITION`, `MERGE PARTITION`, and `SPLIT PARTITION` .

Example 4-27 shows the use of the `ALTER TABLE` statement to move a partition while removing all orders that are not open (closed orders).

The filtering predicate must be on the partitioned table. All partition maintenance operations that can be performed online (`MOVE` and `SPLIT`) can also be performed as filtered partition maintenance operations. With `ONLINE` specified, DML operations on the partitions being maintained are allowed.

Filtered partition maintenance operations performed in online mode do not enforce the filter predicate on concurrent ongoing DML operations. The filter condition is only applied one time at the beginning of the partition maintenance operation. Consequently, any subsequent DML succeeds, but is ignored from a filtering perspective. Records that do not match the filter condition when the partition maintenance started are not preserved, regardless of any DML operation. Newly inserted records are inserted if they match the partition key criteria, regardless of whether they satisfy the filter condition of the partition maintenance operation. Filter conditions are limited to the partitioned table itself and do not allow any reference to other tables, such as a join or subquery expression.

Consider the following scenarios when the keyword `ONLINE` is specified in the SQL statement of Example 4-27.

- An existing order record in partition `q1_2016` that is updated to `status='open'` after the partition maintenance operation has started is not be preserved in the partition.
- A new order record with `status='closed'` can be inserted in partition `q1_2016` after the partition maintenance operation has started and while the partition maintenance operation is ongoing.

Example 4-27 Using a filtering clause when performing maintenance operations

```
ALTER TABLE orders_move_part
  MOVE PARTITION q1_2016 TABLESPACE open_orders COMPRESS ONLINE
  INCLUDING ROWS WHERE order_state = 'open';
```

> **See Also:**
>
> *Oracle Database SQL Language Reference* for the exact syntax of the partitioning clauses for creating and altering partitioned tables and indexes, any restrictions on their use, and specific privileges required for creating and altering tables

4.4 Maintenance Operations for Partitioned Tables and Indexes

There are various maintenance operations that can be performed on partitioned tables and indexes.

The operations to perform partition and subpartition maintenance for both tables and indexes are discussed in the following topics.

- About Adding Partitions and Subpartitions
- About Coalescing Partitions and Subpartitions
- About Dropping Partitions and Subpartitions
- About Exchanging Partitions and Subpartitions
- About Merging Partitions and Subpartitions
- About Modifying Attributes of Tables, Partitions, and Subpartitions
- About Modifying List Partitions
- About Modifying the Partitioning Strategy
- About Moving Partitions and Subpartitions
- About Rebuilding Index Partitions
- About Renaming Partitions and Subpartitions
- About Splitting Partitions and Subpartitions
- About Truncating Partitions and Subpartitions

Note:

- Where the usability of indexes or index partitions affected by the maintenance operation is discussed, consider the following:
  - Only indexes and index partitions that are *not* empty are candidates for being marked `UNUSABLE`. If they are empty, the `USABLE`/`UNUSABLE` status is left unchanged.
  - Only indexes or index partitions with `USABLE` status are updated by subsequent DML.
- `SPLIT`, `MERGE`, `MOVE`, `TRUNCATE`, and `DROP` operations on partitions and subpartitions will result in `ORPHANED_ENTRIES AS YES`. This is expected behavior.

See Also:

- *Oracle Database Administrator's Guide* for information about managing tables
- *Oracle Database SQL Language Reference* for the exact syntax of the partitioning clauses for altering partitioned tables and indexes, any restrictions on their use, and specific privileges required for creating and altering tables

4.4.1 Restrictions and Special Requirements for Multi-Partition Operations on Tables with Domain Indexes

These constraints apply to multi-partition `ADD`, `MERGE`, `SPLIT`, `DROP`, and `TRUNCATE` operations on a table which has a domain index created on it.

About Tables With a Local Domain Index

If a table has a local domain index, then `ADD`, `SPLIT`, and `DROP` operations on multiple partitions or subpartitions in the same statement are not allowed. Each partition or subpartition can be operated on in a separate statement only.

For `MERGE` operations on a table with a local domain index, two partitions are allowed.

If a table has a local domain index, performing `TRUNCATE` on a single partition or the entire table is supported. Truncating multiple partitions in a single `ALTER TABLE` statement is not permitted if a local domain index is present.

About Tables With a Global Domain Index

The restrictions on multi-partition operations described above apply to local domain indexes only. For tables with global domain indexes, multi-partition operations are supported.

> **Note:**
>
> For a table that has a global index, when you use a single statement to operate on multiple partitions it is important to include an `UPDATE INDEXES` clause in the statement, otherwise the index is rendered unusable.

Checking for Indexes

Before performing `ADD`, `SPLIT`, `DROP` or `MERGE` operations on multiple partitions, you can check for indexes through the `ALL_INDEXES` view. For example:

```
SELECT owner, index_name FROM SYS.ALL_INDEXES WHERE index_type IN ('DOMAIN',
'FUNCTION-BASED DOMAIN');
```

The query can be narrowed down by adding specific `TABLE_OWNER` and `TABLE_NAME` predicates. For example:

```
SELECT owner, index_name FROM SYS.ALL_INDEXES WHERE index_type IN ('DOMAIN',
'FUNCTION-BASED DOMAIN') WHERE table_owner = 'MY_USER' AND table_name =
'MY_PART_TABLE';
```

4.4.2 About Adding Partitions and Subpartitions

This section introduces how to manually add new partitions to a partitioned table and explains why partitions cannot be specifically added to most partitioned indexes.

This section contains the following topics:

- Adding a Partition to a Range-Partitioned Table
- Adding a Partition to a Hash-Partitioned Table
- Adding a Partition to a List-Partitioned Table
- Adding a Partition to an Interval-Partitioned Table
- About Adding Partitions to a Composite *-Hash Partitioned Table
- About Adding Partitions to a Composite *-List Partitioned Table
- About Adding Partitions to a Composite *-Range Partitioned Table
- About Adding a Partition or Subpartition to a Reference-Partitioned Table
- Adding Index Partitions
- Adding Multiple Partitions

> **Note:**
>
> See Restrictions and Special Requirements for Multi-Partition Operations on Tables with Domain Indexes

4.4.2.1 Adding a Partition to a Range-Partitioned Table

You can add a partition after the last existing partition of a table or the beginning of a table or in the middle of a table.

Use the `ALTER TABLE ADD PARTITION` statement to add a new partition to the "high" end (the point after the last existing partition). To add a partition at the beginning or in the middle of a table, use the `SPLIT PARTITION` clause.

For example, consider the table, `sales`, which contains data for the current month in addition to the previous 12 months. On January 1, 1999, you add a partition for January, which is stored in tablespace `tsx`.

```
ALTER TABLE sales
      ADD PARTITION jan99 VALUES LESS THAN ( '01-FEB-1999' )
      TABLESPACE tsx;
```

Local and global indexes associated with the range-partitioned table remain usable.

4.4.2.2 Adding a Partition to a Hash-Partitioned Table

When you add a partition to a hash partitioned table, the database populates the new partition with rows rehashed from an existing partition (selected by the database) as determined by the hash function.

Consequently, if the table contains data, then it may take some time to add a hash partition.

The following statements show two ways of adding a hash partition to table `scubagear`. Choosing the first statement adds a new hash partition whose partition name is system generated, and which is placed in the default tablespace. The second statement also adds a new hash partition, but that partition is explicitly named `p_named` and is created in tablespace `gear5`.

```
ALTER TABLE scubagear ADD PARTITION;

ALTER TABLE scubagear
      ADD PARTITION p_named TABLESPACE gear5;
```

Indexes may be marked `UNUSABLE` as explained in the following table:

Table Type	Index Behavior
Regular (Heap)	Unless you specify `UPDATE INDEXES` as part of the `ALTER TABLE` statement: • The local indexes for the new partition, and for the existing partition from which rows were redistributed, are marked `UNUSABLE` and must be rebuilt. • All global indexes, or all partitions of partitioned global indexes, are marked `UNUSABLE` and must be rebuilt.
Index-organized	• For local indexes, the behavior is identical to heap tables. • All global indexes remain usable.

4.4.2.3 Adding a Partition to a List-Partitioned Table

The example in this topic shows how to add a partition to a list-partitioned table.

The following statement illustrates how to add a new partition to a list-partitioned table. In this example, physical attributes and `NOLOGGING` are specified for the partition being added.

```
ALTER TABLE q1_sales_by_region
   ADD PARTITION q1_nonmainland VALUES ('HI', 'PR')
      STORAGE (INITIAL 20K NEXT 20K) TABLESPACE tbs_3
      NOLOGGING;
```

Any value in the set of literal values that describe the partition being added must not exist in any of the other partitions of the table.

You cannot add a partition to a list-partitioned table that has a default partition, but you can split the default partition. By doing so, you effectively create a new partition defined by the values that you specify, and a second partition that remains the default partition.

Local and global indexes associated with the list-partitioned table remain usable.

4.4.2.4 Adding a Partition to an Interval-Partitioned Table

You cannot explicitly add a partition to an interval-partitioned table. The database automatically creates a partition for an interval when data for that interval is inserted.

However, exchanging a partition of an interval-partitioned table that has not been materialized in the data dictionary, meaning to have an explicit entry in the data dictionary beyond the interval definition, you must manually materialize the partition using the `ALTER TABLE LOCK PARTITION` command.

To change the interval for future partitions, use the `SET INTERVAL` clause of the `ALTER TABLE` statement. The `SET INTERVAL` clause converts existing interval partitions to range partitions, determines the high value of the defined range partitions, and automatically creates partitions of the specified interval as needed for data that is beyond that high value. As a side effect, an interval-partitioned table does not have the notation of `MAXVALUES`.

You also use the `SET INTERVAL` clause to migrate an existing range partitioned or range-* composite partitioned table into an interval or interval-* partitioned table. To disable the creation of future interval partitions, and effectively revert to a range-partitioned table, use an empty value in the `SET INTERVAL` clause. Created interval partitions are transformed into range partitions with their current high values.

To increase the interval for date ranges, you must ensure that you are at a relevant boundary for the new interval. For example, if the highest interval partition boundary in your daily interval partitioned table transactions is January 30, 2007 and you want to change to a monthly partition interval, then the following statement results in an error:

```
ALTER TABLE transactions SET INTERVAL (NUMTOYMINTERVAL(1,'MONTH');

ORA-14767: Cannot specify this interval with existing high bounds
```

You must create another daily partition with a high bound of February 1, 2007 to successfully change to a monthly interval:

```
LOCK TABLE transactions PARTITION FOR(TO_DATE('31-JAN-2007','dd-MON-yyyy')
   IN SHARE MODE;

ALTER TABLE transactions SET INTERVAL (NUMTOYMINTERVAL(1,'MONTH');
```

The lower partitions of an interval-partitioned table are range partitions. You can split range partitions to add more partitions in the range portion of the interval-partitioned table.

To disable interval partitioning on the `transactions` table, use:

```
ALTER TABLE transactions SET INTERVAL ();
```

4.4.2.5 About Adding Partitions to a Composite *-Hash Partitioned Table

Partitions can be added at both the partition level and at the hash subpartition level.

- Adding a Partition to a *-Hash Partitioned Table
- Adding a Subpartition to a *-Hash Partitioned Table

4.4.2.5.1 Adding a Partition to a *-Hash Partitioned Table

The example in this topic shows how to add a new partition to a [range | list | interval]-hash partitioned table.

For an interval-hash partitioned table, interval partitions are automatically created. You can specify a `SUBPARTITIONS` clause that lets you add a specified number of subpartitions, or a `SUBPARTITION` clause for naming specific subpartitions. If no `SUBPARTITIONS` or `SUBPARTITION` clause is specified, then the partition inherits table level defaults for subpartitions. For an interval-hash partitioned table, you can only add subpartitions to range or interval partitions that have been materialized.

This example adds a range partition `q1_2000` to the range-hash partitioned table `sales`, which is populated with data for the first quarter of the year 2000. There are eight subpartitions stored in tablespace `tbs5`. The subpartitions cannot be set explicitly to use table compression. Subpartitions inherit the compression attribute from the partition level and are stored in a compressed form in this example:

```
ALTER TABLE sales ADD PARTITION q1_2000
      VALUES LESS THAN (2000, 04, 01) COMPRESS
      SUBPARTITIONS 8 STORE IN tbs5;
```

4.4.2.5.2 Adding a Subpartition to a *-Hash Partitioned Table

Use the `MODIFY PARTITION ADD SUBPARTITION` clause of the `ALTER TABLE` statement to add a hash subpartition to a [range | list | interval]-hash partitioned table.

The newly added subpartition is populated with rows rehashed from other subpartitions of the same partition as determined by the hash function. For an interval-hash partitioned table, you can only add subpartitions to range or interval partitions that have been materialized.

In the following example, a new hash subpartition `us_loc5`, stored in tablespace `us1`, is added to range partition `locations_us` in table `diving`.

```
ALTER TABLE diving MODIFY PARTITION locations_us
      ADD SUBPARTITION us_locs5 TABLESPACE us1;
```

Index subpartitions corresponding to the added and rehashed subpartitions must be rebuilt unless you specify `UPDATE INDEXES`.

4.4.2.6 About Adding Partitions to a Composite *-List Partitioned Table

Partitions can be added at both the partition level and at the list subpartition level.

- Adding a Partition to a *-List Partitioned Table
- Adding a Subpartition to a *-List Partitioned Table

4.4.2.6.1 Adding a Partition to a *-List Partitioned Table

The example in this topic shows how to add a new partition to a [range | list | interval]-list partitioned table.

The database automatically creates interval partitions as data for a specific interval is inserted. You can specify `SUBPARTITION` clauses for naming and providing value lists for the subpartitions. If no `SUBPARTITION` clauses are specified, then the partition inherits the subpartition template. If there is no subpartition template, then a single default subpartition is created.

The statement in Example 4-28 adds a new partition to the `quarterly_regional_sales` table that is partitioned by the range-list method. Some new physical attributes are specified for this new partition while table-level defaults are inherited for those that are not specified.

Example 4-28 Adding partitions to a range-list partitioned table

```
ALTER TABLE quarterly_regional_sales
   ADD PARTITION q1_2000 VALUES LESS THAN (TO_DATE('1-APR-2000','DD-MON-YYYY'))
      STORAGE (INITIAL 20K NEXT 20K) TABLESPACE ts3 NOLOGGING
         (
          SUBPARTITION q1_2000_northwest VALUES ('OR', 'WA'),
          SUBPARTITION q1_2000_southwest VALUES ('AZ', 'UT', 'NM'),
          SUBPARTITION q1_2000_northeast VALUES ('NY', 'VM', 'NJ'),
          SUBPARTITION q1_2000_southeast VALUES ('FL', 'GA'),
          SUBPARTITION q1_2000_northcentral VALUES ('SD', 'WI'),
          SUBPARTITION q1_2000_southcentral VALUES ('OK', 'TX')
         );
```

4.4.2.6.2 Adding a Subpartition to a *-List Partitioned Table

Use the `MODIFY PARTITION ADD SUBPARTITION` clause of the `ALTER TABLE` statement to add a list subpartition to a [range | list | interval]-list partitioned table.

For an interval-list partitioned table, you can only add subpartitions to range or interval partitions that have been materialized.

The following statement adds a new subpartition to the existing set of subpartitions in the range-list partitioned table `quarterly_regional_sales`. The new subpartition is created in tablespace `ts2`.

```
ALTER TABLE quarterly_regional_sales
   MODIFY PARTITION q1_1999
      ADD SUBPARTITION q1_1999_south
         VALUES ('AR','MS','AL') tablespace ts2;
```

4.4.2.7 About Adding Partitions to a Composite *-Range Partitioned Table

Partitions can be added at both the partition level and at the range subpartition level.

- Adding a Partition to a *-Range Partitioned Table
- Adding a Subpartition to a *-Range Partitioned Table

4.4.2.7.1 Adding a Partition to a *-Range Partitioned Table

The example in this topic shows how to add a new partition to a [range | list | interval]-range partitioned table.

The database automatically creates interval partitions for an interval-range partitioned table when data is inserted in a specific interval. You can specify a `SUBPARTITION` clause for naming and providing ranges for specific subpartitions. If no `SUBPARTITION` clause is specified, then the partition inherits the subpartition template specified at the table level. If there is no subpartition template, then a single subpartition with a maximum value of `MAXVALUE` is created.

Example 4-29 adds a range partition `p_2007_jan` to the range-range partitioned table `shipments`, which is populated with data for the shipments ordered in January 2007. There are three subpartitions. Subpartitions inherit the compression attribute from the partition level and are stored in a compressed form in this example:

Example 4-29 Adding partitions to a range-range partitioned table

```
ALTER TABLE shipments
   ADD PARTITION p_2007_jan
      VALUES LESS THAN (TO_DATE('01-FEB-2007','dd-MON-yyyy')) COMPRESS
      ( SUBPARTITION p07_jan_e VALUES LESS THAN (TO_DATE('15-FEB-2007','dd-MON-yyyy'))
      , SUBPARTITION p07_jan_a VALUES LESS THAN (TO_DATE('01-MAR-2007','dd-MON-yyyy'))
      , SUBPARTITION p07_jan_l VALUES LESS THAN (TO_DATE('01-APR-2007','dd-MON-yyyy'))
      ) ;
```

4.4.2.7.2 Adding a Subpartition to a *-Range Partitioned Table

You use the `MODIFY PARTITION ADD SUBPARTITION` clause of the `ALTER TABLE` statement to add a range subpartition to a [range | list | interval]-range partitioned table.

For an interval-range partitioned table, you can only add partitions to range or interval partitions that have been materialized.

The following example adds a range subpartition to the `shipments` table that contains all values with an `order_date` in January 2007 and a `delivery_date` on or after April 1, 2007.

```
ALTER TABLE shipments
   MODIFY PARTITION p_2007_jan
      ADD SUBPARTITION p07_jan_vl VALUES LESS THAN (MAXVALUE) ;
```

4.4.2.8 About Adding a Partition or Subpartition to a Reference-Partitioned Table

A partition or subpartition can be added to a parent table in a reference partition definition just as partitions and subpartitions can be added to a range, hash, list, or composite partitioned table.

The add operation automatically cascades to any descendant reference partitioned tables. The `DEPENDENT TABLES` clause can set specific properties for dependent tables when you add partitions or subpartitions to a parent table.

> **See Also:**
>
> *Oracle Database SQL Language Reference*

4.4.2.9 Adding Index Partitions

You cannot explicitly add a partition to a local index. Instead, a new partition is added to a local index only when you add a partition to the underlying table.

Specifically, when there is a local index defined on a table and you issue the `ALTER TABLE` statement to add a partition, a matching partition is also added to the local index. The database assigns names and default physical storage attributes to the new index partitions, but you can rename or alter them after the `ADD PARTITION` operation is complete.

You can effectively specify a new tablespace for an index partition in an `ADD PARTITION` operation by first modifying the default attributes for the index. For example, assume that a local index, `q1_sales_by_region_locix`, was created for list partitioned table `q1_sales_by_region`. If before adding the new partition `q1_nonmainland`, as shown in Adding a Partition to a List-Partitioned Table, you had issued the following statement, then the corresponding index partition would be created in tablespace `tbs_4`.

```
ALTER INDEX q1_sales_by_region_locix
   MODIFY DEFAULT ATTRIBUTES TABLESPACE tbs_4;
```

Otherwise, it would be necessary for you to use the following statement to move the index partition to `tbs_4` after adding it:

```
ALTER INDEX q1_sales_by_region_locix
   REBUILD PARTITION q1_nonmainland TABLESPACE tbs_4;
```

You can add a partition to a hash partitioned global index using the `ADD PARTITION` syntax of `ALTER INDEX`. The database adds hash partitions and populates them with index entries rehashed from an existing hash partition of the index, as determined by the hash function. The following statement adds a partition to the index `hgidx` shown in Creating a Hash Partitioned Global Index:

```
ALTER INDEX hgidx ADD PARTITION p5;
```

You cannot add a partition to a range-partitioned global index, because the highest partition always has a partition bound of `MAXVALUE`. To add a new highest partition, use the `ALTER INDEX SPLIT PARTITION` statement.

4.4.2.10 Adding Multiple Partitions

You can add multiple new partitions and subpartitions with the `ADD PARTITION` and `ADD SUBPARTITION` clauses of the same `ALTER TABLE` statement.

When adding multiple partitions, local and global index operations are the same as when adding a single partition. Adding multiple partitions and subpartitions is only supported for range, list, and system partitions and subpartitions.

You can add multiple range partitions that are listed in ascending order of their upper bound values to the high end (after the last existing partition) of a range-partitioned or composite range-partitioned table, provided the `MAXVALUE` partition is not defined. Similarly, you can add multiple list partitions to a table using new sets of partition values if the `DEFAULT` partition does not exist.

Multiple system partitions can be added using a single SQL statement by specifying the individual partitions. For example, the following SQL statement adds multiple partitions to the range-partitioned `sales` table created in Example 4-1:

```
ALTER TABLE sales ADD
  PARTITION sales_q1_2007 VALUES LESS THAN (TO_DATE('01-APR-2007','dd-MON-yyyy')),
  PARTITION sales_q2_2007 VALUES LESS THAN (TO_DATE('01-JUL-2007','dd-MON-yyyy')),
  PARTITION sales_q3_2007 VALUES LESS THAN (TO_DATE('01-OCT-2007','dd-MON-yyyy')),
  PARTITION sales_q4_2007 VALUES LESS THAN (TO_DATE('01-JAN-2008','dd-MON-yyyy'))
;
```

You can use the `BEFORE` clause to add multiple new system partitions in relation to only one existing partition. The following SQL statements provide an example of adding multiple individual partitions using the `BEFORE` clause:

```
CREATE TABLE system_part_tab1 (number1 integer, number2 integer)
PARTITION BY SYSTEM
( PARTITION p1,
  PARTITION p2,
  PARTITION p3,
  PARTITION p_last);

ALTER TABLE system_part_tab1 ADD
  PARTITION p4,
  PARTITION p5,
  PARTITION p6
  BEFORE PARTITION p_last;

SELECT SUBSTR(TABLE_NAME,1,18) table_name, TABLESPACE_NAME,
   SUBSTR(PARTITION_NAME,1,16) partition_name
   FROM USER_TAB_PARTITIONS WHERE TABLE_NAME='SYSTEM_PART_TAB1';
TABLE_NAME         TABLESPACE_NAME                PARTITION_NAME
------------------ ------------------------------ ----------------
SYSTEM_PART_TAB1   USERS                          P_LAST
SYSTEM_PART_TAB1   USERS                          P6
SYSTEM_PART_TAB1   USERS                          P5
SYSTEM_PART_TAB1   USERS                          P4
SYSTEM_PART_TAB1   USERS                          P3
SYSTEM_PART_TAB1   USERS                          P2
SYSTEM_PART_TAB1   USERS                          P1
```

> **Note:**
>
> See Restrictions and Special Requirements for Multi-Partition Operations on Tables with Domain Indexes

4.4.3 About Coalescing Partitions and Subpartitions

Coalescing partitions is a way of reducing the number of partitions in a hash partitioned table or index, or the number of subpartitions in a *-hash partitioned table.

When a hash partition is coalesced, its contents are redistributed into one or more remaining partitions determined by the hash function. The specific partition that is coalesced is selected by the database, and is dropped after its contents have been redistributed. If you coalesce a hash partition or subpartition in the parent table of a reference-partitioned table definition, then the reference-partitioned table automatically inherits the new partitioning definition.

Index partitions may be marked `UNUSABLE` as explained in the following table:

Table Type	Index Behavior
Regular (Heap)	Unless you specify `UPDATE INDEXES` as part of the `ALTER TABLE` statement: • Any local index partition corresponding to the selected partition is also dropped. Local index partitions corresponding to the one or more absorbing partitions are marked `UNUSABLE` and must be rebuilt. • All global indexes, or all partitions of partitioned global indexes, are marked `UNUSABLE` and must be rebuilt.
Index-organized	• Some local indexes are marked `UNUSABLE` as noted for heap indexes. • All global indexes remain usable.

This section contains the following topics:

- Coalescing a Partition in a Hash Partitioned Table
- Coalescing a Subpartition in a *-Hash Partitioned Table
- Coalescing Hash Partitioned Global Indexes

4.4.3.1 Coalescing a Partition in a Hash Partitioned Table

The `ALTER TABLE COALESCE PARTITION` statement is used to coalesce a partition in a hash partitioned table.

The following statement reduces by one the number of partitions in a table by coalescing a partition.

```
ALTER TABLE ouu1
     COALESCE PARTITION;
```

4.4.3.2 Coalescing a Subpartition in a *-Hash Partitioned Table

The `ALTER TABLE COALESCE SUBPARTITION` statement is used to coalesce a subpartition in a hash partitioned table.

The following statement distributes the contents of a subpartition of partition `us_locations` into one or more remaining subpartitions (determined by the hash function) of the same partition. For an interval-partitioned table, you can only coalesce hash subpartitions of materialized range or interval partitions. Basically, this operation is the inverse of the `MODIFY PARTITION ADD SUBPARTITION` clause discussed in Adding a Subpartition to a *-Hash Partitioned Table.

```
ALTER TABLE diving MODIFY PARTITION us_locations
     COALESCE SUBPARTITION;
```

4.4.3.3 Coalescing Hash Partitioned Global Indexes

You can instruct the database to reduce by one the number of index partitions in a hash partitioned global index using the `COALESCE PARTITION` clause of `ALTER INDEX`.

The database selects the partition to coalesce based on the requirements of the hash partition. The following statement reduces by one the number of partitions in the `hgidx` index, created in Creating a Hash Partitioned Global Index:

```
ALTER INDEX hgidx COALESCE PARTITION;
```

4.4.4 About Dropping Partitions and Subpartitions

You can drop partitions from range, interval, list, or composite *-[range | list] partitioned tables.

For interval partitioned tables, you can only drop range or interval partitions that have been materialized. For hash partitioned tables, or hash subpartitions of composite *-hash partitioned tables, you must perform a coalesce operation instead.

You cannot drop a partition from a reference-partitioned table. Instead, a drop operation on a parent table cascades to all descendant tables.

This section contains the following topics:

- Dropping Table Partitions
- Dropping Interval Partitions
- Dropping Index Partitions
- Dropping Multiple Partitions

4.4.4.1 Dropping Table Partitions

To drop table partitions, use `DROP PARTITION` or `DROP SUBPARTITION` with the `ALTER TABLE` SQL statement.

The following statements drop a table partition or subpartition:

- `ALTER TABLE DROP PARTITION` to drop a table partition
- `ALTER TABLE DROP SUBPARTITION` to drop a subpartition of a composite *-[range | list] partitioned table

To preserve the data in the partition, use the `MERGE PARTITION` statement instead of the `DROP PARTITION` statement.

To remove data in the partition without dropping the partition, use the `TRUNCATE PARTITION` statement.

If local indexes are defined for the table, then this statement also drops the matching partition or subpartitions from the local index. All global indexes, or all partitions of partitioned global indexes, are marked `UNUSABLE` unless either of the following is true:

- You specify `UPDATE INDEXES` (Cannot be specified for index-organized tables. Use `UPDATE GLOBAL INDEXES` instead.)
- The partition being dropped or its subpartitions are empty.

> **Note:**
>
> - If a table contains only one partition, you cannot drop the partition. Instead, you must drop the table.
> - You cannot drop the highest range partition in the range-partitioned section of an interval-partitioned or interval-* composite partitioned table.
> - With asynchronous global index maintenance, a drop partition update indexes operation is on metadata only and all global indexes remain valid.
> - Dropping a partition does not place the partition in the Oracle Database recycle bin, regardless of the setting of the recycle bin. Dropped partitions are immediately removed from the system.

The following sections contain some scenarios for dropping table partitions.

- Dropping a Partition from a Table that Contains Data and Global Indexes
- Dropping a Partition Containing Data and Referential Integrity Constraints

> **See Also:**
>
> - About Merging Partitions and Subpartitions for information about merging a partition
> - About Truncating Partitions and Subpartitions for information about truncating a partition
> - Asynchronous Global Index Maintenance for Dropping and Truncating Partitions for information about asynchronous index maintenance for dropping partitions

4.4.4.1.1 Dropping a Partition from a Table that Contains Data and Global Indexes

There are several methods you can use to drop a partition from a table that contains data and global indexes.

If the partition contains data and one or more global indexes are defined on the table, then use one of the following methods (method 1, 2 or 3) to drop the table partition.

Method 1

Issue the `ALTER TABLE DROP PARTITION` statement without maintaining global indexes. Afterward, you must rebuild any global indexes (whether partitioned or not) because the index (or index partitions) has been marked `UNUSABLE`. The following statements provide an example of dropping partition `dec98` from the `sales` table, then rebuilding its global nonpartitioned index.

```
ALTER TABLE sales DROP PARTITION dec98;
ALTER INDEX sales_area_ix REBUILD;
```

If index `sales_area_ix` were a range-partitioned global index, then all partitions of the index would require rebuilding. Further, it is not possible to rebuild all partitions of an index in one statement. You must issue a separate `REBUILD` statement for each partition in the index. The following statements rebuild the index partitions `jan99_ix` to `dec99_ix`.

```
ALTER INDEX sales_area_ix REBUILD PARTITION jan99_ix;
ALTER INDEX sales_area_ix REBUILD PARTITION feb99_ix;
ALTER INDEX sales_area_ix REBUILD PARTITION mar99_ix;
...
ALTER INDEX sales_area_ix REBUILD PARTITION dec99_ix;
```

This method is most appropriate for large tables where the partition being dropped contains a significant percentage of the total data in the table. While asynchronous global index maintenance keeps global indexes valid without the need of any index maintenance, you must use the `UPDATE INDEXES` clause to enable this new functionality. This behavior ensures backward compatibility.

Method 2

Issue the `DELETE` statement to delete all rows from the partition before you issue the `ALTER TABLE DROP PARTITION` statement. The `DELETE` statement updates the global indexes.

For example, to drop the first partition, issue the following statements:

```
DELETE FROM sales partition (dec98);
ALTER TABLE sales DROP PARTITION dec98;
```

This method is most appropriate for small tables, or for large tables when the partition being dropped contains a small percentage of the total data in the table.

Method 3

Specify `UPDATE INDEXES` in the `ALTER TABLE` statement. Doing so leverages the new asynchronous global index maintenance. Indexes remain valid.

```
ALTER TABLE sales DROP PARTITION dec98
     UPDATE INDEXES;
```

4.4.4.1.2 Dropping a Partition Containing Data and Referential Integrity Constraints

There are several methods you can use to drop a partition containing data and referential integrity constraints.

If a partition contains data and the table has referential integrity constraints, choose either of the following methods (method 1 or 2) to drop the table partition. This table has a local index only, so it is not necessary to rebuild any indexes.

Method 1

If there is no data referencing the data in the partition to drop, then you can disable the integrity constraints on the referencing tables, issue the `ALTER TABLE DROP PARTITION` statement, then re-enable the integrity constraints.

This method is most appropriate for large tables where the partition being dropped contains a significant percentage of the total data in the table. If there is still data referencing the data in the partition to be dropped, then ensure the removal of all the referencing data so that you can re-enable the referential integrity constraints.

Method 2

If there is data in the referencing tables, then you can issue the `DELETE` statement to delete all rows from the partition before you issue the `ALTER TABLE DROP PARTITION` statement. The `DELETE` statement enforces referential integrity constraints, and also fires triggers and generates redo and undo logs. The delete can succeed if you created the constraints with the `ON DELETE CASCADE` option, deleting all rows from referencing tables as well.

```
DELETE FROM sales partition (dec94);
ALTER TABLE sales DROP PARTITION dec94;
```

This method is most appropriate for small tables or for large tables when the partition being dropped contains a small percentage of the total data in the table.

4.4.4.2 Dropping Interval Partitions

You can drop interval partitions in an interval-partitioned table.

This operation drops the data for the interval only and leaves the interval definition in tact. If data is inserted in the interval just dropped, then the database again creates an interval partition.

You can also drop range partitions in an interval-partitioned table. The rules for dropping a range partition in an interval-partitioned table follow the rules for dropping a range partition in a range-partitioned table. If you drop a range partition in the middle of a set of range partitions, then the lower boundary for the next range partition shifts to the lower boundary of the range partition you just dropped. You cannot drop the highest range partition in the range-partitioned section of an interval-partitioned table.

The following example drops the September 2007 interval partition from the `sales` table. There are only local indexes so no indexes are invalidated.

```
ALTER TABLE sales DROP PARTITION FOR(TO_DATE('01-SEP-2007','dd-MON-yyyy'));
```

4.4.4.3 Dropping Index Partitions

You cannot explicitly drop a partition of a local index. Instead, local index partitions are dropped only when you drop a partition from the underlying table.

If a global index partition is empty, then you can explicitly drop it by issuing the `ALTER INDEX DROP PARTITION` statement. But, if a global index partition contains data, then dropping the partition causes the next highest partition to be marked `UNUSABLE`. For example, you would like to drop the index partition P1, and P2 is the next highest partition. You must issue the following statements:

```
ALTER INDEX npr DROP PARTITION P1;
ALTER INDEX npr REBUILD PARTITION P2;
```

> **Note:**
>
> You cannot drop the highest partition in a global index.

4.4.4.4 Dropping Multiple Partitions

You can remove multiple partitions or subpartitions from a range or list partitioned table with the `DROP PARTITION` and `DROP SUBPARTITION` clauses of the SQL `ALTER TABLE` statement.

For example, the following SQL statement drops multiple partitions from the range-partitioned table `sales`.

```
ALTER TABLE sales DROP PARTITION sales_q1_2008, sales_q2_2008,
     sales_q3_2008, sales_q4_2008;
```

You cannot drop all the partitions of a table. When dropping multiple partitions, local and global index operations are the same as when dropping a single partition.

> **See Also:**
>
> Restrictions and Special Requirements for Multi-Partition Operations on Tables with Domain Indexes.

4.4.5 About Exchanging Partitions and Subpartitions

You can convert a partition or subpartition into a nonpartitioned table, and a nonpartitioned table into a partition or subpartition of a partitioned table by exchanging their data segments.

You can also convert a hash partitioned table into a partition of a composite *-hash partitioned table, or convert the partition of a composite *-hash partitioned table into a hash partitioned table. Similarly, you can convert a range- or list-partitioned table into a partition of a composite *-range or -list partitioned table, or convert a partition of the composite *-range or -list partitioned table into a range- or list-partitioned table.

Exchanging table partitions is useful to get data quickly in or out of a partitioned table. For example, in data warehousing environments, exchanging partitions facilitates high-speed data loading of new, incremental data into an existing partitioned table.

Note that during the exchange process the data from the source is moved to the target and the data from the target is moved to the source.

OLTP and data warehousing environments benefit from exchanging old data partitions out of a partitioned table. The data is purged from the partitioned table without actually being deleted and can be archived separately afterward.

When you exchange partitions, logging attributes are preserved. You can optionally specify if local indexes are also to be exchanged with the `INCLUDING INDEXES` clause, and if rows are to be validated for proper mapping with the `WITH VALIDATION` clause.

> **Note:**
>
> When you specify `WITHOUT VALIDATION` for the exchange partition operation, this is normally a fast operation because it involves only data dictionary updates. However, if the table or partitioned table involved in the exchange operation has a primary key or unique constraint enabled, then the exchange operation is performed as if `WITH VALIDATION` were specified to maintain the integrity of the constraints.
>
> To avoid the overhead of this validation activity, issue the following statement for each constraint before performing the exchange partition operation:
>
> ```
> ALTER TABLE table_name
>      DISABLE CONSTRAINT constraint_name KEEP INDEX
> ```
>
> Enable the constraints after the exchange.
>
> If you specify `WITHOUT VALIDATION`, then you must ensure that the data to be exchanged belongs in the partition you exchange. You can use the `ORA_PARTITION_VALIDATION` SQL function to help identify those records that have been inserted incorrectly in the wrong partition.

Unless you specify `UPDATE INDEXES`, the Oracle Database marks the global indexes or all global index partitions on the table whose partition is being exchanged as `UNUSABLE`. Global indexes or global index partitions on the table being exchanged remain invalidated.

You cannot use `UPDATE INDEXES` for index-organized tables. Use `UPDATE GLOBAL INDEXES` instead.

Incremental statistics on a partitioned table are maintained with a partition exchange operation if the statistics were gathered on the nonpartitioned table when `DBMS_STATS` table preferences `INCREMENTAL` is set to true and `INCREMENTAL_LEVEL` is set to `TABLE`.

> **Note:**
>
> In situations where column statistics for virtual columns are out of order, the column statistics are deleted rather than retaining the stale statistics. Information about this deletion is written to the alert log file.

This section contains the following topics:

- Creating a Table for Exchange With a Partitioned Table
- Exchanging a Range, Hash, or List Partition
- Exchanging a Partition of an Interval Partitioned Table
- Exchanging a Partition of a Reference-Partitioned Table
- About Exchanging a Partition of a Table with Virtual Columns
- Exchanging a Hash Partitioned Table with a *-Hash Partition
- Exchanging a Subpartition of a *-Hash Partitioned Table
- Exchanging a List-Partitioned Table with a *-List Partition

- About Exchanging a Subpartition of a *-List Partitioned Table
- Exchanging a Range-Partitioned Table with a *-Range Partition
- About Exchanging a Subpartition of a *-Range Partitioned Table
- About Exchanging a Partition with the Cascade Option

> **See Also:**
>
> - Partitioning Key for information about validating partition content
> - Viewing Information About Partitioned Tables and Indexes for information about using views to monitor details about partitioned tables and indexes
> - *Oracle Database SQL Tuning Guide* for more information about incremental statistics
> - *Oracle Database PL/SQL Packages and Types Reference* for information about the `DBMS_STATS` package

4.4.5.1 Creating a Table for Exchange With a Partitioned Table

Tables can be created with the `FOR EXCHANGE WITH` clause to exactly match the shape of a partitioned table and be eligible for a partition exchange command. However, indexes are not created as an operation of this command.

Because the `FOR EXCHANGE WITH` clause of `CREATE TABLE` provides an exact match between a non-partitioned and partitioned table, this is an improvement over the `CREATE TABLE AS SELECT` statement.

The following list is a summary of the effects of the `CREATE TABLE FOR EXCHANGE WITH` DDL operation:

- The use case of this DDL operation is to facilitate creation of a table to be used for exchange partition DDL.
- The operation creates a clone of the for exchange table in terms of column ordering and column properties.
- Columns cannot be renamed. The table being created inherits the names from the for exchange table.
- The only logical property that can be specified during the DDL operation is the partitioning specification of the table.

  The partitioning clause is only relevant for the exchange with a partition of a composite-partitioned table. In this case, a partition with *n* subpartitions is exchanged with a partitioned table with *n* partitions matching the subpartitions. You are responsible for the definition of the partitioning clause for this exchange in this scenario.

  The subpartitioning can be asymmetrical across partitions. The partitioning clause has to match exactly the subpartitioning of the partition to being exchanged.
- The physical properties which can be specified are primarily table segment attributes.
- Column properties copied with this DDL operation include, but are not limited to, the following: unusable columns, invisible columns, virtual expression columns, functional index expression columns, and other internal settings and attributes.

The following is an example of the use of the `CREATE TABLE` statement with the `FOR EXCHANGE WITH` clause to create a table that mimics the shape of an existing table in terms of column ordering and properties.

Example 4-30 Using the FOR EXCHANGE WITH clause of CREATE TABLE

```
CREATE TABLE sales_by_year_table
  ( prod_id       NUMBER        NOT NULL,
    cust_id       NUMBER        NOT NULL,
    time_id       DATE          NOT NULL,
    channel_id    NUMBER        NOT NULL,
    promo_id      NUMBER        NOT NULL,
    quantity_sold NUMBER(10,2)  NOT NULL,
    amount_sold   NUMBER(10,2)  NOT NULL
  )
    PARTITION BY RANGE (time_id)
     (PARTITION sales_2016 VALUES LESS THAN (TO_DATE('01-01-2017','dd-mm-yyyy')),
      PARTITION sales_2017 VALUES LESS THAN (TO_DATE('01-01-2018','dd-mm-yyyy')),
      PARTITION sales_2018 VALUES LESS THAN (TO_DATE('01-01-2019','dd-mm-yyyy')),
      PARTITION sales_2019 VALUES LESS THAN (TO_DATE('01-01-2020','dd-mm-yyyy')),
      PARTITION sales_future VALUES LESS THAN (MAXVALUE)
 );

DESCRIBE sales_by_year_table
 Name                                      Null?    Type
 ----------------------------------------- -------- ----------------------------
 PROD_ID                                   NOT NULL NUMBER
 CUST_ID                                   NOT NULL NUMBER
 TIME_ID                                   NOT NULL DATE
 CHANNEL_ID                                NOT NULL NUMBER
 PROMO_ID                                  NOT NULL NUMBER
 QUANTITY_SOLD                             NOT NULL NUMBER(10,2)
 AMOUNT_SOLD                               NOT NULL NUMBER(10,2)

CREATE TABLE sales_later_year_table
  FOR EXCHANGE WITH TABLE sales_by_year_table;

DESCRIBE sales_later_year_table
 Name                                      Null?    Type
 ----------------------------------------- -------- ----------------------------
 PROD_ID                                   NOT NULL NUMBER
 CUST_ID                                   NOT NULL NUMBER
 TIME_ID                                   NOT NULL DATE
 CHANNEL_ID                                NOT NULL NUMBER
 PROMO_ID                                  NOT NULL NUMBER
 QUANTITY_SOLD                             NOT NULL NUMBER(10,2)
 AMOUNT_SOLD                               NOT NULL NUMBER(10,2)
```

4.4.5.2 Exchanging a Range, Hash, or List Partition

To exchange a partition of a range, hash, or list partitioned table with a nonpartitioned table, or the reverse, use the `ALTER TABLE EXCHANGE PARTITION` statement.

The following is an example of exchanging range partitions with a nonpartitioned table.

Example 4-31 Exchanging a Range Partition

```
CREATE TABLE sales_future_table
  ( prod_id       NUMBER        NOT NULL,
    cust_id       NUMBER        NOT NULL,
    time_id       DATE          NOT NULL,
    channel_id    NUMBER        NOT NULL,
```

```
    promo_id      NUMBER        NOT NULL,
    quantity_sold NUMBER(10,2)  NOT NULL,
    amount_sold   NUMBER(10,2)  NOT NULL
  )
    PARTITION BY RANGE (time_id)
     (PARTITION s_2020 VALUES LESS THAN (TO_DATE('01-01-2021','dd-mm-yyyy')),
      PARTITION s_2021 VALUES LESS THAN (TO_DATE('01-01-2022','dd-mm-yyyy')),
      PARTITION s_2022 VALUES LESS THAN (TO_DATE('01-01-2023','dd-mm-yyyy'))
 );

CREATE TABLE sales_exchange_table
  FOR EXCHANGE WITH TABLE sales_future_table;

INSERT INTO sales_exchange_table VALUES (1002,110,TO_DATE('19-02-2020','dd-mm-yyyy'),12,18,150,4800);
INSERT INTO sales_exchange_table VALUES (1001,100,TO_DATE('12-03-2020','dd-mm-yyyy'),10,15,400,6500);
INSERT INTO sales_exchange_table VALUES (1001,100,TO_DATE('31-05-2020','dd-mm-yyyy'),10,15,600,8000);
INSERT INTO sales_exchange_table VALUES (2105,101,TO_DATE('25-06-2020','dd-mm-yyyy'),12,19,100,3000);
INSERT INTO sales_exchange_table VALUES (1002,120,TO_DATE('31-08-2020','dd-mm-yyyy'),10,15,400,6000);
INSERT INTO sales_exchange_table VALUES (2105,101,TO_DATE('25-10-2020','dd-mm-yyyy'),12,19,250,7500);

ALTER TABLE sales_future_table
    EXCHANGE PARTITION s_2020 WITH TABLE sales_exchange_table;

SELECT * FROM sales_future_table PARTITION(s_2020);
   PROD_ID    CUST_ID TIME_ID   CHANNEL_ID   PROMO_ID QUANTITY_SOLD AMOUNT_SOLD
---------- ---------- --------- ---------- ---------- ------------- -----------
      1002        110 19-FEB-20         12         18           150        4800
      1001        100 12-MAR-20         10         15           400        6500
      1001        100 31-MAY-20         10         15           600        8000
      2105        101 25-JUN-20         12         19           100        3000
      1002        120 31-AUG-20         10         15           400        6000
      2105        101 25-OCT-20         12         19           250        7500
6 rows selected.

REM Note that all records have been removed from the sales_exchange_table
SELECT * FROM sales_exchange_table;
no rows selected

INSERT INTO sales_exchange_table VALUES (1002,110,TO_DATE('15-02-2021','dd-mm-yyyy'),12,18,300,9500);
INSERT INTO sales_exchange_table VALUES (1002,120,TO_DATE('31-03-2021','dd-mm-yyyy'),10,15,200,3000);
INSERT INTO sales_exchange_table VALUES (2105,101,TO_DATE('25-04-2021','dd-mm-yyyy'),12,19,150,9000);

ALTER TABLE sales_future_table
    EXCHANGE PARTITION s_2021 WITH TABLE sales_exchange_table;

SELECT * FROM sales_future_table PARTITION(s_2021);
   PROD_ID    CUST_ID TIME_ID   CHANNEL_ID   PROMO_ID QUANTITY_SOLD AMOUNT_SOLD
---------- ---------- --------- ---------- ---------- ------------- -----------
      1002        110 15-FEB-21         12         18           300        9500
      1002        120 31-MAR-21         10         15           200        3000
      2105        101 25-APR-21         12         19           150        9000
3 rows selected.
```

4.4.5.3 Exchanging a Partition of an Interval Partitioned Table

You can exchange interval partitions in an interval-partitioned table. However, you must ensure that the interval partition has been created before you can exchange the partition.

The following example shows a partition exchange for the `interval_sales` table, interval-partitioned using monthly partitions as of January 1, 2007. This example shows how to add data for June 2007 to the table using partition exchange load. Assume there are only local

indexes on the `interval_sales` table, and equivalent indexes have been created on the `interval_sales_june_2007` table.

```
ALTER TABLE interval_sales
  EXCHANGE PARTITION FOR (TO_DATE('01-JUN-2007','dd-MON-yyyy'))
  WITH TABLE interval_sales_jun_2007
  INCLUDING INDEXES;
```

Note the use of the `FOR` syntax to identify a partition that was system-generated. You can determine the partition name by querying the `*_TAB_PARTITIONS` data dictionary view to display the system-generated partition name.

4.4.5.4 Exchanging a Partition of a Reference-Partitioned Table

You can exchange partitions in a reference-partitioned table, but you must ensure that the data that you reference is available in the respective partition in the parent table.

Example 4-32 shows a partition exchange load scenario for the range-partitioned `orders` table, and the reference partitioned `order_items` table. The data in the `order_items_2018_dec` table only contains order item data for orders with an `order_date` in December 2018.

You must use the `UPDATE GLOBAL INDEXES` or `UPDATE INDEXES` on the exchange partition of the parent table in order for the primary key index to remain usable. Note also that you must create or enable the foreign key constraint on the `order_items_2018_dec` table in order for the partition exchange on the reference-partitioned table to succeed.

For information and an example using exchanging with the `CASCADE` keyword, refer to About Exchanging a Partition with the Cascade Option.

Example 4-32 Exchanging a partition for a reference-partitioned table

```
CREATE TABLE orders (
       order_id number NOT NULL,
       order_date DATE,
       CONSTRAINT order_pk PRIMARY KEY (order_id))
       PARTITION by range (order_date)
       (PARTITION p_2018_dec values less than ('01-JAN-2019'));

CREATE TABLE order_items  (
       order_item_id NUMBER NOT NULL,
       order_id NUMBER not null,
       order_item VARCHAR2(100),
       CONSTRAINT order_item_pk PRIMARY KEY (order_item_id),
       CONSTRAINT order_item_fk FOREIGN KEY (order_id) references orders(order_id) on delete cascade)
       PARTITION by reference (order_item_fk);

CREATE TABLE orders_2018_dec (
       order_id NUMBER,
       order_date DATE,
       CONSTRAINT order_2018_dec_pk PRIMARY KEY (order_id));

INSERT into orders_2018_dec values (1,'01-DEC-2018');
COMMIT;

CREATE TABLE order_items_2018_dec (
       order_item_id NUMBER,
       order_id NUMBER NOT NULL,
       order_item VARCHAR2(100),
       CONSTRAINT order_item_2018_dec_pk PRIMARY KEY (order_item_id),
       CONSTRAINT order_item_2018_dec_fk FOREIGN KEY (order_id) references orders_2018_dec (order_id) on
delete cascade);
```

```
INSERT into order_items_2018_dec values (1,1,'item A');
INSERT into order_items_2018_dec values (2,1,'item B');

REM You must disable or DROP the constraint before the exchange
ALTER TABLE order_items_2018_dec DROP CONSTRAINT order_item_2018_dec_fk;

REM ALTER TABLE is successful with disabled PK-FK
ALTER TABLE orders
  EXCHANGE PARTITION p_2018_dec
  WITH TABLE orders_2018_dec
  UPDATE GLOBAL INDEXES;

REM You must establish the PK-FK with the future parent prior to this exchange
ALTER TABLE order_items_2018_dec
  ADD CONSTRAINT order_items_dec_2018_fk
  FOREIGN KEY (order_id)
  REFERENCES orders(order_id) ;

REM Complete the exchange
ALTER TABLE order_items
  EXCHANGE PARTITION p_2018_dec
  WITH TABLE order_items_2018_dec;

REM Display the data
SELECT * FROM orders;
  ORDER_ID ORDER_DAT
---------- ---------
         1 01-DEC-18

SELECT * FROM order_items;
ORDER_ITEM_ID   ORDER_ID ORDER_ITEM
------------- ---------- ------------
            1          1 item A
            2          1 item B
```

4.4.5.5 About Exchanging a Partition of a Table with Virtual Columns

You can exchange partitions in the presence of virtual columns.

In order for a partition exchange on a partitioned table with virtual columns to succeed, you must create a table that matches the definition of all non-virtual columns in a single partition of the partitioned table. You do not need to include the virtual column definitions, unless constraints or indexes have been defined on the virtual column.

In this case, you must include the virtual column definition to match the partitioned table's constraint and index definitions. This scenario also applies to virtual column-based partitioned tables.

4.4.5.6 Exchanging a Hash Partitioned Table with a *-Hash Partition

You can exchange a whole hash partitioned table, with all of its partitions, with the partition of a *-hash partitioned table and all of its hash subpartitions.

The following example illustrates this concept for a range-hash partitioned table.

First, create a hash partitioned table:

```
CREATE TABLE t1 (i NUMBER, j NUMBER)
     PARTITION BY HASH(i)
       (PARTITION p1, PARTITION p2);
```

Populate the table, then create a range-hash partitioned table as follows:

```
CREATE TABLE t2 (i NUMBER, j NUMBER)
     PARTITION BY RANGE(j)
     SUBPARTITION BY HASH(i)
        (PARTITION p1 VALUES LESS THAN (10)
            (SUBPARTITION t2_pls1,
             SUBPARTITION t2_pls2),
         PARTITION p2 VALUES LESS THAN (20)
            (SUBPARTITION t2_p2s1,
             SUBPARTITION t2_p2s2)
         );
```

It is important that the partitioning key in table `t1` equals the subpartitioning key in table `t2`.

To migrate the data in `t1` to `t2`, and validate the rows, use the following statement:

```
ALTER TABLE t2 EXCHANGE PARTITION p1 WITH TABLE t1
     WITH VALIDATION;
```

4.4.5.7 Exchanging a Subpartition of a *-Hash Partitioned Table

You can use the `ALTER TABLE EXCHANGE SUBPARTITION` statement to convert a hash subpartition of a *-hash partitioned table into a nonpartitioned table, or the reverse.

The following example converts the subpartition `q3_1999_s1` of table `sales` into the nonpartitioned table `q3_1999`. Local index partitions are exchanged with corresponding indexes on `q3_1999`.

```
ALTER TABLE sales EXCHANGE SUBPARTITION q3_1999_s1
      WITH TABLE q3_1999 INCLUDING INDEXES;
```

4.4.5.8 Exchanging a List-Partitioned Table with a *-List Partition

You can use the `ALTER TABLE EXCHANGE PARTITION` statement to exchange a list-partitioned table with a *-list partition.

The semantics are the same as described previously in Exchanging a Hash Partitioned Table with a *-Hash Partition. The following example shows an exchange partition scenario for a list-list partitioned table.

```
CREATE TABLE customers_apac
( id            NUMBER
, name          VARCHAR2(50)
, email         VARCHAR2(100)
, region        VARCHAR2(4)
, credit_rating VARCHAR2(1)
)
PARTITION BY LIST (credit_rating)
( PARTITION poor VALUES ('P')
, PARTITION mediocre VALUES ('C')
, PARTITION good VALUES ('G')
, PARTITION excellent VALUES ('E')
);
```

Populate the table with APAC customers. Then create a list-list partitioned table:

```
CREATE TABLE customers
( id            NUMBER
, name          VARCHAR2(50)
, email         VARCHAR2(100)
```

```
, region        VARCHAR2(4)
, credit_rating VARCHAR2(1)
)
PARTITION BY LIST (region)
SUBPARTITION BY LIST (credit_rating)
SUBPARTITION TEMPLATE
( SUBPARTITION poor VALUES ('P')
, SUBPARTITION mediocre VALUES ('C')
, SUBPARTITION good VALUES ('G')
, SUBPARTITION excellent VALUES ('E')
)
(PARTITION americas VALUES ('AMER')
, PARTITION emea VALUES ('EMEA')
, PARTITION apac VALUES ('APAC')
);
```

It is important that the partitioning key in the `customers_apac` table matches the subpartitioning key in the `customers` table.

Next, exchange the `apac` partition.

```
ALTER TABLE customers
EXCHANGE PARTITION apac
WITH TABLE customers_apac
WITH VALIDATION;
```

4.4.5.9 About Exchanging a Subpartition of a *-List Partitioned Table

You can use the `ALTER TABLE EXCHANGE SUBPARTITION` statement to exchange a subpartition of a *-list partitioned table.

The semantics of the `ALTER TABLEEXCHANGE SUBPARTITION` statement are the same as described previously in Exchanging a Subpartition of a *-Hash Partitioned Table.

4.4.5.10 Exchanging a Range-Partitioned Table with a *-Range Partition

You can use the `ALTER TABLE EXCHANGE PARTITION` statement to exchange a range-partitioned table with a *-range partition.

The semantics of the `ALTER TABLE EXCHANGE PARTITION` statement are the same as described previously in Exchanging a Hash Partitioned Table with a *-Hash Partition. The example below shows the `orders` table, which is interval partitioned by `order_date`, and subpartitioned by range on `order_total`. The example shows how to exchange a single monthly interval with a range-partitioned table.

```
CREATE TABLE orders_mar_2007
 ( id          NUMBER
  , cust_id     NUMBER
  , order_date  DATE
  , order_total NUMBER
 )
PARTITION BY RANGE (order_total)
 ( PARTITION p_small VALUES LESS THAN (1000)
  , PARTITION p_medium VALUES LESS THAN (10000)
  , PARTITION p_large VALUES LESS THAN (100000)
  , PARTITION p_extraordinary VALUES LESS THAN (MAXVALUE)
 );
```

Populate the table with orders for March 2007. Then create an interval-range partitioned table:

```
CREATE TABLE orders
 ( id          NUMBER
  , cust_id     NUMBER
  , order_date  DATE
  , order_total NUMBER
 )
PARTITION BY RANGE (order_date) INTERVAL (NUMTOYMINTERVAL(1,'MONTH'))
  SUBPARTITION BY RANGE (order_total)
  SUBPARTITION TEMPLATE
  ( SUBPARTITION p_small VALUES LESS THAN (1000)
   , SUBPARTITION p_medium VALUES LESS THAN (10000)
   , SUBPARTITION p_large VALUES LESS THAN (100000)
   , SUBPARTITION p_extraordinary VALUES LESS THAN (MAXVALUE)
  )
 (PARTITION p_before_2007 VALUES LESS THAN (TO_DATE('01-JAN-2007','dd-MON-yyyy')));
```

It is important that the partitioning key in the `orders_mar_2007` table matches the subpartitioning key in the `orders` table.

Next, exchange the partition.

```
ALTER TABLE orders
  EXCHANGE PARTITION
   FOR (TO_DATE('01-MAR-2007','dd-MON-yyyy'))
   WITH TABLE orders_mar_2007
   WITH VALIDATION;
```

4.4.5.11 About Exchanging a Subpartition of a *-Range Partitioned Table

You can use the `ALTER TABLE EXCHANGE SUBPARTITION` statement to exchange a subpartition of a *-range partition.

The semantics of the `ALTER TABLE EXCHANGE SUBPARTITION` are the same as described previously in Exchanging a Subpartition of a *-Hash Partitioned Table.

4.4.5.12 About Exchanging a Partition with the Cascade Option

You can cascade exchange operations to reference partitioned child tables with the `CASCADE` option of the `ALTER TABLE EXCHANGE PARTITION` and `ALTER TABLE EXCHANGE SUBPARTITION` SQL statements.

Cascading exchange operations require all foreign key constraints to being defined as `ON DELETE CASCADE`.

When the `CASCADE` option for `ALTER TABLE EXCHANGE PARTITION` and `ALTER TABLE EXCHANGE SUBPARTITION` is specified, the `EXCHANGE` operation cascades to reference partitioned tables that are children of the targeted table. The exchange operation can be targeted at any level in a reference partitioned hierarchy and cascades to child tables starting from the targeted table. Privileges are not required on the child tables, but ordinary restrictions on the exchange operation apply for all tables affected by the operation. The `CASCADE` option is ignored if it is specified for a table that does not have reference partitioned children.

The reference partitioned hierarchy of the targeted table and the reference partitioned hierarchy of the exchange table must match. The `CASCADE` option is not supported if the same parent key is referenced by multiple dependent tables. Having more than one dependent table relying on the same primary key makes it impossible for the kernel to unambiguously identify how to exchange the dependent partitions. Any other options specified for the operation, such as `UPDATE INDEXES`, applies for all tables affected by the operation.

The cascade options are off by default so they do not affect Oracle Database compatibility.

The following example shows the use of `CASCADE` when exchanging the a partition of a referenced-partitioned table.

Example 4-33 Exchanging a partition using cascade for a reference-partitioned table

```
CREATE TABLE orders (
       order_id number NOT NULL,
       order_date DATE,
       CONSTRAINT order_pk PRIMARY KEY (order_id))
       PARTITION by range (order_date)
       (PARTITION p_2018_dec values less than ('01-JAN-2019'));

CREATE TABLE order_items  (
       order_item_id NUMBER NOT NULL,
       order_id NUMBER not null,
       order_item VARCHAR2(100),
       CONSTRAINT order_item_pk PRIMARY KEY (order_item_id),
       CONSTRAINT order_item_fk FOREIGN KEY (order_id) references orders(order_id) on delete cascade)
       PARTITION by reference (order_item_fk);

CREATE TABLE orders_2018_dec (
       order_id NUMBER,
       order_date DATE,
       CONSTRAINT order_2018_dec_pk PRIMARY KEY (order_id));

INSERT into orders_2018_dec values (1,'01-DEC-2018');

CREATE TABLE order_items_2018_dec (
       order_item_id NUMBER,
       order_id NUMBER NOT NULL,
       order_item VARCHAR2(100),
       CONSTRAINT order_item_2018_dec_pk PRIMARY KEY (order_item_id),
       CONSTRAINT order_item_2018_dec_fk FOREIGN KEY (order_id) references orders_2018_dec (order_id) on
delete cascade);

INSERT into order_items_2018_dec values (1,1,'item A new');
INSERT into order_items_2018_dec values (2,1,'item B new');

REM Display data from reference partitioned tables before exchange
SELECT * FROM orders;
no rows selected

SELECT * FROM order_items;
no rows selected

REM ALTER TABLE using cascading exchange
ALTER TABLE orders
   EXCHANGE PARTITION p_2018_dec
   WITH TABLE orders_2018_dec
    CASCADE
   UPDATE GLOBAL INDEXES;

REM Display data from reference partitioned tables after exchange
SELECT * FROM orders;
  ORDER_ID ORDER_DAT
---------- ---------
         1 01-DEC-18

SELECT * FROM order_items;
ORDER_ITEM_ID   ORDER_ID ORDER_ITEM
------------- ---------- ------------
```

```
1          1 item A new
2          1 item B new
```

4.4.6 About Merging Partitions and Subpartitions

Use the `ALTER TABLE MERGE PARTITION` and `SUBPARTITION` SQL statements to merge the contents of two partitions or subpartitions.

The two original partitions or subpartitions are dropped, as are any corresponding local indexes. You cannot use this statement for a hash partitioned table or for hash subpartitions of a composite *-hash partitioned table.

You cannot merge partitions for a reference-partitioned table. Instead, a merge operation on a parent table cascades to all descendant tables. However, you can use the `DEPENDENT TABLES` clause to set specific properties for dependent tables when you issue the merge operation on the parent table to merge partitions or subpartitions.

You can use the `ONLINE` keyword with the `ALTER TABLE MERGE PARTITION` and `SUBPARTITION` SQL statements to enable online merge operations for regular (heap-organized) tables. For an example of the use of the `ONLINE` keyword, see Example 4-34.

If the involved partitions or subpartitions contain data, then indexes may be marked `UNUSABLE` as described in the following table:

Table Type	Index Behavior
Regular (Heap)	Unless you specify `UPDATE INDEXES` as part of the `ALTER TABLE` statement: • The database marks `UNUSABLE` all resulting corresponding local index partitions or subpartitions. • Global indexes, or all partitions of partitioned global indexes, are marked `UNUSABLE` and must be rebuilt.
Index-organized	• The database marks `UNUSABLE` all resulting corresponding local index partitions. • All global indexes remain usable.

This section contains the following topics:

- Merging Range Partitions
- Merging Interval Partitions
- Merging List Partitions
- Merging *-Hash Partitions
- About Merging *-List Partitions
- About Merging *-Range Partitions
- Merging Multiple Partitions

> **See Also:**
>
> Restrictions and Special Requirements for Multi-Partition Operations on Tables with Domain Indexes
>
> *Oracle Database SQL Language Reference*

4.4.6.1 Merging Range Partitions

You can merge the contents of two adjacent range partitions into one partition.

Nonadjacent range partitions cannot be merged. The resulting partition inherits the higher upper bound of the two merged partitions.

One reason for merging range partitions is to keep historical data online in larger partitions. For example, you can have daily partitions, with the oldest partition rolled up into weekly partitions, which can then be rolled up into monthly partitions, and so on.

Example 4-34 shows an example of merging range partitions using the `ONLINE` keyword.

Example 4-34 Merging range partitions

```
-- First, create a partitioned table with four partitions, each on its own
-- tablespace, partitioned by range on the date column
--
CREATE TABLE four_seasons
(       one DATE,
        two VARCHAR2(60),
      three NUMBER
)
PARTITION BY RANGE (one)
(
PARTITION quarter_one
   VALUES LESS THAN ( TO_DATE('01-APR-2017','dd-mon-yyyy'))
   TABLESPACE quarter_one,
PARTITION quarter_two
   VALUES LESS THAN ( TO_DATE('01-JUL-2017','dd-mon-yyyy'))
   TABLESPACE quarter_two,
PARTITION quarter_three
   VALUES LESS THAN ( TO_DATE('01-OCT-2017','dd-mon-yyyy'))
   TABLESPACE quarter_three,
PARTITION quarter_four
   VALUES LESS THAN ( TO_DATE('01-JAN-2018','dd-mon-yyyy'))
   TABLESPACE quarter_four
);
--
-- Create local PREFIXED indexes on four_seasons
-- Prefixed because the leftmost columns of the index match the
-- Partitioning key
--
CREATE INDEX i_four_seasons_l ON four_seasons (one,two)
  LOCAL (
  PARTITION i_quarter_one TABLESPACE i_quarter_one,
  PARTITION i_quarter_two TABLESPACE i_quarter_two,
  PARTITION i_quarter_three TABLESPACE i_quarter_three,
  PARTITION i_quarter_four TABLESPACE i_quarter_four
);


SELECT TABLE_NAME, PARTITION_NAME FROM USER_TAB_PARTITIONS WHERE TABLE_NAME
='FOUR_SEASONS';
TABLE_NAME                       PARTITION_NAME
-------------------------------- -------------------------
FOUR_SEASONS                     QUARTER_FOUR
FOUR_SEASONS                     QUARTER_ONE
FOUR_SEASONS                     QUARTER_THREE
FOUR_SEASONS                     QUARTER_TWO
```

```
-- Next, merge the first two partitions
ALTER TABLE four_seasons
  MERGE PARTITIONS quarter_one, quarter_two INTO PARTITION quarter_two
  UPDATE INDEXES
  ONLINE;

SELECT TABLE_NAME, PARTITION_NAME FROM USER_TAB_PARTITIONS WHERE TABLE_NAME
='FOUR_SEASONS';
TABLE_NAME                       PARTITION_NAME
-------------------------------- ------------------------
FOUR_SEASONS                     QUARTER_FOUR
FOUR_SEASONS                     QUARTER_THREE
FOUR_SEASONS                     QUARTER_TWO
```

If you omit the `UPDATE INDEXES` clause from the `ALTER TABLE` *four_season* statement, then you must rebuild the local index for the affected partition.

```
-- Rebuild the index for quarter_two, which has been marked unusable
-- because it has not had all of the data from quarter_one added to it.
-- Rebuilding the index corrects this condition.
--
ALTER TABLE four_seasons MODIFY PARTITION quarter_two
  REBUILD UNUSABLE LOCAL INDEXES;
```

4.4.6.2 Merging Interval Partitions

The contents of two adjacent interval partitions can be merged into one partition.

Nonadjacent interval partitions cannot be merged. The first interval partition can also be merged with the highest range partition. The resulting partition inherits the higher upper bound of the two merged partitions.

Merging interval partitions always results in the transition point being moved to the higher upper bound of the two merged partitions. This result is that the range section of the interval-partitioned table is extended to the upper bound of the two merged partitions. Any materialized interval partitions with boundaries lower than the newly merged partition are automatically converted into range partitions, with their upper boundaries defined by the upper boundaries of their intervals.

For example, consider the following interval-partitioned table transactions:

```
CREATE TABLE transactions
( id               NUMBER
, transaction_date DATE
, value            NUMBER
)
PARTITION BY RANGE (transaction_date)
INTERVAL (NUMTODSINTERVAL(1,'DAY'))
( PARTITION p_before_2007 VALUES LESS THAN (TO_DATE('01-JAN-2007','dd-MON-yyyy')));
```

Inserting data into the interval section of the table creates the interval partitions for these days. The data for January 15, 2007 and January 16, 2007 are stored in adjacent interval partitions.

```
INSERT INTO transactions VALUES (1,TO_DATE('15-JAN-2007','dd-MON-yyyy'),100);
INSERT INTO transactions VALUES (2,TO_DATE('16-JAN-2007','dd-MON-yyyy'),600);
INSERT INTO transactions VALUES (3,TO_DATE('30-JAN-2007','dd-MON-yyyy'),200);
```

Next, merge the two adjacent interval partitions. The new partition again has a system-generated name.

```
ALTER TABLE transactions
MERGE PARTITIONS FOR(TO_DATE('15-JAN-2007','dd-MON-yyyy'))
, FOR(TO_DATE('16-JAN-2007','dd-MON-yyyy'));
```

The transition point for the `transactions` table has now moved to January 17, 2007. The range section of the interval-partitioned table contains two range partitions: values less than January 1, 2007 and values less than January 17, 2007. Values greater than January 17, 2007 fall in the interval portion of the interval-partitioned table.

4.4.6.3 Merging List Partitions

When you merge list partitions, the partitions being merged can be any two partitions.

They do not need to be adjacent, as for range partitions, because list partitioning does not assume any order for partitions. The resulting partition consists of all of the data from the original two partitions. If you merge a default list partition with any other partition, then the resulting partition is the default partition.

The following statement merges two partitions of a table partitioned using the list method into a partition that inherits all of its attributes from the table-level default attributes. `MAXEXTENTS` is specified in the statement.

```
ALTER TABLE q1_sales_by_region
   MERGE PARTITIONS q1_northcentral, q1_southcentral
   INTO PARTITION q1_central
      STORAGE(MAXEXTENTS 20);
```

The value lists for the two original partitions were specified as:

```
PARTITION q1_northcentral VALUES ('SD','WI')
PARTITION q1_southcentral VALUES ('OK','TX')
```

The resulting `sales_west` partition value list comprises the set that represents the union of these two partition value lists, or specifically:

```
('SD','WI','OK','TX')
```

4.4.6.4 Merging *-Hash Partitions

When you merge *-hash partitions, the subpartitions are rehashed into the number of subpartitions specified by `SUBPARTITIONS` *`n`* or the `SUBPARTITION` clause. If neither is included, table-level defaults are used.

The inheritance of properties is different when a *-hash partition is split, as opposed to when two *-hash partitions are merged. When a partition is split, the new partitions can inherit properties from the original partition because there is only one parent. However, when partitions are merged, properties must be inherited from the table level.

For interval-hash partitioned tables, you can only merge two adjacent interval partitions, or the highest range partition with the first interval partition. The transition point moves when you merge intervals in an interval-hash partitioned table.

The following example merges two range-hash partitions:

```
ALTER TABLE all_seasons
   MERGE PARTITIONS quarter_1, quarter_2 INTO PARTITION quarter_2
   SUBPARTITIONS 8;
```

> **See Also:**
>
> - Splitting a *-Hash Partition for information about splitting a hash partition
> - Merging Interval Partitions for information about merging interval partitions

4.4.6.5 About Merging *-List Partitions

Partitions can be merged at the partition level and subpartitions can be merged at the list subpartition level.

This section contains the following topics.

- Merging Partitions in a *-List Partitioned Table
- Merging Subpartitions in a *-List Partitioned Table

4.4.6.5.1 Merging Partitions in a *-List Partitioned Table

When you merge two *-list partitions, the resulting new partition inherits the subpartition descriptions from the subpartition template, if a template exists. If no subpartition template exists, then a single default subpartition is created for the new partition.

For interval-list partitioned tables, you can only merge two adjacent interval partitions, or the highest range partition with the first interval partition. The transition point moves when you merge intervals in an interval-list partitioned table.

The following statement merges two partitions in the range-list partitioned `stripe_regional_sales` table. A subpartition template exists for the table.

```
ALTER TABLE stripe_regional_sales
   MERGE PARTITIONS q1_1999, q2_1999 INTO PARTITION q1_q2_1999
      STORAGE(MAXEXTENTS 20);
```

Some new physical attributes are specified for this new partition while table-level defaults are inherited for those that are not specified. The new resulting partition `q1_q2_1999` inherits the high-value bound of the partition `q2_1999` and the subpartition value-list descriptions from the subpartition template description of the table.

The data in the resulting partitions consists of data from both the partitions. However, there may be cases where the database returns an error. This can occur because data may map out of the new partition when both of the following conditions exist:

This error condition can be eliminated by always specifying a default partition in the default subpartition template.

- Some literal values of the merged subpartitions were not included in the subpartition template.
- The subpartition template does not contain a default partition definition.

> **See Also:**
>
> - Merging List Partitions for information about merging partitions in a *-list partitioned table
> - Merging Interval Partitions for information about merging interval partitions

4.4.6.5.2 Merging Subpartitions in a *-List Partitioned Table

You can merge the contents of any two arbitrary list subpartitions belonging to the *same* partition.

The resulting subpartition value-list descriptor includes all of the literal values in the value lists for the partitions being merged.

The following statement merges two subpartitions of a table partitioned using range-list method into a new subpartition located in tablespace `ts4`:

```
ALTER TABLE quarterly_regional_sales
   MERGE SUBPARTITIONS q1_1999_northwest, q1_1999_southwest
      INTO SUBPARTITION q1_1999_west
         TABLESPACE ts4;
```

The value lists for the original two partitions were:

- Subpartition `q1_1999_northwest` was described as `('WA','OR')`
- Subpartition `q1_1999_southwest` was described as `('AZ','NM','UT')`

The resulting subpartition value list comprises the set that represents the union of these two subpartition value lists:

- Subpartition `q1_1999_west` has a value list described as `('WA','OR','AZ','NM','UT')`

The tablespace in which the resulting subpartition is located and the subpartition attributes are determined by the partition-level default attributes, except for those specified explicitly. If any of the existing subpartition names are being reused, then the new subpartition inherits the subpartition attributes of the subpartition whose name is being reused.

4.4.6.6 About Merging *-Range Partitions

Partitions can be merged at the partition level and subpartitions can be merged at the range subpartition level.

- Merging Partitions in a *-Range Partitioned Table

4.4.6.6.1 Merging Partitions in a *-Range Partitioned Table

When you merge two *-range partitions, the resulting new partition inherits the subpartition descriptions from the subpartition template, if one exists. If no subpartition template exists, then a single subpartition with an upper boundary `MAXVALUE` is created for the new partition.

For interval-range partitioned tables, you can only merge two adjacent interval partitions, or the highest range partition with the first interval partition. The transition point moves when you merge intervals in an interval-range partitioned table.

The following statement merges two partitions in the monthly interval-range partitioned `orders` table. A subpartition template exists for the table.

```
ALTER TABLE orders
MERGE PARTITIONS FOR(TO_DATE('01-MAR-2007','dd-MON-yyyy')),
FOR(TO_DATE('01-APR-2007','dd-MON-yyyy'))
INTO PARTITION p_pre_may_2007;
```

If the March 2007 and April 2007 partitions were still in the interval section of the interval-range partitioned table, then the merge operation would move the transition point to May 1, 2007.

The subpartitions for partition `p_pre_may_2007` inherit their properties from the subpartition template. The data in the resulting partitions consists of data from both the partitions. However, there may be cases where the database returns an error. This can occur because data may map out of the new partition when both of the following conditions are met:

The error condition can be eliminated by always specifying a subpartition with an upper boundary of `MAXVALUE` in the subpartition template.

- Some range values of the merged subpartitions were not included in the subpartition template.
- The subpartition template does not have a subpartition definition with a `MAXVALUE` upper boundary.

> **See Also:**
>
> - Merging Range Partitions for information about merging partitions in a *-range partitioned table
> - Merging Interval Partitions for information about merging interval partitions

4.4.6.7 Merging Multiple Partitions

You can merge the contents of two or more partitions or subpartitions into one new partition or subpartition and then drop the original partitions or subpartitions with the `MERGE PARTITIONS` and `MERGE SUBPARTITIONS` clauses of the `ALTER TABLE` SQL statement.

The `MERGE PARTITIONS` and `MERGE SUBPARTITIONS` clauses are synonymous with the `MERGE PARTITION` and `MERGE SUBPARTITION` clauses.

For example, the following SQL statement merges four partitions into one partition and drops the four partitions that were merged.

```
ALTER TABLE t1 MERGE PARTITIONS p01, p02, p03, p04 INTO p0;
```

When merging multiple range partitions, the partitions must be adjacent and specified in the ascending order of their partition bound values. The new partition inherits the partition upper bound of the highest of the original partitions.

You can specify the lowest and the highest partitions to be merged when merging multiple range partitions with the `TO` syntax. All partitions between specified partitions, including those specified, are merged into the target partition. You cannot use this syntax for list and system partitions.

For example, the following SQL statements merges partitions `p01` through `p04` into the partition `p0`.

```
ALTER TABLE t1 MERGE PARTITIONS p01 TO p04 INTO p0;
```

List partitions and system partitions that you want to merge do not need to be adjacent, because no ordering of the partitions is assumed. When merging multiple list partitions, the resulting partition value list are the union of the set of partition value list of all of the partitions to be merged. A `DEFAULT` list partition merged with other list partitions results in a `DEFAULT` partition.

When merging multiple partitions of a composite partitioned table, the resulting new partition inherits the subpartition descriptions from the subpartition template, if one exists. If no subpartition template exists, then Oracle creates one `MAXVALUE` subpartition from range subpartitions or one `DEFAULT` subpartition from list subpartitions for the new partition. When merging multiple subpartitions of a composite partitioned table, the subpartitions to be merged must belong to the same partition.

When merging multiple partitions, local and global index operations and semantics for inheritance of unspecified physical attributes are the same for merging two partitions.

In the following SQL statement, four partitions of the partitioned by range table `sales` are merged. These four partitions that correspond to the four quarters of the oldest year are merged into a single partition containing the entire sales data of the year.

```
ALTER TABLE sales
  MERGE PARTITIONS sales_q1_2009, sales_q2_2009, sales_q3_2009, sales_q4_2009
  INTO PARTITION sales_2009;
```

The previous SQL statement can be rewritten as the following SQL statement to obtain the same result.

```
ALTER TABLE sales
  MERGE PARTITIONS sales_q1_2009 TO sales_q4_2009
  INTO PARTITION sales_2009;
```

> **Note:**
>
> See Restrictions and Special Requirements for Multi-Partition Operations on Tables with Domain Indexes

4.4.7 About Modifying Attributes of Tables, Partitions, and Subpartitions

The modification of attributes of tables, partitions, and subpartitions is introduced in this topic.

- About Modifying Default Attributes
- About Modifying Real Attributes of Partitions

4.4.7.1 About Modifying Default Attributes

You can modify the default attributes of a table, or for a partition of a composite partitioned table.

When you modify default attributes, the new attributes affect only future partitions, or subpartitions, that are created. The default values can still be specifically overridden when creating a new partition or subpartition. You can modify the default attributes of a reference-partitioned table.

This section contains the following topics:

- Modifying Default Attributes of a Table

- Modifying Default Attributes of a Partition
- Modifying Default Attributes of Index Partitions

4.4.7.1.1 Modifying Default Attributes of a Table

You can modify the default attributes that are inherited for range, hash, list, interval, or reference partitions using the `MODIFY DEFAULT ATTRIBUTES` clause of `ALTER TABLE`.

For hash partitioned tables, only the `TABLESPACE` attribute can be modified.

4.4.7.1.2 Modifying Default Attributes of a Partition

To modify the default attributes inherited when creating subpartitions, use the `ALTER TABLE MODIFY DEFAULT ATTRIBUTES FOR PARTITION`.

The following statement modifies the `TABLESPACE` in which future subpartitions of partition `p1` in the range-hash partitioned table reside.

```
ALTER TABLE employees_subpartitions
     MODIFY DEFAULT ATTRIBUTES FOR PARTITION p1 TABLESPACE ts1;
```

Because all subpartitions of a range-hash partitioned table must share the same attributes, except `TABLESPACE`, it is the only attribute that can be changed.

You cannot modify default attributes of interval partitions that have not yet been created. To change the way in which future subpartitions in an interval-partitioned table are created, you must modify the subpartition template.

4.4.7.1.3 Modifying Default Attributes of Index Partitions

In a similar fashion to table partitions, you can alter the default attributes that are inherited by partitions of a range-partitioned global index, or local index partitions of partitioned tables.

For this you use the `ALTER INDEX MODIFY DEFAULT ATTRIBUTES` statement. Use the `ALTER INDEX MODIFY DEFAULT ATTRIBUTES FOR PARTITION` statement if you are altering default attributes to be inherited by subpartitions of a composite partitioned table.

4.4.7.2 About Modifying Real Attributes of Partitions

It is possible to modify attributes of an existing partition of a table or index.

You cannot change the `TABLESPACE` attribute. Use `ALTER TABLE MOVE PARTITION/SUBPARTITION` to move a partition or subpartition to a new tablespace.

This section contains the following topics:

- Modifying Real Attributes for a Range or List Partition
- Modifying Real Attributes for a Hash Partition
- Modifying Real Attributes of a Subpartition
- Modifying Real Attributes of Index Partitions

4.4.7.2.1 Modifying Real Attributes for a Range or List Partition

Use the `ALTER TABLE MODIFY PARTITION` statement to modify existing attributes of a range partition or list partition.

You can modify segment attributes (except TABLESPACE), or you can allocate and deallocate extents, mark local index partitions UNUSABLE, or rebuild local indexes that have been marked UNUSABLE.

If this is a range partition of a *-hash partitioned table, then note the following:

- If you allocate or deallocate an extent, this action is performed for every subpartition of the specified partition.
- Likewise, changing any other attributes results in corresponding changes to those attributes of all the subpartitions for that partition. The partition level default attributes are changed as well. To avoid changing attributes of existing subpartitions, use the FOR PARTITION clause of the MODIFY DEFAULT ATTRIBUTES statement.

The following are some examples of modifying the real attributes of a partition.

This example modifies the MAXEXTENTS storage attribute for the range partition sales_q1 of table sales:

```
ALTER TABLE sales MODIFY PARTITION sales_q1
     STORAGE (MAXEXTENTS 10);
```

All of the local index subpartitions of partition ts1 in range-hash partitioned table scubagear are marked UNUSABLE in the following example:

```
ALTER TABLE scubagear MODIFY PARTITION ts1 UNUSABLE LOCAL INDEXES;
```

For an interval-partitioned table you can only modify real attributes of range partitions or interval partitions that have been created.

4.4.7.2.2 Modifying Real Attributes for a Hash Partition

You can use the ALTER TABLE MODIFY PARTITION statement to modify attributes of a hash partition.

However, because the physical attributes of individual hash partitions must all be the same (except for TABLESPACE), you are restricted to:

- Allocating a new extent
- Deallocating an unused extent
- Marking a local index subpartition UNUSABLE
- Rebuilding local index subpartitions that are marked UNUSABLE

The following example rebuilds any unusable local index partitions associated with hash partition p1 of the table:

```
ALTER TABLE departments_rebuild_index MODIFY PARTITION p1
     REBUILD UNUSABLE LOCAL INDEXES;
```

4.4.7.2.3 Modifying Real Attributes of a Subpartition

With the MODIFY SUBPARTITION clause of ALTER TABLE you can perform the same actions as listed previously for partitions, but at the specific composite partitioned table subpartition level.

For example:

```
ALTER TABLE employees_rebuild_index MODIFY SUBPARTITION p3_s1
     REBUILD UNUSABLE LOCAL INDEXES;
```

4.4.7.2.4 Modifying Real Attributes of Index Partitions

The `MODIFY PARTITION` clause of `ALTER INDEX` enables you to modify the real attributes of an index partition or its subpartitions.

The rules are very similar to those for table partitions, but unlike the `MODIFY PARTITION` clause for `ALTER INDEX`, there is no subclause to rebuild an unusable index partition, but there is a subclause to coalesce an index partition or its subpartitions. In this context, coalesce means to merge index blocks where possible to free them for reuse.

You can also allocate or deallocate storage for a subpartition of a local index, or mark it `UNUSABLE`, using the `MODIFY PARTITION` clause.

4.4.8 About Modifying List Partitions

The modification of values in list partitions and subpartitions is introduced in this topic.

- About Modifying List Partitions: Adding Values
- About Modifying List Partitions: Dropping Values

4.4.8.1 About Modifying List Partitions: Adding Values

List partitioning enables you to optionally add literal values from the defining value list.

This section contains the following topics:

- Adding Values for a List Partition
- Adding Values for a List Subpartition

4.4.8.1.1 Adding Values for a List Partition

Use the `MODIFY PARTITION ADD VALUES` clause of the `ALTER TABLE` statement to extend the value list of an existing partition.

Literal values being added must not have been included in any other partition value list. The partition value list for any corresponding local index partition is correspondingly extended, and any global indexes, or global or local index partitions, remain usable.

The following statement adds a new set of state codes ('`OK`', '`KS`') to an existing partition list.

```
ALTER TABLE sales_by_region
   MODIFY PARTITION region_south
      ADD VALUES ('OK', 'KS');
```

The existence of a default partition can have a performance impact when adding values to other partitions. This is because to add values to a list partition, the database must check that the values being added do not exist in the default partition. If any of the values do exist in the default partition, then an error is displayed.

> **Note:**
>
> The database runs a query to check for the existence of rows in the default partition that correspond to the literal values being added. Therefore, it is advisable to create a local prefixed index on the table. This speeds up the execution of the query and the overall operation.

You cannot add values to a default list partition.

4.4.8.1.2 Adding Values for a List Subpartition

Use the `MODIFY SUBPARTITION ADD VALUES` clause of the `ALTER TABLE` statement to extend the value list of an existing subpartition.

This operation is essentially the same as described for About Modifying List Partitions: Adding Values, however, you use a `MODIFY SUBPARTITION` clause instead of the `MODIFY PARTITION` clause. For example, to extend the range of literal values in the value list for subpartition `q1_1999_southeast`, use the following statement:

```
ALTER TABLE quarterly_regional_sales
   MODIFY SUBPARTITION q1_1999_southeast
      ADD VALUES ('KS');
```

Literal values being added must not have been included in any other subpartition value list within the owning partition. However, they can be duplicates of literal values in the subpartition value lists of other partitions within the table.

For an interval-list composite partitioned table, you can only add values to subpartitions of range partitions or interval partitions that have been created. To add values to subpartitions of interval partitions that have not yet been created, you must modify the subpartition template.

4.4.8.2 About Modifying List Partitions: Dropping Values

List partitioning enables you to optionally drop literal values from the defining value list.

This section contains the following topics:

- Dropping Values from a List Partition
- Dropping Values from a List Subpartition

4.4.8.2.1 Dropping Values from a List Partition

Use the `MODIFY PARTITION DROP VALUES` clause of the `ALTER TABLE` statement to remove literal values from the value list of an existing partition.

The statement is always executed with validation, meaning that it checks to see if any rows exist in the partition that corresponds to the set of values being dropped. If any such rows are found then the database returns an error message and the operation fails. When necessary, use a `DELETE` statement to delete corresponding rows from the table before attempting to drop values.

> **Note:**
>
> You cannot drop all literal values from the value list describing the partition. You must use the `ALTER TABLE DROP PARTITION` statement instead.

The partition value list for any corresponding local index partition reflects the new value list, and any global index, or global or local index partitions, remain usable.

The following statement drops a set of state codes ('`OK`' and '`KS`') from an existing partition value list.

```
ALTER TABLE sales_by_region
   MODIFY PARTITION region_south
      DROP VALUES ('OK', 'KS');
```

> **Note:**
>
> The database runs a query to check for the existence of rows in the partition that correspond to the literal values being dropped. Therefore, it is advisable to create a local prefixed index on the table. This speeds up the query and the overall operation.

You cannot drop values from a default list partition.

4.4.8.2.2 Dropping Values from a List Subpartition

Use the `MODIFY SUBPARTITION DROP VALUES` clause of the `ALTER TABLE` statement to remove literal values from the value list of an existing subpartition.

This operation is essentially the same as described for About Modifying List Partitions: Dropping Values, however, you use a `MODIFY SUBPARTITION` clause instead of the `MODIFY PARTITION` clause. For example, to remove a set of literal values in the value list for subpartition `q1_1999_southeast`, use the following statement:

```
ALTER TABLE quarterly_regional_sales
   MODIFY SUBPARTITION q1_1999_southeast
      DROP VALUES ('KS');
```

For an interval-list composite partitioned table, you can only drop values from subpartitions of range partitions or interval partitions that have been created. To drop values from subpartitions of interval partitions that have not yet been created, you must modify the subpartition template.

4.4.9 About Modifying the Partitioning Strategy

You can change the partitioning strategy of a regular (heap-organized) table with the `ALTER TABLE MODIFY PARTITION` SQL statement.

Modifying the partitioning strategy, such as hash partitioning to composite range-hash partitioning, can be performed offline or online. When performed in online mode, the conversion does not impact ongoing DML operations. When performed in offline mode, the conversion does not allow concurrent DML operations during the modification.

Indexes are maintained as part of the table modification. When modifying the partitioning strategy, all unspecified indexes whose index columns are a prefix of the new partitioning key

are automatically converted to a local partitioned index; otherwise, an index is converted to global index.

The modification operation is not supported with domain indexes. The `UPDATE INDEXES` clause cannot change the columns on which the list of indexes was originally defined or the uniqueness property of the index or any other index property.

For information about converting a non-partitioned table to a partitioned table, refer to Converting a Non-Partitioned Table to a Partitioned Table.

Example 4-35 shows the use of `ALTER TABLE` to convert a range partitioned table to a composite range-hash partitioned table online. During the `ALTER TABLE` modification in the example, indexes are updated.

> **Live SQL:**
>
> View and run a related example on Oracle Live SQL at *Modifying the Partitioning Strategy of a Table*.

Example 4-35 Modifying the partitioning strategy

```
CREATE TABLE mod_sales_partitioning
  ( prod_id       NUMBER        NOT NULL,
    cust_id       NUMBER        NOT NULL,
    time_id       DATE          NOT NULL,
    channel_id    NUMBER        NOT NULL,
    promo_id      NUMBER        NOT NULL,
    quantity_sold NUMBER(10,2)  NOT NULL,
    amount_sold   NUMBER(10,2)  NOT NULL
  )
  PARTITION BY RANGE (time_id)
  (PARTITION sales_q1_2017 VALUES LESS THAN (TO_DATE('01-APR-2017','dd-MON-yyyy')),
   PARTITION sales_q2_2017 VALUES LESS THAN (TO_DATE('01-JUL-2017','dd-MON-yyyy')),
   PARTITION sales_q3_2017 VALUES LESS THAN (TO_DATE('01-OCT-2017','dd-MON-yyyy')),
   PARTITION sales_q4_2017 VALUES LESS THAN (TO_DATE('01-JAN-2018','dd-MON-yyyy'))
 );

CREATE INDEX i1_cust_id_indx ON mod_sales_partitioning (cust_id) LOCAL;
CREATE INDEX i2_time_id_indx ON mod_sales_partitioning (time_id);
CREATE INDEX i3_prod_id_indx ON mod_sales_partitioning (prod_id);

SELECT TABLE_NAME, PARTITIONING_TYPE FROM USER_PART_TABLES WHERE TABLE_NAME ='MOD_SALES_PARTITIONING';
TABLE_NAME                PARTITION_NAME
------------------------- --------------
MOD_SALES_PARTITIONING    RANGE

SELECT TABLE_NAME, PARTITION_NAME FROM USER_TAB_PARTITIONS WHERE TABLE_NAME ='MOD_SALES_PARTITIONING';
TABLE_NAME                PARTITION_NAME
------------------------- --------------
MOD_SALES_PARTITIONING    SALES_Q1_2017
MOD_SALES_PARTITIONING    SALES_Q2_2017
MOD_SALES_PARTITIONING    SALES_Q3_2017
MOD_SALES_PARTITIONING    SALES_Q4_2017
...

ALTER TABLE mod_sales_partitioning
 MODIFY
 PARTITION BY RANGE (time_id) SUBPARTITION BY HASH (cust_id)
```

```
  SUBPARTITIONS 8
 ( PARTITION  sales_q1_2017 VALUES LESS THAN (TO_DATE('01-APR-2017','dd-MON-yyyy')),
    PARTITION sales_q2_2017 VALUES LESS THAN (TO_DATE('01-JUL-2017','dd-MON-yyyy')),
    PARTITION sales_q3_2017 VALUES LESS THAN (TO_DATE('01-OCT-2017','dd-MON-yyyy')),
    PARTITION sales_q4_2017 VALUES LESS THAN (TO_DATE('01-JAN-2018','dd-MON-yyyy')))
    ONLINE
  UPDATE INDEXES
   ( i1_cust_id_indx LOCAL,
     i2_time_id_indx GLOBAL PARTITION BY RANGE (time_id)
     (PARTITION ip1_indx VALUES LESS THAN (MAXVALUE) ) );

SELECT TABLE_NAME, PARTITIONING_TYPE, SUBPARTITIONING_TYPE FROM USER_PART_TABLES WHERE TABLE_NAME
='MOD_SALES_PARTITIONING';
TABLE_NAME                     PARTITION      SUBPARTIT
------------------------------ -------------- ----------
MOD_SALES_PARTITIONING         RANGE          HASH

SELECT TABLE_NAME, PARTITION_NAME, SUBPARTITION_NAME FROM USER_TAB_SUBPARTITIONS WHERE TABLE_NAME
='MOD_SALES_PARTITIONING';
TABLE_NAME                     PARTITION_NAME     SUBPARTITION_NAME
------------------------------ ------------------ ------------------
MOD_SALES_PARTITIONING         SALES_Q1_2017      SYS_SUBP567
MOD_SALES_PARTITIONING         SALES_Q1_2017      SYS_SUBP568
MOD_SALES_PARTITIONING         SALES_Q1_2017      SYS_SUBP569
MOD_SALES_PARTITIONING         SALES_Q1_2017      SYS_SUBP570
...
```

4.4.10 About Moving Partitions and Subpartitions

Use the `MOVE PARTITION` clause of the `ALTER TABLE` statement to change the physical storage attributes of a partition.

With the `MOVE PARTITION` clause of the `ALTER TABLE` statement, you can:

- Re-cluster data and reduce fragmentation
- Move a partition to another tablespace
- Modify create-time attributes
- Store the data in compressed format using table compression

Typically, you can change the physical storage attributes of a partition in a single step using an `ALTER TABLE/INDEX MODIFY PARTITION` statement. However, there are some physical attributes, such as `TABLESPACE`, that you cannot modify using `MODIFY PARTITION`. In these cases, use the `MOVE PARTITION` clause. Modifying some other attributes, such as table compression, affects only future storage, but not existing data.

If the partition being moved contains any data, then indexes may be marked `UNUSABLE` according to the following table:

Table Type	Index Behavior
Regular (Heap)	Unless you specify `UPDATE INDEXES` as part of the `ALTER TABLE` statement: • The matching partition in each local index is marked `UNUSABLE`. You must rebuild these index partitions after issuing `MOVE PARTITION`. • Any global indexes, or all partitions of partitioned global indexes, are marked `UNUSABLE`.

Table Type	Index Behavior
Index-organized	Any local or global indexes defined for the partition being moved remain usable because they are primary-key based logical rowids. However, the guess information for these rowids becomes incorrect.

This section contains the following topics:

- Moving Table Partitions
- Moving Subpartitions
- Moving Index Partitions

> **see Also:**
>
> - *Oracle Database SQL Language Reference* for information the `ALTER TABLE MOVE` statement
> - *Oracle Database Administrator's Guide* for information moving tables and partitions

4.4.10.1 Moving Table Partitions

Use the `MOVE PARTITION` clause to move a partition.

For example, to move the most active partition to a tablespace that resides on its own set of disks (to balance I/O), not log the action, and compress the data, issue the following statement:

```
ALTER TABLE parts MOVE PARTITION depot2
     TABLESPACE ts094 NOLOGGING COMPRESS;
```

This statement always drops the old partition segment and creates a new segment, even if you do not specify a new tablespace.

If you are moving a partition of a partitioned index-organized table, then you can specify the `MAPPING TABLE` clause as part of the `MOVE PARTITION` clause, and the mapping table partition are moved to the new location along with the table partition.

For an interval or interval-* partitioned table, you can only move range partitions or interval partitions that have been materialized. A partition move operation does not move the transition point in an interval or interval-* partitioned table.

You can move a partition in a reference-partitioned table independent of the partition in the parent table.

4.4.10.2 Moving Subpartitions

Use the `MOVE SUBPARTITION` clause to move a subpartition.

The following statement shows how to move data in a subpartition of a table. In this example, a `PARALLEL` clause has also been specified.

```
ALTER TABLE scuba_gear MOVE SUBPARTITION bcd_types
     TABLESPACE tbs23 PARALLEL (DEGREE 2);
```

You can move a subpartition in a reference-partitioned table independent of the subpartition in the parent table.

Note that `SPLIT`, `MERGE`, `MOVE`, `TRUNCATE`, and `DROP` operations on partitions and subpartitions will result in `ORPHANED_ENTRIES AS YES`. This is expected behavior.

4.4.10.3 Moving Index Partitions

The `ALTER TABLE MOVE PARTITION` statement for regular tables marks all partitions of a global index `UNUSABLE`.

You can rebuild the entire index by rebuilding each partition individually using the `ALTER INDEX REBUILD PARTITION` statement. You can perform these rebuilds concurrently.

You can also simply drop the index and re-create it.

4.4.11 About Rebuilding Index Partitions

Rebuilding an index provides several advantages.

Some reasons for rebuilding index partitions include:

- To recover space and improve performance
- To repair a damaged index partition caused by media failure
- To rebuild a local index partition after loading the underlying table partition with SQL*Loader or an import utility
- To rebuild index partitions that have been marked `UNUSABLE`
- To enable key compression for B-tree indexes

The following sections discuss options for rebuilding index partitions and subpartitions.

This section contains the following topics:

- About Rebuilding Global Index Partitions
- About Rebuilding Local Index Partitions

4.4.11.1 About Rebuilding Global Index Partitions

You can rebuild global index partitions with several methods.

- Rebuild each partition by issuing the `ALTER INDEX REBUILD PARTITION` statement (you can run the rebuilds concurrently).
- Drop the entire global index and re-create it. This method is more efficient because the table is scanned only one time.

For most maintenance operations on partitioned tables with indexes, you can optionally avoid the need to rebuild the index by specifying `UPDATE INDEXES` on your DDL statement.

4.4.11.2 About Rebuilding Local Index Partitions

You can rebuild local index partitions with several methods.

Rebuild local indexes using either `ALTER INDEX` or `ALTER TABLE` as follows:

- `ALTER INDEX REBUILD PARTITION/SUBPARTITION`

This statement rebuilds an index partition or subpartition unconditionally.

- `ALTER TABLE MODIFY PARTITION/SUBPARTITION REBUILD UNUSABLE LOCAL INDEXES`

  This statement finds all of the unusable indexes for the given table partition or subpartition and rebuilds them. It only rebuilds an index partition if it has been marked `UNUSABLE`.

The following sections contain examples about rebuilding indexes.

- Using ALTER INDEX to Rebuild a Partition
- Using ALTER TABLE to Rebuild an Index Partition

4.4.11.2.1 Using ALTER INDEX to Rebuild a Partition

The `ALTER INDEX REBUILD PARTITION` statement rebuilds one partition of an index.

It cannot be used for composite-partitioned tables. Only real physical segments can be rebuilt with this command. When you re-create the index, you can also choose to move the partition to a new tablespace or change attributes.

For composite-partitioned tables, use `ALTER INDEX REBUILD SUBPARTITION` to rebuild a subpartition of an index. You can move the subpartition to another tablespace or specify a parallel clause. The following statement rebuilds a subpartition of a local index on a table and moves the index subpartition to another tablespace.

```
ALTER INDEX scuba
   REBUILD SUBPARTITION bcd_types
   TABLESPACE tbs23 PARALLEL (DEGREE 2);
```

4.4.11.2.2 Using ALTER TABLE to Rebuild an Index Partition

The `REBUILD UNUSABLE LOCAL INDEXES` clause of `ALTER TABLE MODIFY PARTITION` enables you to rebuild an unusable index partition.

However, the statement does not allow you to specify any new attributes for the rebuilt index partition. The following example finds and rebuilds any unusable local index partitions for table `scubagear`, partition `p1`.

```
ALTER TABLE scubagear
   MODIFY PARTITION p1 REBUILD UNUSABLE LOCAL INDEXES;
```

The `ALTER TABLE MODIFY SUBPARTITION` is the clause for rebuilding unusable local index subpartitions.

4.4.12 About Renaming Partitions and Subpartitions

You can rename partitions and subpartitions of both tables and indexes.

One reason for renaming a partition might be to assign a meaningful name, as opposed to a default system name that was assigned to the partition in another maintenance operation.

All partitioning methods support the `FOR(value)` method to identify a partition. You can use this method to rename a system-generated partition name into a more meaningful name. This is particularly useful in interval or interval-* partitioned tables.

You can independently rename partitions and subpartitions for reference-partitioned parent and child tables. The rename operation on the parent table is not cascaded to descendant tables.

This section contains the following topics:

- Renaming a Table Partition
- Renaming a Table Subpartition
- About Renaming Index Partitions

4.4.12.1 Renaming a Table Partition

You can rename a range, hash, or list partition, using the `ALTER TABLE RENAME PARTITION` statement.

For example:

```
ALTER TABLE scubagear RENAME PARTITION sys_p636 TO tanks;
```

4.4.12.2 Renaming a Table Subpartition

You can assign new names to subpartitions of a table.

In this case, you would use the `ALTER TABLE RENAME SUBPARTITION` syntax.

4.4.12.3 About Renaming Index Partitions

You can rename index partitions and subpartitions with the `ALTER INDEX` statement.

- Renaming an Index Partition
- Renaming an Index Subpartition

4.4.12.3.1 Renaming an Index Partition

Use the `ALTER INDEX RENAME PARTITION` statement to rename an index partition.

The `ALTER INDEX` statement does not support the use of `FOR(`*`value`*`)` to identify a partition. You must use the original partition name in the rename operation.

4.4.12.3.2 Renaming an Index Subpartition

Use the `ALTER INDEX RENAME SUBPARTITION` statement to rename an index subpartition.

The following statement simply shows how to rename a subpartition that has a system generated name that was a consequence of adding a partition to an underlying table:

```
ALTER INDEX scuba RENAME SUBPARTITION sys_subp3254 TO bcd_types;
```

4.4.13 About Splitting Partitions and Subpartitions

You can split the contents of a partition into two new partitions.

The `SPLIT PARTITION` clause of the `ALTER TABLE` or `ALTER INDEX` statement is used to redistribute the contents of a partition into two new partitions. Consider doing this when a partition becomes too large and causes backup, recovery, or maintenance operations to take a long time to complete or it is felt that there is simply too much data in the partition. You can also use the `SPLIT PARTITION` clause to redistribute the I/O load. This clause cannot be used for hash partitions or subpartitions.

If the partition you are splitting contains any data, then indexes may be marked `UNUSABLE` as explained in the following table:

Table Type	Index Behavior
Regular (Heap)	Unless you specify `UPDATE INDEXES` as part of the `ALTER TABLE` statement: • The database marks `UNUSABLE` the new partitions (there are two) in each local index. • Any global indexes, or all partitions of partitioned global indexes, are marked `UNUSABLE` and must be rebuilt.
Index-organized	• The database marks `UNUSABLE` the new partitions (there are two) in each local index. • All global indexes remain usable.

You cannot split partitions or subpartitions in a reference-partitioned table except for the parent table. When you split partitions or subpartitions in the parent table then the split is cascaded to all descendant tables. However, you can use the `DEPENDENT TABLES` clause to set specific properties for dependent tables when you issue the `SPLIT` statement on the parent table to split partitions or subpartitions.

Partition maintenance with `SPLIT` operations are supported as online operations with the keyword `ONLINE` for heap organized tables, enabling concurrent DML operations while a partition maintenance operation is ongoing.

For `ONLINE` operations, split indexes are always updated by default, regardless whether you specify the `UPDATE INDEXES` clause.

For an example of the use of the keyword `ONLINE` with a `SPLIT` operation, see Example 4-37.

This section contains the following topics:

- Splitting a Partition of a Range-Partitioned Table
- Splitting a Partition of a List-Partitioned Table
- Splitting a Partition of an Interval-Partitioned Table
- Splitting a *-Hash Partition
- Splitting Partitions in a *-List Partitioned Table
- Splitting a *-Range Partition
- Splitting Index Partitions
- Splitting into Multiple Partitions
- Fast SPLIT PARTITION and SPLIT SUBPARTITION Operations

> **See Also:**
>
> *Oracle Database SQL Language Reference*

4.4.13.1 Splitting a Partition of a Range-Partitioned Table

You can split a range partition using the `ALTER TABLE SPLIT PARTITION` statement.

In the SQL statement, you must specify values of the partitioning key column within the range of the partition at which to split the partition.

You can optionally specify new attributes for the partitions resulting from the split. If there are local indexes defined on the table, this statement also splits the matching partition in each local index.

If you do not specify new partition names, then the database assigns names of the form `SYS_Pn`. You can examine the data dictionary to locate the names assigned to the new local index partitions. You may want to rename them. Any attributes that you do not specify are inherited from the original partition.

Example 4-36 Splitting a partition of a range-partitioned table and rebuilding indexes

In this example `fee_katy` is a partition in the table `vet_cats`, which has a local index, `jaf1`. There is also a global index, `vet` on the table. `vet` contains two partitions, `vet_parta`, and `vet_partb`. The first of the resulting two new partitions includes all rows in the original partition whose partitioning key column values map lower than the specified value. The second partition contains all rows whose partitioning key column values map greater than or equal to the specified value. The following SQL statement split the partition `fee_katy`, and rebuild the index partitions.

```
ALTER TABLE vet_cats SPLIT PARTITION
      fee_katy at (100) INTO ( PARTITION
      fee_katy1, PARTITION fee_katy2);
ALTER INDEX JAF1 REBUILD PARTITION fee_katy1;
ALTER INDEX JAF1 REBUILD PARTITION fee_katy2;
ALTER INDEX VET REBUILD PARTITION vet_parta;
ALTER INDEX VET REBUILD PARTITION vet_partb;
```

Example 4-37 Splitting a partition of a range-partitioned table online

In this example, the `sales_q4_2016` partition of the`ORDERS` table is split into separate partitions for each month. The `ONLINE` keyword is specified to enable concurrent DML operations while a partition maintenance operation is ongoing.

If there were any indexes on the `ORDERS` table, then those would be maintained automatically as part of the online split.

```
CREATE TABLE orders
 (prod_id       NUMBER(6),
  cust_id       NUMBER,
  time_id       DATE,
  channel_id    CHAR(1),
  promo_id      NUMBER(6),
  quantity_sold NUMBER(3),
  amount_sold   NUMBER(10,2)
  )
  PARTITION BY RANGE (time_id)
  (PARTITION sales_q1_2016 VALUES LESS THAN (TO_DATE('01-APR-2016','dd-MON-yyyy')),
   PARTITION sales_q2_2016 VALUES LESS THAN (TO_DATE('01-JUL-2016','dd-MON-yyyy')),
   PARTITION sales_q3_2016 VALUES LESS THAN (TO_DATE('01-OCT-2016','dd-MON-yyyy')),
   PARTITION sales_q4_2016 VALUES LESS THAN (TO_DATE('01-JAN-2017','dd-MON-yyyy'))
   );

ALTER TABLE orders
  SPLIT PARTITION sales_q4_2016 INTO
  (PARTITION sales_oct_2016 VALUES LESS THAN (TO_DATE('01-NOV-2016','dd-MON-yyyy')),
   PARTITION sales_nov_2016 VALUES LESS THAN (TO_DATE('01-DEC-2016','dd-MON-yyyy')),
   PARTITION sales_dec_2016
   )
  ONLINE;
```

4.4.13.2 Splitting a Partition of a List-Partitioned Table

You can split a list partition with the `ALTER TABLE SPLIT PARTITION` statement.

The `SPLIT PARTITION` clause enables you to specify a list of literal values that define a partition into which rows with corresponding partitioning key values are inserted. The remaining rows of the original partition are inserted into a second partition whose value list contains the remaining values from the original partition. You can optionally specify new attributes for the two partitions that result from the split.

The following statement splits the partition `region_east` into two partitions:

```
ALTER TABLE sales_by_region
   SPLIT PARTITION region_east VALUES ('CT', 'MA', 'MD')
   INTO
    ( PARTITION region_east_1
         TABLESPACE tbs2,
      PARTITION region_east_2
        STORAGE (INITIAL 8M))
   PARALLEL 5;
```

The literal value list for the original `region_east` partition was specified as:

```
PARTITION region_east VALUES ('MA','NY','CT','NH','ME','MD','VA','PA','NJ')
```

The two new partitions are:

- `region_east_1` with a literal value list of `('CT','MA','MD')`
- `region_east_2` inheriting the remaining literal value list of `('NY','NH','ME','VA','PA','NJ')`

The individual partitions have new physical attributes specified at the partition level. The operation is executed with parallelism of degree 5.

You can split a default list partition just like you split any other list partition. This is also the only means of adding a new partition to a list-partitioned table that contains a default partition. When you split the default partition, you create a new partition defined by the values that you specify, and a second partition that remains the default partition.

> **Live SQL:**
>
> View and run a related example on Oracle Live SQL at *Oracle Live SQL: Splitting the DEFAULT Partition of a List-Partitioned Table*.

Example 4-38 Splitting the default partition of a list-partitioned table

This example splits the default partition of `sales_by_region`, creating new partitions.

```
CREATE TABLE sales_by_region
     (dept_number         NUMBER NOT NULL,
      dept_name           VARCHAR2(20),
      quarterly_sales     NUMBER(10,2),
      state               VARCHAR2(2)
 )
 PARTITION BY LIST (state)
 (
```

```
  PARTITION yearly_north VALUES ('MN','WI','MI'),
  PARTITION yearly_south VALUES ('NM','TX','GA'),
  PARTITION yearly_east VALUES  ('MA','NY','NC'),
  PARTITION yearly_west VALUES  ('CA','OR','WA'),
  PARTITION unknown VALUES (DEFAULT)
 );

SELECT PARTITION_NAME, HIGH_VALUE FROM USER_TAB_PARTITIONS WHERE TABLE_NAME ='SALES_BY_REGION';
PARTITION_NAME       HIGH_VALUE
--------------       ---------------
UNKNOWN              DEFAULT
YEARLY_EAST          'MA', 'NY', 'NC'
YEARLY_NORTH         'MN', 'WI', 'MI'
YEARLY_SOUTH         'NM', 'TX', 'GA'
YEARLY_WEST          'CA', 'OR', 'WA
5 rows selected.

INSERT INTO SALES_BY_REGION VALUES (002, 'AUTO NORTH', 450000, 'MN');
INSERT INTO SALES_BY_REGION VALUES (002, 'AUTO NORTH', 495000, 'WI');
INSERT INTO SALES_BY_REGION VALUES (002, 'AUTO NORTH', 850000, 'MI');

INSERT INTO SALES_BY_REGION VALUES (004, 'AUTO SOUTH',  595000, 'NM');
INSERT INTO SALES_BY_REGION VALUES (004, 'AUTO SOUTH', 4825000, 'TX');
INSERT INTO SALES_BY_REGION VALUES (004, 'AUTO SOUTH',  945000, 'GA');

INSERT INTO SALES_BY_REGION VALUES (006, 'AUTO EAST', 2125000, 'MA');
INSERT INTO SALES_BY_REGION VALUES (006, 'AUTO EAST', 6101000, 'NY');
INSERT INTO SALES_BY_REGION VALUES (006, 'AUTO EAST',  741000, 'NC');

INSERT INTO SALES_BY_REGION VALUES (008, 'AUTO WEST', 7201000, 'CA');
INSERT INTO SALES_BY_REGION VALUES (008, 'AUTO WEST',  901000, 'OR');
INSERT INTO SALES_BY_REGION VALUES (008, 'AUTO WEST', 1125000, 'WA');

INSERT INTO SALES_BY_REGION VALUES (009, 'AUTO MIDWEST', 1950000, 'AZ');
INSERT INTO SALES_BY_REGION VALUES (009, 'AUTO MIDWEST', 5725000, 'UT');

SELECT DEPT_NUMBER, DEPT_NAME, QUARTERLY_SALES, STATE FROM SALES_BY_REGION PARTITION(yearly_north);
DEPT_NUMBER DEPT_NAME            QUARTERLY_SALES ST
----------- -------------------- --------------- --
2           AUTO NORTH                    450000 MN
2           AUTO NORTH                    495000 WI
2           AUTO NORTH                    850000 MI

SELECT DEPT_NUMBER, DEPT_NAME, QUARTERLY_SALES, STATE FROM SALES_BY_REGION PARTITION(yearly_south);
DEPT_NUMBER DEPT_NAME            QUARTERLY_SALES ST
----------- -------------------- --------------- --
4           AUTO SOUTH                    595000 NM
4           AUTO SOUTH                   4825000 TX
4           AUTO SOUTH                    945000 GA

…

SELECT DEPT_NUMBER, DEPT_NAME, QUARTERLY_SALES, STATE FROM SALES_BY_REGION PARTITION(unknown);
DEPT_NUMBER DEPT_NAME            QUARTERLY_SALES ST
----------- -------------------- --------------- --
9           AUTO MIDWEST                 1950000 AZ
9           AUTO MIDWEST                 5725000 UT

REM Note that the following ADD PARTITION statement fails. This action fails because
REM all undefined values are automatically included in the DEFAULT partition.
ALTER TABLE sales_by_region ADD PARTITION yearly_midwest VALUES ('AZ', 'UT');
ORA-14323: cannot add partition when DEFAULT partition exists
```

```
REM You must SPLIT the DEFAULT partition to add a new partition.
ALTER TABLE sales_by_region
   SPLIT PARTITION unknown VALUES ('AZ', 'UT')
   INTO
    ( PARTITION yearly_midwest,
      PARTITION unknown);

SELECT PARTITION_NAME, HIGH_VALUE FROM USER_TAB_PARTITIONS WHERE TABLE_NAME ='SALES_BY_REGION';
PARTITION_NAME       HIGH_VALUE
--------------       ---------------
UNKNOWN              DEFAULT
YEARLY_EAST          'MA', 'NY', 'NC'
YEARLY_MIDWEST       'AZ', 'UT'
YEARLY_NORTH         'MN', 'WI', 'MI'
YEARLY_SOUTH         'NM', 'TX', 'GA'
YEARLY_WEST          'CA', 'OR', 'WA'
6 Rows selected.

SELECT DEPT_NUMBER, DEPT_NAME, QUARTERLY_SALES, STATE FROM SALES_BY_REGION PARTITION(yearly_midwest);
DEPT_NUMBER DEPT_NAME            QUARTERLY_SALES ST
----------- -------------------- --------------- --
          9 AUTO MIDWEST                1950000 AZ
          9 AUTO MIDWEST                5725000 UT

SELECT DEPT_NUMBER, DEPT_NAME, QUARTERLY_SALES, STATE FROM SALES_BY_REGION PARTITION(unknown);
no rows selected

REM Split the DEFAULT partition again to add a new 'yearly_mideast' partition.
ALTER TABLE sales_by_region
   SPLIT PARTITION unknown VALUES ('OH', 'IL')
   INTO
    ( PARTITION yearly_mideast,
      PARTITION unknown);
Table altered.

SELECT PARTITION_NAME, HIGH_VALUE FROM USER_TAB_PARTITIONS WHERE TABLE_NAME ='SALES_BY_REGION';
PARTITION_NAME      HIGH_VALUE
------------------  ------------------
UNKNOWN             DEFAULT
YEARLY_EAST         'MA', 'NY', 'NC'
YEARLY_MIDEAST      'OH', 'IL'
YEARLY_MIDWEST      'AZ', 'UT'
YEARLY_NORTH        'MN', 'WI', 'MI'
YEARLY_SOUTH        'NM', 'TX', 'GA'
YEARLY_WEST         'CA', 'OR', 'WA'
7 rows selected.

INSERT INTO SALES_BY_REGION VALUES (007, 'AUTO MIDEAST',  925000, 'OH');
INSERT INTO SALES_BY_REGION VALUES (007, 'AUTO MIDEAST', 1325000, 'IL');

SELECT DEPT_NUMBER, DEPT_NAME, QUARTERLY_SALES, STATE FROM SALES_BY_REGION PARTITION(yearly_mideast);
DEPT_NUMBER DEPT_NAME            QUARTERLY_SALES ST
----------- -------------------- --------------- --
          7 AUTO MIDEAST                 925000 OH
          7 AUTO MIDEAST                1325000 IL

SELECT DEPT_NUMBER, DEPT_NAME, QUARTERLY_SALES, STATE FROM SALES_BY_REGION PARTITION(unknown);
no rows selected
```

4.4.13.3 Splitting a Partition of an Interval-Partitioned Table

You can split a range or a materialized interval partition with the `ALTER TABLE SPLIT PARTITION` statement in an interval-partitioned table.

Splitting a range partition in the interval-partitioned table is described in Splitting a Partition of a Range-Partitioned Table.

To split a materialized interval partition, you specify a value of the partitioning key column within the interval partition at which to split the partition. The first of the resulting two new partitions includes all rows in the original partition whose partitioning key column values map lower than the specified value. The second partition contains all rows whose partitioning key column values map greater than or equal to the specified value. The split partition operation moves the transition point up to the higher boundary of the partition you just split, and all materialized interval partitions lower than the newly split partitions are implicitly converted into range partitions, with their upper boundaries defined by the upper boundaries of the intervals.

You can optionally specify new attributes for the two range partitions resulting from the split. If there are local indexes defined on the table, then this statement also splits the matching partition in each local index. You cannot split interval partitions that have not yet been created.

The following example shows splitting the May 2007 partition in the monthly interval partitioned table `transactions`.

```
ALTER TABLE transactions
    SPLIT PARTITION FOR(TO_DATE('01-MAY-2007','dd-MON-yyyy'))
    AT (TO_DATE('15-MAY-2007','dd-MON-yyyy'));
```

4.4.13.4 Splitting a *-Hash Partition

You can split a hash partition with the `ALTER TABLE SPLIT PARTITION` statement.

This is the opposite of merging *-hash partitions. When you split *-hash partitions, the new subpartitions are rehashed into either the number of subpartitions specified in a `SUBPARTITIONS` or `SUBPARTITION` clause. Or, if no such clause is included, the new partitions inherit the number of subpartitions (and tablespaces) from the partition being split.

The inheritance of properties is different when a *-hash partition is split, versus when two *-hash partitions are merged. When a partition is split, the new partitions can inherit properties from the original partition because there is only one parent. However, when partitions are merged, properties must be inherited from *table level* defaults because there are two parents and the new partition cannot inherit from either at the expense of the other.

The following example splits a range-hash partition:

```
ALTER TABLE all_seasons SPLIT PARTITION quarter_1
     AT (TO_DATE('16-dec-1997','dd-mon-yyyy'))
     INTO (PARTITION q1_1997_1 SUBPARTITIONS 4 STORE IN (ts1,ts3),
           PARTITION q1_1997_2);
```

The rules for splitting an interval-hash partitioned table follow the rules for splitting an interval-partitioned table. As described in Splitting a Partition of an Interval-Partitioned Table, the transition point is changed to the higher boundary of the split partition.

4.4.13.5 Splitting Partitions in a *-List Partitioned Table

Partitions can be split at both the partition level and at the subpartition level in a list partitioned table..

- Splitting a *-List Partition
- Splitting a *-List Subpartition

4.4.13.5.1 Splitting a *-List Partition

You can split a list partition with the `ALTER TABLE SPLIT PARTITION` statement.

Splitting a partition of a *-list partitioned table is similar to the description in Splitting a Partition of a List-Partitioned Table. No subpartition literal value list can be specified for either of the new partitions. The new partitions inherit the subpartition descriptions from the original partition being split.

The following example splits the `q1_1999` partition of the `quarterly_regional_sales` table:

```
ALTER TABLE quarterly_regional_sales SPLIT PARTITION q1_1999
   AT (TO_DATE('15-Feb-1999','dd-mon-yyyy'))
   INTO ( PARTITION q1_1999_jan_feb
             TABLESPACE ts1,
          PARTITION q1_1999_feb_mar
             STORAGE (INITIAL 8M) TABLESPACE ts2)
   PARALLEL 5;
```

This operation splits the partition `q1_1999` into two resulting partitions: `q1_1999_jan_feb` and `q1_1999_feb_mar`. Both partitions inherit their subpartition descriptions from the original partition. The individual partitions have new physical attributes, including tablespaces, specified at the partition level. These new attributes become the default attributes of the new partitions. This operation is run with parallelism of degree 5.

The `ALTER TABLE SPLIT PARTITION` statement provides no means of specifically naming subpartitions resulting from the split of a partition in a composite partitioned table. However, for those subpartitions in the parent partition with names of the form *`partition name_subpartition name`*, the database generates corresponding names in the newly created subpartitions using the new partition names. All other subpartitions are assigned system generated names of the form `SYS_SUBP`*`n`*. System generated names are also assigned for the subpartitions of any partition resulting from the split for which a name is not specified. Unnamed partitions are assigned a system generated partition name of the form `SYS_P`*`n`*.

The following query displays the subpartition names resulting from the previous split partition operation on table `quarterly_regional_sales`. It also reflects the results of other operations performed on this table in preceding sections of this chapter since its creation in About Creating Composite Range-List Partitioned Tables.

```
SELECT PARTITION_NAME, SUBPARTITION_NAME, TABLESPACE_NAME
  FROM DBA_TAB_SUBPARTITIONS
  WHERE TABLE_NAME='QUARTERLY_REGIONAL_SALES'
  ORDER BY PARTITION_NAME;

PARTITION_NAME       SUBPARTITION_NAME              TABLESPACE_NAME
-------------------- ------------------------------ ---------------
Q1_1999_FEB_MAR      Q1_1999_FEB_MAR_WEST           TS2
Q1_1999_FEB_MAR      Q1_1999_FEB_MAR_NORTHEAST      TS2
Q1_1999_FEB_MAR      Q1_1999_FEB_MAR_SOUTHEAST      TS2
Q1_1999_FEB_MAR      Q1_1999_FEB_MAR_NORTHCENTRAL   TS2
```

```
Q1_1999_FEB_MAR      Q1_1999_FEB_MAR_SOUTHCENTRAL   TS2
Q1_1999_FEB_MAR      Q1_1999_FEB_MAR_SOUTH          TS2
Q1_1999_JAN_FEB      Q1_1999_JAN_FEB_WEST           TS1
Q1_1999_JAN_FEB      Q1_1999_JAN_FEB_NORTHEAST      TS1
Q1_1999_JAN_FEB      Q1_1999_JAN_FEB_SOUTHEAST      TS1
Q1_1999_JAN_FEB      Q1_1999_JAN_FEB_NORTHCENTRAL   TS1
Q1_1999_JAN_FEB      Q1_1999_JAN_FEB_SOUTHCENTRAL   TS1
Q1_1999_JAN_FEB      Q1_1999_JAN_FEB_SOUTH          TS1
Q1_2000              Q1_2000_NORTHWEST              TS3
Q1_2000              Q1_2000_SOUTHWEST              TS3
Q1_2000              Q1_2000_NORTHEAST              TS3
Q1_2000              Q1_2000_SOUTHEAST              TS3
Q1_2000              Q1_2000_NORTHCENTRAL           TS3
Q1_2000              Q1_2000_SOUTHCENTRAL           TS3
Q2_1999              Q2_1999_NORTHWEST              TS4
Q2_1999              Q2_1999_SOUTHWEST              TS4
Q2_1999              Q2_1999_NORTHEAST              TS4
Q2_1999              Q2_1999_SOUTHEAST              TS4
Q2_1999              Q2_1999_NORTHCENTRAL           TS4
Q2_1999              Q2_1999_SOUTHCENTRAL           TS4
Q3_1999              Q3_1999_NORTHWEST              TS4
Q3_1999              Q3_1999_SOUTHWEST              TS4
Q3_1999              Q3_1999_NORTHEAST              TS4
Q3_1999              Q3_1999_SOUTHEAST              TS4
Q3_1999              Q3_1999_NORTHCENTRAL           TS4
Q3_1999              Q3_1999_SOUTHCENTRAL           TS4
Q4_1999              Q4_1999_NORTHWEST              TS4
Q4_1999              Q4_1999_SOUTHWEST              TS4
Q4_1999              Q4_1999_NORTHEAST              TS4
Q4_1999              Q4_1999_SOUTHEAST              TS4
Q4_1999              Q4_1999_NORTHCENTRAL           TS4
Q4_1999              Q4_1999_SOUTHCENTRAL           TS4

36 rows selected.
```

4.4.13.5.2 Splitting a *-List Subpartition

You can split a list subpartition with the `ALTER TABLE SPLIT SUBPARTITION` statement.

Splitting a list subpartition of a *-list partitioned table is similar to the description in Splitting a Partition of a List-Partitioned Table, but the syntax is that of `SUBPARTITION` rather than `PARTITION`. For example, the following statement splits a subpartition of the `quarterly_regional_sales` table:

```
ALTER TABLE quarterly_regional_sales SPLIT SUBPARTITION q2_1999_southwest
   VALUES ('UT') INTO
      ( SUBPARTITION q2_1999_utah
           TABLESPACE ts2,
        SUBPARTITION q2_1999_southwest
           TABLESPACE ts3
      )
   PARALLEL;
```

This operation splits the subpartition `q2_1999_southwest` into two subpartitions:

- `q2_1999_utah` with literal value list of `('UT')`
- `q2_1999_southwest` which inherits the remaining literal value list of `('AZ','NM')`

The individual subpartitions have new physical attributes that are inherited from the subpartition being split.

You can only split subpartitions in an interval-list partitioned table for range partitions or materialized interval partitions. To change subpartition values for future interval partitions, you must modify the subpartition template.

4.4.13.6 Splitting a *-Range Partition

You can split a range partition using the `ALTER TABLE SPLIT PARTITION` statement.

Splitting a partition of a *-range partitioned table is similar to the description in Splitting a Partition of a Range-Partitioned Table. No subpartition range values can be specified for either of the new partitions. The new partitions inherit the subpartition descriptions from the original partition being split.

The following example splits the May 2007 interval partition of the interval-range partitioned `orders` table:

```
ALTER TABLE orders
    SPLIT PARTITION FOR(TO_DATE('01-MAY-2007','dd-MON-yyyy'))
    AT (TO_DATE('15-MAY-2007','dd-MON-yyyy'))
    INTO (PARTITION p_fh_may07,PARTITION p_sh_may2007);
```

This operation splits the interval partition `FOR('01-MAY-2007')` into two resulting partitions: `p_fh_may07` and `p_sh_may_2007`. Both partitions inherit their subpartition descriptions from the original partition. Any interval partitions before the June 2007 partition have been converted into range partitions, as described in Merging Interval Partitions.

The `ALTER TABLE SPLIT PARTITION` statement provides no means of specifically naming subpartitions resulting from the split of a partition in a composite partitioned table. However, for those subpartitions in the parent partition with names of the form *`partition name_subpartition name`*, the database generates corresponding names in the newly created subpartitions using the new partition names. All other subpartitions are assigned system generated names of the form `SYS_SUBP`*`n`*. System generated names are also assigned for the subpartitions of any partition resulting from the split for which a name is not specified. Unnamed partitions are assigned a system generated partition name of the form `SYS_P`*`n`*.

The following query displays the subpartition names and high values resulting from the previous split partition operation on table `orders`. It also reflects the results of other operations performed on this table in preceding sections of this chapter since its creation.

```
BREAK ON partition_name

SELECT partition_name, subpartition_name, high_value
FROM user_tab_subpartitions
WHERE table_name = 'ORCERS'
ORDER BY partition_name, subpartition_position;

PARTITION_NAME           SUBPARTITION_NAME              HIGH_VALUE
------------------------ ------------------------------ ---------------
P_BEFORE_2007            P_BEFORE_2007_P_SMALL          1000
                         P_BEFORE_2007_P_MEDIUM         10000
                         P_BEFORE_2007_P_LARGE          100000
                         P_BEFORE_2007_P_EXTRAORDINARY  MAXVALUE
P_FH_MAY07               SYS_SUBP2985                   1000
                         SYS_SUBP2986                   10000
                         SYS_SUBP2987                   100000
                         SYS_SUBP2988                   MAXVALUE
P_PRE_MAY_2007           P_PRE_MAY_2007_P_SMALL         1000
                         P_PRE_MAY_2007_P_MEDIUM        10000
                         P_PRE_MAY_2007_P_LARGE         100000
                         P_PRE_MAY_2007_P_EXTRAORDINARY MAXVALUE
```

```
P_SH_MAY2007            SYS_SUBP2989                1000
                        SYS_SUBP2990                10000
                        SYS_SUBP2991                100000
                        SYS_SUBP2992                MAXVALUE
```

4.4.13.6.1 Splitting a *-Range Subpartition

You can split a range subpartition using the `ALTER TABLE SPLIT SUBPARTITION` statement.

Splitting a range subpartition of a *-range partitioned table is similar to the description in Splitting a Partition of a Range-Partitioned Table, but the syntax is that of `SUBPARTITION` rather than `PARTITION`. For example, the following statement splits a subpartition of the `orders` table:

```
ALTER TABLE orders
SPLIT SUBPARTITION p_pre_may_2007_p_large AT (50000)
INTO (SUBPARTITION p_pre_may_2007_med_large TABLESPACE TS4
     , SUBPARTITION p_pre_may_2007_large_large TABLESPACE TS5
     );
```

This operation splits the subpartition `p_pre_may_2007_p_large` into two subpartitions:

- `p_pre_may_2007_med_large` with values between 10000 and 50000
- `p_pre_may_2007_large_large` with values between 50000 and 100000

The individual subpartitions have new physical attributes that are inherited from the subpartition being split.

You can only split subpartitions in an interval-range partitioned table for range partitions or materialized interval partitions. To change subpartition boundaries for future interval partitions, you must modify the subpartition template.

4.4.13.7 Splitting Index Partitions

You cannot explicitly split a partition in a local index. A local index partition is split only when you split a partition in the underlying table.

However, you can split a global index partition as is done in the following example:

```
ALTER INDEX quon1 SPLIT
    PARTITION canada AT ( 100 ) INTO
    PARTITION canada1 ..., PARTITION canada2 ...);
ALTER INDEX quon1 REBUILD PARTITION canada1;
ALTER INDEX quon1 REBUILD PARTITION canada2;
```

The index being split can contain index data, and the resulting partitions do not require rebuilding, unless the original partition was previously marked `UNUSABLE`.

4.4.13.8 Splitting into Multiple Partitions

You can redistribute the contents of one partition or subpartition into multiple partitions or subpartitions with the `SPLIT PARTITION` and `SPLIT SUBPARTITION` clauses of the `ALTER TABLE` statement.

When splitting multiple partitions, the segment associated with the current partition is discarded. Each new partitions obtains a new segment and inherits all unspecified physical attributes from the current source partition. Fast split optimization is applied to multipartition split operations when required conditions are met.

You can use the extended split syntax to specify a list of new partition descriptions similar to the create partitioned table SQL statements, rather than specifying the `AT` or `VALUES` clause. Additionally, the range or list values clause for the last new partition description is derived based on the high bound of the source partition and the bound values specified for the first (N-1) new partitions resulting from the split.

The following SQL statements are examples of splitting a partition into multiple partitions.

```
ALTER TABLE SPLIT PARTITION p0 INTO
  (PARTITION p01 VALUES LESS THAN (25),
   PARTITION p02 VALUES LESS THAN (50),
   PARTITION p03 VALUES LESS THAN (75),
   PARTITION p04);

ALTER TABLE SPLIT PARTITION p0 INTO
  (PARTITION p01 VALUES LESS THAN (25),
   PARTITION p02);
```

In the second SQL example, partition `p02` has the high bound of the original partition `p0`.

To split a range partition into N partitions, (N-1) values of the partitioning key column must be specified within the range of the partition at which to split the partition. The new non-inclusive upper bound values specified must be in ascending order. The high bound of Nth new partition is assigned the value of the high bound of the partition being split. The names and physical attributes of the N new partitions resulting from the split can be optionally specified.

To split a list partition into N partitions, (N-1) lists of literal values must be specified, each of which defines the first (N-1) partitions into which rows with corresponding partitioning key values are inserted. The remaining rows of the original partition are inserted into the Nth new partition whose value list contains the remaining literal values from the original partition. No two value lists can contain the same partition value. The (N-1) value lists that are specified cannot contain all of the partition values of the current partition because the Nth new partition would be empty. Also, the (N-1) value lists cannot contain any partition values that do not exist for the current partition.

When splitting a `DEFAULT` list partition or a `MAXVALUE` range partition into multiple partitions, the first (N-1) new partitions are created using the literal value lists or high bound values specified, while the Nth new partition resulting from the split have the `DEFAULT` value or `MAXVALUE`. Splitting a partition of a composite partitioned table into multiple partitions assumes the existing behavior with respect to inheritance of the number, names, bounds and physical properties of the subpartitions of the new partitions resulting from the split. The `SPLIT_TABLE_SUBPARTITION` clause is extended similarly to allow split of a range or list subpartition into N new subpartitions.

The behavior of the SQL statement with respect to local and global indexes remains unchanged. Corresponding local index partition are split into multiple partitions. If the partitioned table contains LOB columns, then existing semantics for the `SPLIT PARTITION` clause apply with the extended syntax; that is, LOB data and index segments is dropped for current partition and new segments are created for each LOB column for each new partition. Fast split optimization is applied to multipartition split operations when required conditions are met.

For example, the following SQL statement splits the `sales_Q4_2007` partition of the partitioned by range table `sales` splits into five partitions corresponding to the quarters of the next year. In this example, the partition `sales_Q4_2008` implicitly becomes the high bound of the split partition.

```
ALTER TABLE sales SPLIT PARTITION sales_Q4_2007 INTO
( PARTITION sales_Q4_2007 VALUES LESS THAN (TO_DATE('01-JAN-2008','dd-MON-yyyy')),
  PARTITION sales_Q1_2008 VALUES LESS THAN (TO_DATE('01-APR-2008','dd-MON-yyyy')),
  PARTITION sales_Q2_2008 VALUES LESS THAN (TO_DATE('01-JUL-2008','dd-MON-yyyy')),
```

```
PARTITION sales_Q3_2008 VALUES LESS THAN (TO_DATE('01-OCT-2008','dd-MON-yyyy')),
PARTITION sales_Q4_2008);
```

For the sample table `customers` partitioned by list, the following statement splits the partition **Europe** into three partitions.

```
ALTER TABLE list_customers SPLIT PARTITION Europe INTO
  (PARTITION western-europe VALUES ('GERMANY', 'FRANCE'),
   PARTITION southern-europe VALUES ('ITALY'),
   PARTITION rest-europe);
```

See Also:

Restrictions and Special Requirements for Multi-Partition Operations on Tables with Domain Indexes

4.4.13.9 Fast SPLIT PARTITION and SPLIT SUBPARTITION Operations

Oracle Database implements a `SPLIT PARTITION` operation by creating two or more new partitions and redistributing the rows from the partition being split into the new partitions.

This is a time-consuming operation because it is necessary to scan all the rows of the partition being split and then insert them one-by-one into the new partitions. Further if you do not use the `UPDATE INDEXES` clause, then both local and global indexes also require rebuilding.

Sometimes after a split operation, one new partition contains all of the rows from the partition being split, while the other partitions contain no rows. This is often the case when splitting the first or last partition of a table. The database can detect such situations and can optimize the split operation. This optimization results in a fast split operation that behaves like an add partition operation.

Specifically, the database can optimize and speed up `SPLIT PARTITION` operations if all of the following conditions are met:

- One of the resulting partitions contains all of the rows.
- The non-empty resulting partition must have storage characteristics identical to those of the partition being split. Specifically:
  - If the partition being split is composite, then the storage characteristics of each subpartition in the new resulting partition must be identical to those of the subpartitions of the partition being split.
  - If the partition being split contains a `LOB` column, then the storage characteristics of each `LOB` (sub)partition in the new non-empty resulting partition must be identical to those of the `LOB` (sub)partitions of the partition being split.
  - If a partition of an index-organized table with overflow is being split, then the storage characteristics of each overflow (sub)partition in the new nonempty resulting partition must be identical to those of the overflow (sub)partitions of the partition being split.
  - If a partition of an index-organized table with mapping table is being split, then the storage characteristics of each mapping table (sub)partition in the new nonempty resulting partition must be identical to those of the mapping table (sub)partitions of the partition being split.

If these conditions are met after the split, then all global indexes remain usable, even if you did not specify the `UPDATE INDEXES` clause. Local index (sub)partitions associated with the resulting

partitions remain usable if they were usable before the split. Local index (sub)partitions corresponding to the non-empty resulting partition are identical to the local index (sub)partitions of the partition that was split. The same optimization holds for `SPLIT SUBPARTITION` operations.

4.4.14 About Truncating Partitions and Subpartitions

Truncating a partition is similar to dropping a partition, except that the partition is emptied of its data, but not physically dropped.

Use the `ALTER TABLE TRUNCATE PARTITION` statement to remove all rows from a table partition. You cannot truncate an index partition. However, if local indexes are defined for the table, the `ALTER TABLE TRUNCATE PARTITION` statement truncates the matching partition in each local index. Unless you specify `UPDATE INDEXES`, any global indexes are marked `UNUSABLE` and must be rebuilt. You cannot use `UPDATE INDEXES` for index-organized tables. Use `UPDATE GLOBAL INDEXES` instead.

This section contains the following topics:

- About Truncating a Table Partition
- Truncating Multiple Partitions
- Truncating Subpartitions
- Truncating a Partition with the Cascade Option

> **See Also:**
>
> - Asynchronous Global Index Maintenance for Dropping and Truncating Partitions for information about asynchronous index maintenance for truncating partitions
> - About Dropping Partitions and Subpartitions for information about dropping a partition

4.4.14.1 About Truncating a Table Partition

Use the `ALTER TABLE TRUNCATE PARTITION` statement to remove all rows from a table partition, with or without reclaiming space.

Truncating a partition in an interval-partitioned table does not move the transition point. You can truncate partitions and subpartitions in a reference-partitioned table.

- Truncating Table Partitions Containing Data and Global Indexes
- Truncating a Partition Containing Data and Referential Integrity Constraints

4.4.14.1.1 Truncating Table Partitions Containing Data and Global Indexes

When truncating a table partition that contains data and global indexes, you can use one of several methods.

If the partition contains data and global indexes, use one of the following methods (method 1, 2, or 3) to truncate the table partition.

Method 1

Leave the global indexes in place during the `ALTER TABLE TRUNCATE PARTITION` statement. In this example, table `sales` has a global index `sales_area_ix`, which is rebuilt.

```
ALTER TABLE sales TRUNCATE PARTITION dec98;
ALTER INDEX sales_area_ix REBUILD;
```

This method is most appropriate for large tables where the partition being truncated contains a significant percentage of the total data in the table.

Method 2

Run the `DELETE` statement to delete all rows from the partition before you issue the `ALTER TABLE TRUNCATE PARTITION` statement. The `DELETE` statement updates the global indexes, and also fires triggers and generates redo and undo logs.

For example, to truncate the first partition, run the following statements:

```
DELETE FROM sales PARTITION (dec98);
ALTER TABLE sales TRUNCATE PARTITION dec98;
```

This method is most appropriate for small tables, or for large tables when the partition being truncated contains a small percentage of the total data in the table.

Method 3

Specify `UPDATE INDEXES` in the `ALTER TABLE` statement. This causes the global index to be truncated at the time the partition is truncated.

```
ALTER TABLE sales TRUNCATE PARTITION dec98
     UPDATE INDEXES;
```

With asynchronous global index maintenance, this operation is a metadata-only operation.

4.4.14.1.2 Truncating a Partition Containing Data and Referential Integrity Constraints

If a partition contains data and has referential integrity constraints, then you cannot truncate the partition. However, if no other data is referencing any data in the partition to remove, then you can use one of several methods.

Choose either of the following methods (method 1 or 2) to truncate the table partition.

Method 1

Disable the integrity constraints, run the `ALTER TABLE TRUNCATE PARTITION` statement, then re-enable the integrity constraints. This method is most appropriate for large tables where the partition being truncated contains a significant percentage of the total data in the table. If there is still referencing data in other tables, then you must remove that data to be able to re-enable the integrity constraints.

Method 2

Issue the `DELETE` statement to delete all rows from the partition before you issue the `ALTER TABLE TRUNCATE PARTITION` statement. The `DELETE` statement enforces referential integrity constraints, and also fires triggers and generates redo and undo logs. Data in referencing tables is deleted if the foreign key constraints were created with the `ON DELETE CASCADE` option.

```
DELETE FROM sales partition (dec94);
ALTER TABLE sales TRUNCATE PARTITION dec94;
```

This method is most appropriate for small tables, or for large tables when the partition being truncated contains a small percentage of the total data in the table.

4.4.14.2 Truncating Multiple Partitions

You can truncate multiple partitions from a range or list partitioned table with the `TRUNCATE PARTITION` clause of the `ALTER TABLE` statement.

The corresponding partitions of local indexes are truncated in the operation. Global indexes must be rebuilt unless `UPDATE INDEXES` is specified.

In the following example, the `ALTER TABLE` SQL statement truncates multiple partitions in a table. Note that the data is truncated, but the partitions are not dropped.

> **Live SQL:**
>
> View and run a related example on Oracle Live SQL at *Oracle Live SQL: Truncating a Range-Partitioned Table*.

Example 4-39 Truncating Multiple Partitions

```
CREATE TABLE sales_partition_truncate
  ( product_id       NUMBER(6) NOT NULL,
    customer_id      NUMBER    NOT NULL,
    channel_id       CHAR(1),
    promo_id         NUMBER(6),
    sales_date       DATE,
    quantity_sold    INTEGER,
    amount_sold      NUMBER(10,2)
  )
 PARTITION BY RANGE (sales_date)
 SUBPARTITION BY LIST (channel_id)
(  PARTITION q3_2018 VALUES LESS THAN (TO_DATE('1-OCT-2018','DD-MON-YYYY'))
   ( SUBPARTITION q3_2018_p_catalog VALUES ('C'),
     SUBPARTITION q3_2018_p_internet VALUES ('I'),
     SUBPARTITION q3_2018_p_partners VALUES ('P'),
     SUBPARTITION q3_2018_p_direct_sales VALUES ('S'),
     SUBPARTITION q3_2018_p_tele_sales VALUES ('T')
   ),
  PARTITION q4_2018 VALUES LESS THAN (TO_DATE('1-JAN-2019','DD-MON-YYYY'))
   ( SUBPARTITION q4_2018_p_catalog VALUES ('C'),
     SUBPARTITION q4_2018_p_internet VALUES ('I'),
     SUBPARTITION q4_2018_p_partners VALUES ('P'),
     SUBPARTITION q4_2018_p_direct_sales VALUES ('S'),
     SUBPARTITION q4_2018_p_tele_sales VALUES ('T')
   ),
  PARTITION q1_2019 VALUES LESS THAN (TO_DATE('1-APR-2019','DD-MON-YYYY'))
   ( SUBPARTITION q1_2019_p_catalog VALUES ('C')
   , SUBPARTITION q1_2019_p_internet VALUES ('I')
   , SUBPARTITION q1_2019_p_partners VALUES ('P')
   , SUBPARTITION q1_2019_p_direct_sales VALUES ('S')
   , SUBPARTITION q1_2019_p_tele_sales VALUES ('T')
   ),
  PARTITION q2_2019 VALUES LESS THAN (TO_DATE('1-JUL-2019','DD-MON-YYYY'))
   ( SUBPARTITION q2_2019_p_catalog VALUES ('C'),
```

```
    SUBPARTITION q2_2019_p_internet VALUES ('I'),
    SUBPARTITION q2_2019_p_partners VALUES ('P'),
    SUBPARTITION q2_2019_p_direct_sales VALUES ('S'),
    SUBPARTITION q2_2019_p_tele_sales VALUES ('T')
  ),
 PARTITION q3_2019 VALUES LESS THAN (TO_DATE('1-OCT-2019','DD-MON-YYYY'))
  ( SUBPARTITION q3_2019_p_catalog VALUES ('C'),
    SUBPARTITION q3_2019_p_internet VALUES ('I'),
    SUBPARTITION q3_2019_p_partners VALUES ('P'),
    SUBPARTITION q3_2019_p_direct_sales VALUES ('S'),
    SUBPARTITION q3_2019_p_tele_sales VALUES ('T')
  ),
 PARTITION q4_2019 VALUES LESS THAN (TO_DATE('1-JAN-2020','DD-MON-YYYY'))
  ( SUBPARTITION q4_2019_p_catalog VALUES ('C'),
    SUBPARTITION q4_2019_p_internet VALUES ('I'),
    SUBPARTITION q4_2019_p_partners VALUES ('P'),
    SUBPARTITION q4_2019_p_direct_sales VALUES ('S'),
    SUBPARTITION q4_2019_p_tele_sales VALUES ('T')
  )
 );

SELECT TABLE_NAME, PARTITION_NAME, SUBPARTITION_NAME FROM USER_TAB_SUBPARTITIONS
       WHERE TABLE_NAME ='SALES_PARTITION_TRUNCATE';
TABLE_NAME                PARTITION_NAME    SUBPARTITION_NAME
------------------------  ---------------   -------------------
SALES_PARTITION_TRUNCATE  Q1_2019           Q1_2019_P_CATALOG
SALES_PARTITION_TRUNCATE  Q1_2019           Q1_2019_P_DIRECT_SALES
...
30 rows selected.

INSERT INTO sales_partition_truncate VALUES (1001,100,'C',150,'10-SEP-2018',500,2000);
INSERT INTO sales_partition_truncate VALUES (1021,200,'C',160,'16-NOV-2018',100,1500);
INSERT INTO sales_partition_truncate VALUES (1001,100,'C',150,'10-FEB-2019',500,2000);
INSERT INTO sales_partition_truncate VALUES (1021,200,'S',160,'16-FEB-2019',100,1500);
INSERT INTO sales_partition_truncate VALUES (1002,110,'I',180,'15-JUN-2019',100,1000);
INSERT INTO sales_partition_truncate VALUES (5010,150,'P',200,'20-AUG-2019',1000,10000);
INSERT INTO sales_partition_truncate VALUES (1001,100,'T',150,'12-OCT-2019',500,2000);

SELECT * FROM sales_partition_truncate;
PRODUCT_ID CUSTOMER_ID C   PROMO_ID SALES_DAT QUANTITY_SOLD AMOUNT_SOLD
---------- ----------- - ---------- --------- ------------- -----------
      1001         100 C        150 10-SEP-18           500        2000
      1021         200 C        160 16-NOV-18           100        1500
      1001         100 C        150 10-FEB-19           500        2000
      1021         200 S        160 16-FEB-19           100        1500
      1002         110 I        180 15-JUN-19           100        1000
      5010         150 P        200 20-AUG-19          1000       10000
      1001         100 T        150 12-OCT-19           500        2000
7 rows selected.

ALTER TABLE sales_partition_truncate
   TRUNCATE PARTITIONS q3_2018, q4_2018;

SELECT * FROM sales_partition_truncate;
PRODUCT_ID CUSTOMER_ID C   PROMO_ID SALES_DAT QUANTITY_SOLD AMOUNT_SOLD
---------- ----------- - ---------- --------- ------------- -----------
      1001         100 C        150 10-FEB-19           500        2000
      1021         200 S        160 16-FEB-19           100        1500
      1002         110 I        180 15-JUN-19           100        1000
      5010         150 P        200 20-AUG-19          1000       10000
      1001         100 T        150 12-OCT-19           500        2000
5 rows selected.
```

```
SELECT TABLE_NAME, PARTITION_NAME, SUBPARTITION_NAME FROM USER_TAB_SUBPARTITIONS
       WHERE TABLE_NAME ='SALES_PARTITION_TRUNCATE';
TABLE_NAME                PARTITION_NAME    SUBPARTITION_NAME
------------------------  ---------------   -------------------
SALES_PARTITION_TRUNCATE    Q1_2019           Q1_2019_P_CATALOG
SALES_PARTITION_TRUNCATE    Q1_2019           Q1_2019_P_DIRECT_SALES
...
SALES_PARTITION_TRUNCATE    Q3_2018           Q3_2018_P_CATALOG
SALES_PARTITION_TRUNCATE    Q3_2018           Q3_2018_P_DIRECT_SALES
SALES_PARTITION_TRUNCATE    Q3_2018           Q3_2018_P_INTERNET
SALES_PARTITION_TRUNCATE    Q3_2018           Q3_2018_P_PARTNERS
SALES_PARTITION_TRUNCATE    Q3_2018           Q3_2018_P_TELE_SALES
...
SALES_PARTITION_TRUNCATE    Q4_2018           Q4_2018_P_CATALOG
SALES_PARTITION_TRUNCATE    Q4_2018           Q4_2018_P_DIRECT_SALES
SALES_PARTITION_TRUNCATE    Q4_2018           Q4_2018_P_INTERNET
SALES_PARTITION_TRUNCATE    Q4_2018           Q4_2018_P_PARTNERS
SALES_PARTITION_TRUNCATE    Q4_2018           Q4_2018_P_TELE_SALES
...
30 rows selected.
```

Note:

If a table has a local domain index, truncating a single partition or the entire table is supported. Truncating multiple partitions in a single `ALTER TABLE` statement is not permitted if a local index is present.

See Also:

Restrictions and Special Requirements for Multi-Partition Operations on Tables with Domain Indexes

4.4.14.3 Truncating Subpartitions

Use the `ALTER TABLE TRUNCATE SUBPARTITION` statement to remove all rows from a subpartition of a composite partitioned table.

When truncating a subpartition, corresponding local index subpartitions are also truncated.

In the following example, the `ALTER TABLE` statement truncates data in subpartitions of a table. In this example, the space occupied by the deleted rows is made available for use by other schema objects in the tablespace with the `DROP STORAGE` clause. Note that the data is truncated, but the subpartitions are not dropped.

Example 4-40 Truncating Multiple Subpartitions

```
CREATE TABLE sales_partition_truncate
  ( product_id       NUMBER(6) NOT NULL,
    customer_id      NUMBER    NOT NULL,
    channel_id       CHAR(1),
    promo_id         NUMBER(6),
    sales_date       DATE,
    quantity_sold    INTEGER,
    amount_sold      NUMBER(10,2)
```

```
  )
 PARTITION BY RANGE (sales_date)
 SUBPARTITION BY LIST (channel_id)
(  PARTITION q3_2018 VALUES LESS THAN (TO_DATE('1-OCT-2018','DD-MON-YYYY'))
   ( SUBPARTITION q3_2018_p_catalog VALUES ('C'),
     SUBPARTITION q3_2018_p_internet VALUES ('I'),
     SUBPARTITION q3_2018_p_partners VALUES ('P'),
     SUBPARTITION q3_2018_p_direct_sales VALUES ('S'),
     SUBPARTITION q3_2018_p_tele_sales VALUES ('T')
   ),
  PARTITION q4_2018 VALUES LESS THAN (TO_DATE('1-JAN-2019','DD-MON-YYYY'))
   ( SUBPARTITION q4_2018_p_catalog VALUES ('C'),
     SUBPARTITION q4_2018_p_internet VALUES ('I'),
     SUBPARTITION q4_2018_p_partners VALUES ('P'),
     SUBPARTITION q4_2018_p_direct_sales VALUES ('S'),
     SUBPARTITION q4_2018_p_tele_sales VALUES ('T')
   ),
  PARTITION q1_2019 VALUES LESS THAN (TO_DATE('1-APR-2019','DD-MON-YYYY'))
   ( SUBPARTITION q1_2019_p_catalog VALUES ('C')
   , SUBPARTITION q1_2019_p_internet VALUES ('I')
   , SUBPARTITION q1_2019_p_partners VALUES ('P')
   , SUBPARTITION q1_2019_p_direct_sales VALUES ('S')
   , SUBPARTITION q1_2019_p_tele_sales VALUES ('T')
   ),
  PARTITION q2_2019 VALUES LESS THAN (TO_DATE('1-JUL-2019','DD-MON-YYYY'))
   ( SUBPARTITION q2_2019_p_catalog VALUES ('C'),
     SUBPARTITION q2_2019_p_internet VALUES ('I'),
     SUBPARTITION q2_2019_p_partners VALUES ('P'),
     SUBPARTITION q2_2019_p_direct_sales VALUES ('S'),
     SUBPARTITION q2_2019_p_tele_sales VALUES ('T')
   ),
  PARTITION q3_2019 VALUES LESS THAN (TO_DATE('1-OCT-2019','DD-MON-YYYY'))
   ( SUBPARTITION q3_2019_p_catalog VALUES ('C'),
     SUBPARTITION q3_2019_p_internet VALUES ('I'),
     SUBPARTITION q3_2019_p_partners VALUES ('P'),
     SUBPARTITION q3_2019_p_direct_sales VALUES ('S'),
     SUBPARTITION q3_2019_p_tele_sales VALUES ('T')
   ),
  PARTITION q4_2019 VALUES LESS THAN (TO_DATE('1-JAN-2020','DD-MON-YYYY'))
   ( SUBPARTITION q4_2019_p_catalog VALUES ('C'),
     SUBPARTITION q4_2019_p_internet VALUES ('I'),
     SUBPARTITION q4_2019_p_partners VALUES ('P'),
     SUBPARTITION q4_2019_p_direct_sales VALUES ('S'),
     SUBPARTITION q4_2019_p_tele_sales VALUES ('T')
   )
  );

SELECT TABLE_NAME, PARTITION_NAME, SUBPARTITION_NAME FROM USER_TAB_SUBPARTITIONS
       WHERE TABLE_NAME ='SALES_PARTITION_TRUNCATE';
TABLE_NAME                 PARTITION_NAME    SUBPARTITION_NAME
-------------------------  ---------------   -------------------
SALES_PARTITION_TRUNCATE    Q1_2019           Q1_2019_P_CATALOG
SALES_PARTITION_TRUNCATE    Q1_2019           Q1_2019_P_DIRECT_SALES
...
30 rows selected.

INSERT INTO sales_partition_truncate VALUES (1001,100,'C',150,'10-SEP-2018',500,2000);
INSERT INTO sales_partition_truncate VALUES (1021,200,'C',160,'16-NOV-2018',100,1500);
INSERT INTO sales_partition_truncate VALUES (1001,100,'C',150,'10-FEB-2019',500,2000);
INSERT INTO sales_partition_truncate VALUES (1021,200,'S',160,'16-FEB-2019',100,1500);
INSERT INTO sales_partition_truncate VALUES (1002,110,'I',180,'15-JUN-2019',100,1000);
INSERT INTO sales_partition_truncate VALUES (5010,150,'P',200,'20-AUG-2019',1000,10000);
```

```
INSERT INTO sales_partition_truncate VALUES (1001,100,'T',150,'12-OCT-2019',500,2000);

SELECT * FROM sales_partition_truncate;
PRODUCT_ID CUSTOMER_ID C   PROMO_ID SALES_DAT QUANTITY_SOLD AMOUNT_SOLD
---------- ----------- - ---------- --------- ------------- -----------
      1001         100 C        150 10-SEP-18           500        2000
      1021         200 C        160 16-NOV-18           100        1500
      1001         100 C        150 10-FEB-19           500        2000
      1021         200 S        160 16-FEB-19           100        1500
      1002         110 I        180 15-JUN-19           100        1000
      5010         150 P        200 20-AUG-19          1000       10000
      1001         100 T        150 12-OCT-19           500        2000
7 rows selected.

ALTER TABLE sales_subpartition_truncate
   TRUNCATE SUBPARTITIONS q3_2018_p_catalog, q4_2018_p_catalog, q1_2019_p_catalog,
                          q2_2019_p_catalog, q3_2019_p_catalog, q4_2019_p_catalog
        DROP STORAGE;

SELECT * FROM sales_partition_truncate;
PRODUCT_ID CUSTOMER_ID C   PROMO_ID SALES_DAT QUANTITY_SOLD AMOUNT_SOLD
---------- ----------- - ---------- --------- ------------- -----------
      1021         200 S        160 16-FEB-19           100        1500
      1002         110 I        180 15-JUN-19           100        1000
      5010         150 P        200 20-AUG-19          1000       10000
      1001         100 T        150 12-OCT-19           500        2000
4 rows selected.

SELECT TABLE_NAME, PARTITION_NAME, SUBPARTITION_NAME FROM USER_TAB_SUBPARTITIONS
       WHERE TABLE_NAME ='SALES_PARTITION_TRUNCATE';
TABLE_NAME                PARTITION_NAME    SUBPARTITION_NAME
------------------------  ---------------   -------------------
SALES_PARTITION_TRUNCATE    Q1_2019           Q1_2019_P_CATALOG
SALES_PARTITION_TRUNCATE    Q1_2019           Q1_2019_P_DIRECT_SALES
...
30 rows selected.
```

4.4.14.4 Truncating a Partition with the Cascade Option

You can use cascade truncate operations to a reference partitioned child table with the `CASCADE` option of `TRUNCATE TABLE`, `ALTER TABLE TRUNCATE PARTITION`, and `ALTER TABLE TRUNCATE SUBPARTITION` SQL statements.

When the `CASCADE` option is specified for `TRUNCATE TABLE`, the truncate table operation also truncates child tables that reference the targeted table through an enabled referential constraint that has `ON DELETE CASCADE` enabled. This cascading action applies recursively to grandchildren, great-grandchildren, and so on. After determining the set of tables to be truncated based on the enabled `ON DELETE CASCADE` referential constraints, an error is raised if any table in this set is referenced through an enabled constraint from a child outside of the set. If a parent and child are connected by multiple referential constraints, a `TRUNCATE TABLE CASCADE` operation targeting the parent succeeds if at least one constraint has `ON DELETE CASCADE` enabled.

Privileges are required on all tables affected by the operation. Any other options specified for the operation, such as `DROP STORAGE` or `PURGE MATERIALIZED VIEW LOG`, apply for all tables affected by the operation.

When the `CASCADE` option is specified, the `TRUNCATE PARTITION` and `TRUNCATE SUBPARTITION` operations cascade to reference partitioned tables that are children of the targeted table. The `TRUNCATE` can be targeted at any level in a reference partitioned hierarchy and cascades to

child tables starting from the targeted table. Privileges are not required on the child tables, but the usual restrictions on the TRUNCATE operation, such as the table cannot be referenced by an enabled referential constraint that is not a partitioning constraint, apply for all tables affected by the operation.

The CASCADE option is ignored if it is specified for a table that does not have reference partitioned children. Any other options specified for the operation, such as DROP STORAGE or UPDATE INDEXES, apply to all tables affected by the operation.

The cascade options are off by default so they do not affect Oracle Database compatibility.

```
ALTER TABLE sales
     TRUNCATE PARTITION dec2016
     DROP STORAGE
     CASCADE
     UPDATE INDEXES;
```

4.5 About Dropping Partitioned Tables

Dropping partitioned tables is similar to dropping nonpartitioned tables.

Oracle Database processes a DROP TABLE statement for a partitioned table in the same way that it processes the statement for a nonpartitioned table. One exception is when you use the PURGE keyword.

To avoid running into resource constraints, the DROP TABLE...PURGE statement for a partitioned table drops the table in multiple transactions, where each transaction drops a subset of the partitions or subpartitions and then commits. The table is dropped at the conclusion of the final transaction.

This behavior comes with some changes to the DROP TABLE statement. First, if the DROP TABLE...PURGE statement fails, then you can take corrective action, if any, and then reissue the statement. The statement resumes at the point where it failed. Second, while the DROP TABLE...PURGE statement is in progress, the table is marked as unusable by setting the STATUS column to the value UNUSABLE in the following data dictionary views:

- USER_TABLES, ALL_TABLES, DBA_TABLES
- USER_PART_TABLES, ALL_PART_TABLES, DBA_PART_TABLES
- USER_OBJECT_TABLES, ALL_OBJECT_TABLES, DBA_OBJECT_TABLES

You can list all UNUSABLE partitioned tables by querying the STATUS column of these views.

Queries against other data dictionary views pertaining to partitioning, such as DBA_TAB_PARTITIONS and DBA_TAB_SUBPARTITIONS, exclude rows belonging to an UNUSABLE table.

After a table is marked UNUSABLE, the only statement that can be issued against it is another DROP TABLE...PURGE statement, and only if the previous DROP TABLE...PURGE statement failed. Any other statement issued against an UNUSABLE table results in an error. The table remains in the UNUSABLE state until the drop operation is complete.

> **See Also:**
>
> - Viewing Information About Partitioned Tables and Indexes for a list of these views that contain information related to partitioning
> - *Oracle Database SQL Language Reference* for the syntax of the `DROP TABLE` statement
> - *Oracle Database Reference* for a description of the data dictionary views mentioned in this section

4.6 Changing a Nonpartitioned Table into a Partitioned Table

You can change a nonpartitioned table into a partitioned table.

The following topics are discussed:

- Using Online Redefinition to Partition Collection Tables
- Converting a Non-Partitioned Table to a Partitioned Table

> **See Also:**
>
> *Oracle Database Administrator's Guide* for information about redefining partitions of a table

4.6.1 Using Online Redefinition to Partition Collection Tables

Oracle Database provides a mechanism to move one or more partitions or to make other changes to the partitions' physical structures without significantly affecting the availability of the partitions for DML. This mechanism is called online table redefinition.

You can use online redefinition to copy nonpartitioned Collection Tables to partitioned Collection Tables and Oracle Database inserts rows into the appropriate partitions in the Collection Table. Example 4-41 illustrates how this is done for nested tables inside an Objects column; a similar example works for Ordered Collection Type Tables inside an `XMLType` table or column. During the `copy_table_dependents` operation, you specify `0` or `false` for copying the indexes and constraints, because you want to keep the indexes and constraints of the newly defined collection table. However, the Collection Tables and its partitions have the same names as that of the interim table (`print_media2` in Example 4-41). You must take explicit steps to preserve the Collection Table names.

Example 4-41 Redefining partitions with collection tables

```
REM Connect as a user with appropriate privileges, then run the following
DROP USER eqnt CASCADE;
CREATE USER eqnt IDENTIFIED BY eqnt;
GRANT CONNECT, RESOURCE TO eqnt;

-- Grant privleges required for online redefinition.
GRANT EXECUTE ON DBMS_REDEFINITION TO eqnt;
GRANT ALTER ANY TABLE TO eqnt;
GRANT DROP ANY TABLE TO eqnt;
```

```
GRANT LOCK ANY TABLE TO eqnt;
GRANT CREATE ANY TABLE TO eqnt;
GRANT SELECT ANY TABLE TO eqnt;

-- Privileges required to perform cloning of dependent objects.
GRANT CREATE ANY TRIGGER TO eqnt;
GRANT CREATE ANY INDEX TO eqnt;

CONNECT eqnt/eqnt

CREATE TYPE textdoc_typ AS OBJECT ( document_typ VARCHAR2(32));
/
CREATE TYPE textdoc_tab AS TABLE OF textdoc_typ;
/

-- (old) non partitioned nested table
CREATE TABLE print_media
    ( product_id        NUMBER(6) primary key
    , ad_textdocs_ntab  textdoc_tab
    )
NESTED TABLE ad_textdocs_ntab STORE AS equi_nestedtab
(   (document_typ NOT NULL)
    STORAGE (INITIAL 8M)
)

;

-- Insert into base table
INSERT INTO print_media VALUES (1,
   textdoc_tab(textdoc_typ('xx'), textdoc_typ('yy')));
INSERT INTO print_media VALUES (11,
   textdoc_tab(textdoc_typ('aa'), textdoc_typ('bb')));
COMMIT;

-- Insert into nested table
INSERT INTO TABLE
  (SELECT p.ad_textdocs_ntab FROM print_media p WHERE p.product_id = 11)
   VALUES ('cc');

SELECT * FROM print_media;

PRODUCT_ID   AD_TEXTDOCS_NTAB(DOCUMENT_TYP)
----------   ------------------------------
         1   TEXTDOC_TAB(TEXTDOC_TYP('xx'), TEXTDOC_TYP('yy'))
        11   TEXTDOC_TAB(TEXTDOC_TYP('aa'), TEXTDOC_TYP('bb'), TEXTDOC_TYP('cc'))

-- Creating partitioned Interim Table
CREATE TABLE print_media2
    ( product_id        NUMBER(6)
    , ad_textdocs_ntab  textdoc_tab
    )
NESTED TABLE ad_textdocs_ntab STORE AS equi_nestedtab2
(   (document_typ NOT NULL)
    STORAGE (INITIAL 8M)
)
PARTITION BY RANGE (product_id)
(
    PARTITION P1 VALUES LESS THAN (10),
    PARTITION P2 VALUES LESS THAN (20)
);

EXEC dbms_redefinition.start_redef_table('eqnt', 'print_media', 'print_media2');
```

```
DECLARE
 error_count pls_integer := 0;
BEGIN
  dbms_redefinition.copy_table_dependents('eqnt', 'print_media', 'print_media2',
                                          0, true, false, true, false,
                                          error_count);

  dbms_output.put_line('errors := ' || to_char(error_count));
END;
/

EXEC  dbms_redefinition.finish_redef_table('eqnt', 'print_media', 'print_media2');

-- Drop the interim table
DROP TABLE print_media2;

-- print_media has partitioned nested table here

SELECT * FROM print_media PARTITION (p1);

PRODUCT_ID   AD_TEXTDOCS_NTAB(DOCUMENT_TYP)
----------   ------------------------------
         1   TEXTDOC_TAB(TEXTDOC_TYP('xx'), TEXTDOC_TYP('yy'))

SELECT * FROM print_media PARTITION (p2);

PRODUCT_ID   AD_TEXTDOCS_NTAB(DOCUMENT_TYP)
----------   ------------------------------
        11   TEXTDOC_TAB(TEXTDOC_TYP('aa'), TEXTDOC_TYP('bb'), TEXTDOC_TYP('cc'))
```

4.6.2 Converting a Non-Partitioned Table to a Partitioned Table

A non-partitioned table can be converted to a partitioned table with a `MODIFY` clause added to the `ALTER TABLE` SQL statement.

In addition, the keyword `ONLINE` can be specified, enabling concurrent DML operations while the conversion is ongoing.

The following is an example of the `ALTER TABLE` statement using the `ONLINE` keyword for an online conversion to a partitioned table.

Example 4-42 Using the MODIFY clause of ALTER TABLE to convert online to a partitioned table

```
ALTER TABLE employees_convert MODIFY
  PARTITION BY RANGE (employee_id) INTERVAL (100)
  ( PARTITION P1 VALUES LESS THAN (100),
    PARTITION P2 VALUES LESS THAN (500)
   ) ONLINE
  UPDATE INDEXES
 ( IDX1_SALARY LOCAL,
   IDX2_EMP_ID GLOBAL PARTITION BY RANGE (employee_id)
  ( PARTITION IP1 VALUES LESS THAN (MAXVALUE))
 );
```

Considerations When Using the UPDATE INDEXES Clause

When using the `UPDATE INDEXES` clause, note the following.

- This clause can be used to change the partitioning state of indexes and storage properties of the indexes being converted.
- The specification of the `UPDATE INDEXES` clause is optional.

  Indexes are maintained both for the online and offline conversion to a partitioned table.
- This clause cannot change the columns on which the original list of indexes are defined.
- This clause cannot change the uniqueness property of the index or any other index property.
- If you do not specify the tablespace for any of the indexes, then the following tablespace defaults apply.
  - Local indexes after the conversion collocate with the table partition.
  - Global indexes after the conversion reside in the same tablespace of the original global index on the non-partitioned table.
- If you do not specify the `INDEXES` clause or the `INDEXES` clause does not specify all the indexes on the original non-partitioned table, then the following default behavior applies for all unspecified indexes.
  - Global partitioned indexes remain the same and retain the original partitioning shape.
  - Non-prefixed indexes become global nonpartitioned indexes.
  - Prefixed indexes are converted to local partitioned indexes.

    Prefixed means that the partition key columns are included in the index definition, but the index definition is not limited to including the partitioning keys only.
  - Bitmap indexes become local partitioned indexes, regardless whether they are prefixed or not.

    Bitmap indexes must always be local partitioned indexes.
- The conversion operation cannot be performed if there are domain indexes.

4.7 Managing Hybrid Partitioned Tables

The following topics are discussed in this section:

- Creating Hybrid Partitioned Tables
- Converting to Hybrid Partitioned Tables
- Converting Hybrid Partitioned Tables to Internal Partitioned Tables
- Using ADO With Hybrid Partitioned Tables
- Splitting Partitions in a Hybrid Partitioned Table

> **See Also:**
>
> - Hybrid Partitioned Tables for an overview of hybrid partitioned tables, including information about limitations

4.7.1 Creating Hybrid Partitioned Tables

You can use the `EXTERNAL PARTITION ATTRIBUTES` clause of the `CREATE TABLE` statement to determine hybrid partitioning for a table. The partitions of the table can be external and or internal.

> **Note:**
>
> As of Oracle Database 23ai, single-level partitioning with interval, autolist, range and list partitioning are supported for hybrid partitioned tables.

A hybrid partitioned table enables partitions to reside both in database data files (internal partitions) and in external files and sources (external partitions). You can create and query a hybrid partitioned table to utilize the benefits of partitioning with classic partitioned tables, such as pruning, on data that is contained in both internal and external partitions.

The `EXTERNAL PARTITION ATTRIBUTES` clause of the `CREATE TABLE` statement is defined at the table level for specifying table level external parameters in the hybrid partitioned table, such as:

- The access driver type, such as `ORACLE_LOADER, ORACLE_DATAPUMP, ORACLE_HDFS, ORACLE_HIVE, ORACLE_BIGDATA.`

  > **Note:**
  >
  > `ORACLE_HIVE` does not support interval and autolist partitioning.

- The default directory for all external partitions files
- The access parameters

The `EXTERNAL` clause of the `PARTITION` clause defines the partition as an external partition. When there is no `EXTERNAL` clause, the partition is an internal partition. You can specify a default location for each external partition different then the default directory defined at the table level. For example in Creating a Hybrid Range-Partitioned Table, the `DEFAULT DIRECTORY` value for partitions `sales_data2`, `sales_data3`, and `sales_data_acfs` is different than the `DEFAULT DIRECTORY` value defined in the `EXTERNAL PARTITION ATTRIBUTES` clause.

When there is no external file defined for an external partition, the external partition is empty. It can be populated with an external file by using an `ALTER TABLE MODIFY PARTITION` statement. Note that at least one partition must be an internal partition.

> **See Also:**
>
> - Hybrid Partitioned Tables

4.7.1.1 Creating a Hybrid Range-Partitioned Table

In the first example below, a hybrid range-partitioned table is a created with four external partitions and two internal partitions. The external CSV data files are stored in the `sales_data` ,

`sales_data2`, `sales_data3`, and `sales_data_acfs` directories defined by the `DEFAULT DIRECTORY` clauses. `sales_data` is defined as the overall `DEFAULT DIRECTORY` in the `EXTERNAL PARTITION ATTRIBUTES` clause. The other directories are defined at the partition level. `sales_2014` and `sales_2015` are internal partitions. Data directory `sales_data_acfs` is stored on an Oracle ACFS file system to illustrate the use of that storage option.

In the second example, an additional external partition is added to the table.

Example 4-43 Creating a Hybrid Range-Partitioned Table

```
REM Connect as a user with appropriate privileges,
REM then run the following to set up data directories that contain the data files
CREATE DIRECTORY sales_data AS '/u01/my_data/sales_data1';
GRANT READ,WRITE ON DIRECTORY sales_data TO hr;

CREATE DIRECTORY sales_data2 AS '/u01/my_data/sales_data2';
GRANT READ,WRITE ON DIRECTORY sales_data2 TO hr;

CREATE DIRECTORY sales_data3 AS '/u01/my_data/sales_data3';
GRANT READ,WRITE ON DIRECTORY sales_data3 TO hr;

REM set up a data directory on an Oracle ACFS mount point (file system)
CREATE DIRECTORY sales_data_acfs AS '/u01/acfsmounts/acfs1';
GRANT READ,WRITE ON DIRECTORY sales_data_acfs TO hr;

CONNECT AS hr, run the following
CREATE TABLE hybrid_partition_table
  ( prod_id       NUMBER        NOT NULL,
    cust_id       NUMBER        NOT NULL,
    time_id       DATE          NOT NULL,
    channel_id    NUMBER        NOT NULL,
    promo_id      NUMBER        NOT NULL,
    quantity_sold NUMBER(10,2)  NOT NULL,
    amount_sold   NUMBER(10,2)  NOT NULL
  )
    EXTERNAL PARTITION ATTRIBUTES (
      TYPE ORACLE_LOADER
      DEFAULT DIRECTORY sales_data
       ACCESS PARAMETERS(
         FIELDS TERMINATED BY ','
         (prod_id,cust_id,time_id DATE 'dd-mm-yyyy',channel_id,promo_id,quantity_sold,amount_sold)
       )
      REJECT LIMIT UNLIMITED
     )
    PARTITION BY RANGE (time_id)
    (PARTITION sales_2014 VALUES LESS THAN (TO_DATE('01-01-2015','dd-mm-yyyy')),
     PARTITION sales_2015 VALUES LESS THAN (TO_DATE('01-01-2016','dd-mm-yyyy')),
     PARTITION sales_2016 VALUES LESS THAN (TO_DATE('01-01-2017','dd-mm-yyyy')) EXTERNAL
          LOCATION ('sales2016_data.txt'),
     PARTITION sales_2017 VALUES LESS THAN (TO_DATE('01-01-2018','dd-mm-yyyy')) EXTERNAL
          DEFAULT DIRECTORY sales_data2 LOCATION ('sales2017_data.txt'),
     PARTITION sales_2018 VALUES LESS THAN (TO_DATE('01-01-2019','dd-mm-yyyy')) EXTERNAL
          DEFAULT DIRECTORY sales_data3 LOCATION ('sales2018_data.txt'),
     PARTITION sales_2019 VALUES LESS THAN (TO_DATE('01-01-2020','dd-mm-yyyy')) EXTERNAL
          DEFAULT DIRECTORY sales_data_acfs LOCATION ('sales2019_data.txt')
 );
```

Example 4-44 Adding an External Partition to a Hybrid Range-Partitioned Table

```
ALTER TABLE hybrid_partition_table
      ADD PARTITION sales_2020 VALUES LESS THAN (TO_DATE('01-01-2021','dd-mm-yyyy'))
          EXTERNAL DEFAULT DIRECTORY sales_data_acfs  LOCATION ('sales2020_data.txt');
```

4.7.1.2 Creating a Hybrid Partitioned Table Containing Interval Partitions

The table created below has a top level `INTERVAL` clause indicating that the table is interval partitioned. The create table statement specifies four partitions with varying widths. It also specifies that above the transition point of `1-Jan-2014` partitions are created with a width of one month. The high bound of partition `P3` represents a transition point. `P3` and all partitions below it (`P0`, `P1`, and `P2` in this example) are in the range section while all partitions above it fall into the interval section

```
CREATE TABLE SALES_INTERVAL
( prod_id NUMBER NOT NULL,
    cust_id NUMBER NOT NULL,
    time_id DATE NOT NULL,
    channel_id NUMBER NOT NULL,
    promo_id NUMBER NOT NULL,
    quantity_sold NUMBER(10,2) NOT NULL,
    amount_sold NUMBER(10,2) NOT NULL
  )
   EXTERNAL PARTITION ATTRIBUTES( TYPE ORACLE_LOADER
       DEFAULT DIRECTORY Data_pump_dir
       ACCESS PARAMETERS
       (
         RECORDS DELIMITED BY NEWLINE
         BADFILE 'saleshyt_%a_%p.bad'
         LOGFILE 'saleshyt_%a_%p.log'
         FIELDS TERMINATED BY ','
         MISSING FIELD VALUES ARE NULL
        )
        REJECT LIMIT UNLIMITED
                                    )
PARTITION BY RANGE (time_id)
INTERVAL(NUMTOYMINTERVAL(1,'month'))
(PARTITION P0 values less than (TO_DATE('01-01-2012', 'dd-mm-yyyy')) EXTERNAL
LOCATION('sales_interval_hypt1.csv'),
PARTITION P1 values less than (TO_DATE('01-01-2013', 'dd-mm-yyyy')) EXTERNAL
LOCATION('sales_interval_hypt1.csv'),
PARTITION P2 values less than (TO_DATE('01-07-2013', 'dd-mm-yyyy')),
PARTITION P3 values less than (TO_DATE('01-01-2014', 'dd-mm-yyyy')));
```

Note that when a row with the `time_id` of `01-01-2014` is inserted, a partition `SYS_P` is automatically created with a high bound of `01-02-2014`, and when a row for `07-01-2020` is subsequently added, another partition `SYS_P` with a high bound of `08-01-2020` is added. The high bound of the automatically created partitions in this example is one month above the preceding high value, as specified in the interval clause of the hybrid partitioned table.

A hybrid partitioned table with interval partitioning allows both internal and external partitions in the range section. In the example, `P1` is an external partition in the table's range section, and internal partition `P3` is the transition point. When a row corresponding containing a `time_id` of `01-01-2014` is inserted into the table, internal partition `Pi0` is created in the interval section and a row is inserted into the partition. Note that a partition is required at the end of the range section to specify the transition point. An interval hybrid partitioned table must contain at least one internal partition.

4.7.1.3 Creating a Hybrid Partitioned Table With Automatic List Partitioning

As of Oracle Database 23ai, Oracle supports automatic list partitioning on hybrid partitioned tables

The following example describes the syntax for creating a hybrid partitioned table with automatic list partitioning. The table has an external partition `P_NY` containing a list of values for New York, in addition to the internal partition `P_CAL` for California. When an inserted row does not map to one of these existing partitions and a new partition is needed (for New Jersey, for example), the new internal partition is automatically created in the table.

```
CREATE TABLE SALES
(
    SALESPERSON_ID  NUMBER(5),
    SALESPERSON_NAME VARCHAR2(30),
    SALES_STATE   VARCHAR2(20),
    SALES_AMOUNT  NUMBER(10),
    SALES_DATE  DATE
)
EXTERNAL PARTITION ATTRIBUTES(
      TYPE ORACLE_LOADER
       DEFAULT DIRECTORY DEF_DIR1
       ACCESS PARAMETERS
        (
           RECORDS DELIMITED BY NEWLINE
            BADFILE 'saleshyt_%a_%p.bad'
             LOGFILE 'saleshyt_%a_%p.log'
             FIELDS TERMINATED BY ','
             MISSING FIELD VALUES ARE NULL
          )
          REJECT LIMIT UNLIMITED
 )
PARTITION BY LIST(SALES_STATE) AUTOMATIC
(PARTITION P_CAL VALUES ('CALIFORNIA'),
 PARTITION P_NY   VALUES (‘NEW YORK’) EXTERNAL LOCATION(‘p_ny.csv’));
```

4.7.2 Converting to Hybrid Partitioned Tables

You can convert a table with only internal partitions to a hybrid partitioned table.

In Example 4-45, an internal range partitioned table is converted to a hybrid partitioned table. You must add external partition attributes to an existing table first, then add external partitions. Note that at least one partition must be an internal partition.

Example 4-45 Converting to a Hybrid Range-Partitioned Table

```
CREATE TABLE internal_to_hypt_table (
    prod_id       NUMBER        NOT NULL,
    cust_id       NUMBER        NOT NULL,
    time_id       DATE          NOT NULL,
    channel_id    NUMBER        NOT NULL,
    promo_id      NUMBER        NOT NULL,
    quantity_sold NUMBER(10,2)  NOT NULL,
    amount_sold   NUMBER(10,2)  NOT NULL
  )
   PARTITION by range (time_id)
```

```
   (PARTITION sales_2014 VALUES LESS THAN (TO_DATE('01-01-2015','dd-mm-yyyy'))
  );

SELECT HYBRID FROM USER_TABLES WHERE TABLE_NAME = 'INTERNAL_TO_HYPT_TABLE';
HYB
---
NO

ALTER TABLE internal_to_hypt_table
  ADD EXTERNAL PARTITION ATTRIBUTES
   (TYPE ORACLE_LOADER
     DEFAULT DIRECTORY sales_data
     ACCESS PARAMETERS (
       FIELDS TERMINATED BY ','
       (prod_id,cust_id,time_id DATE 'dd-mm-yyyy',channel_id,promo_id,quantity_sold,amount_sold)
     )
   )
;

ALTER TABLE internal_to_hypt_table
      ADD PARTITION sales_2015 VALUES LESS THAN (TO_DATE('01-01-2016','dd-mm-yyyy'))
          EXTERNAL LOCATION ('sales2015_data.txt');

ALTER TABLE internal_to_hypt_table
      ADD PARTITION sales_2016 VALUES LESS THAN (TO_DATE('01-01-2017','dd-mm-yyyy'))
          EXTERNAL LOCATION ('sales2016_data.txt');

SELECT HYBRID FROM USER_TABLES WHERE TABLE_NAME = 'INTERNAL_TO_HYPT_TABLE';
HYB
---
YES

SELECT DEFAULT_DIRECTORY_NAME FROM USER_EXTERNAL_TABLES WHERE TABLE_NAME = 'INTERNAL_TO_HYPT_TABLE';
DEFAULT_DIRECTORY_NAME
--------------------------------------------------------------------------------
SALES_DATA
```

See Also:

- Hybrid Partitioned Tables

4.7.3 Converting Hybrid Partitioned Tables to Internal Partitioned Tables

You can convert a hybrid partitioned table to a table with only internal partitions.

In Example 4-46, a hybrid partitioned table is converted to an internal range partitioned table. First, you must drop all external partitions and then you can drop the external partition attributes.

Example 4-46 Converting from a Hybrid Partitioned Table to an Internal Table

```
CREATE TABLE hypt_to_int_table
  ( prod_id       NUMBER        NOT NULL,
    cust_id       NUMBER        NOT NULL,
    time_id       DATE          NOT NULL,
    channel_id    NUMBER        NOT NULL,
    promo_id      NUMBER        NOT NULL,
    quantity_sold NUMBER(10,2)  NOT NULL,
```

```
    amount_sold   NUMBER(10,2)  NOT NULL
  )
    EXTERNAL PARTITION ATTRIBUTES (
     TYPE ORACLE_LOADER
     DEFAULT DIRECTORY sales_data
       ACCESS PARAMETERS(
        FIELDS TERMINATED BY ','
        (prod_id,cust_id,time_id DATE 'dd-mm-yyyy',channel_id,promo_id,quantity_sold,amount_sold)
       )
      REJECT LIMIT UNLIMITED
     )
    PARTITION BY RANGE (time_id)
    (PARTITION sales_2014 VALUES LESS THAN (TO_DATE('01-01-2015','dd-mm-yyyy')),
     PARTITION sales_2015 VALUES LESS THAN (TO_DATE('01-01-2016','dd-mm-yyyy')),
     PARTITION sales_2016 VALUES LESS THAN (TO_DATE('01-01-2017','dd-mm-yyyy'))
          EXTERNAL LOCATION ('sales2016_data.txt'),
     PARTITION sales_2017 VALUES LESS THAN (TO_DATE('01-01-2018','dd-mm-yyyy'))
          EXTERNAL DEFAULT DIRECTORY sales_data2 LOCATION ('sales2017_data.txt'),
     PARTITION sales_2018 VALUES LESS THAN (TO_DATE('01-01-2019','dd-mm-yyyy'))
          EXTERNAL DEFAULT DIRECTORY sales_data3 LOCATION ('sales2018_data.txt'),
     PARTITION sales_2019 VALUES LESS THAN (TO_DATE('01-01-2020','dd-mm-yyyy'))
          EXTERNAL DEFAULT DIRECTORY sales_data_acfs LOCATION ('sales2019_data.txt')
 );

SELECT HYBRID FROM USER_TABLES WHERE TABLE_NAME = 'HYPT_TO_INT_TABLE';
HYB
---
YES

ALTER TABLE hypt_to_int_table DROP PARTITION sales_2016;
ALTER TABLE hypt_to_int_table DROP PARTITION sales_2017;
ALTER TABLE hypt_to_int_table DROP PARTITION sales_2018;
ALTER TABLE hypt_to_int_table DROP PARTITION sales_2019;

ALTER TABLE hypt_to_int_table DROP EXTERNAL PARTITION ATTRIBUTES();

SELECT HYBRID FROM USER_TABLES WHERE TABLE_NAME = 'HYPT_TO_INT_TABLE';
HYB
---
NO
```

See Also:

- Hybrid Partitioned Tables

4.7.4 Using ADO With Hybrid Partitioned Tables

You can use Automatic Data Optimization (ADO) policies with hybrid partitioned tables under some conditions.

In Example 4-47, note that ADO policies are only defined on the internal partitions of the table.

Example 4-47 Using ADO with a Hybrid Partitioned Table

```
SQL> CREATE TABLE hypt_ado_table
  ( prod_id       NUMBER        NOT NULL,
    cust_id       NUMBER        NOT NULL,
    time_id       DATE          NOT NULL,
```

```
    channel_id    NUMBER        NOT NULL,
    promo_id      NUMBER        NOT NULL,
    quantity_sold NUMBER(10,2)  NOT NULL,
    amount_sold   NUMBER(10,2)  NOT NULL
  )
    EXTERNAL PARTITION ATTRIBUTES (
     TYPE ORACLE_LOADER
     DEFAULT DIRECTORY sales_data
       ACCESS PARAMETERS(
        FIELDS TERMINATED BY ','
        (prod_id,cust_id,time_id DATE 'dd-mm-yyyy',channel_id,promo_id,quantity_sold,amount_sold)
       )
      REJECT LIMIT UNLIMITED
     )
    PARTITION BY RANGE (time_id)
    (PARTITION sales_2014 VALUES LESS THAN (TO_DATE('01-01-2015','dd-mm-yyyy')),
     PARTITION sales_2015 VALUES LESS THAN (TO_DATE('01-01-2016','dd-mm-yyyy')),
     PARTITION sales_2016 VALUES LESS THAN (TO_DATE('01-01-2017','dd-mm-yyyy'))
          EXTERNAL LOCATION ('sales2016_data.txt'),
     PARTITION sales_2017 VALUES LESS THAN (TO_DATE('01-01-2018','dd-mm-yyyy'))
          EXTERNAL DEFAULT DIRECTORY sales_data2 LOCATION ('sales2017_data.txt'),
     PARTITION sales_2018 VALUES LESS THAN (TO_DATE('01-01-2019','dd-mm-yyyy'))
          EXTERNAL DEFAULT DIRECTORY sales_data3 LOCATION ('sales2018_data.txt'),
     PARTITION sales_2019 VALUES LESS THAN (TO_DATE('01-01-2020','dd-mm-yyyy'))
          EXTERNAL DEFAULT DIRECTORY sales_data4 LOCATION ('sales2019_data.txt')
 );
Table created.

SQL> SELECT HYBRID FROM USER_TABLES WHERE TABLE_NAME = 'HYPT_ADO_TABLE';
HYB
---
YES

SQL> ALTER TABLE hypt_ado_table MODIFY PARTITION sales_2014 ILM ADD POLICY ROW STORE COMPRESS ADVANCED
ROW AFTER 6 MONTHS OF NO MODIFICATION;
Table altered.

SQL> ALTER TABLE hypt_ado_table MODIFY PARTITION sales_2015 ILM ADD POLICY ROW STORE COMPRESS ADVANCED
ROW AFTER 6 MONTHS OF NO MODIFICATION;
Table altered.

SQL> SELECT POLICY_NAME, POLICY_TYPE, ENABLED FROM USER_ILMPOLICIES;
POLICY_NAME    POLICY_TYPE     ENA
-------------  --------------- -----
P1             DATA MOVEMENT   YES
P2             DATA MOVEMENT   YES
```

See Also:

- Using Automatic Data Optimization for information about ADO policies

4.7.5 Splitting Partitions in a Hybrid Partitioned Table

In Example 4-48, the default (MAXVALUE) partition is split into a two partitions: a new partition and the existing default position. You can split a default partition similar to splitting any other partition.

Example 4-48 Splitting the Default Partition in a Hybrid Partitioned Table

4.7.6 Exchanging Data in Hybrid Partitioned Tables

You can exchange data of an internal partition in a hybrid partitioned table with an external nonpartitioned table, and exchange data of external nonpartitioned table with an internal partition in a hybrid partitioned table. Oracle supports exchange between internal and external storage but does not support a move operation between these tiers. Moving data between internal and external storage is a separate operation prior to exchange. Such a data "move" is normally done by physically copying the data to its future location, doing the exchange, and then removing or dropping the data from the old location.

Example 4-49 Exchanging data of an internal partition of a hybrid partitioned table with an external nonpartioned table

In this example, data of an internal partition of an hybrid partitioned table is "moved" to external storage using `EXCHANGE PARTITION` with a nonpartitioned external table containing the exact same data. Older data is moved to external storage for archival purposes.

Create a hybrid partitioned table of TYPE ORACLE_DATAPUMP.

```
CREATE TABLE hybrid_datapump_sales
  ( prod_id  NUMBER     NOT NULL,
    cust_id       NUMBER        NOT NULL,
    time_id       DATE          NOT NULL,
    channel_id    NUMBER        NOT NULL,
    promo_id      NUMBER        NOT NULL,
    quantity_sold NUMBER(10,2)  NOT NULL,
    amount_sold   NUMBER(10,2)  NOT NULL
  )
  EXTERNAL PARTITION ATTRIBUTES
  (TYPE ORACLE_DATAPUMP
   DEFAULT DIRECTORY sales_data
   ACCESS PARAMETERS (NOLOGFILE)
  )
  PARTITION by range (time_id)
  (
   PARTITION sales_2018 VALUES LESS THAN (TO_DATE('01-01-2019','dd-mm-yyyy')),
   PARTITION sales_2019 VALUES LESS THAN (TO_DATE('01-01-2020','dd-mm-yyyy')),
   PARTITION sales_2020 VALUES LESS THAN (TO_DATE('01-01-2021','dd-mm-yyyy')),
   PARTITION sales_future VALUES LESS THAN (MAXVALUE)
);
```

Populate the hybrid partitioned table `hybrid_datapump_sales` with some data for this example.

```
SELECT * FROM hybrid_datapump_sales PARTITION(sales_2018);

   PROD_ID    CUST_ID TIME_ID   CHANNEL_ID   PROMO_ID QUANTITY_SOLD AMOUNT_SOLD
---------- ---------- --------- ---------- ---------- ------------- -----------
      1001        100 10-FEB-18         10         15           500        7500
      1002        110 15-JUN-18         12         18           100        3200
      1002        110 30-MAR-18         10         15           500        6500
      2105        102 21-APR-18         18         12           100        2000
      1200        155 30-APR-18         20         20           300        3600
```

Create an external table with the same structure as the `sales_2018` partition. The `SELECT` clause completes data movement. The data movement operation needs to be done prior to exchange.

```
CREATE TABLE year_2018_datapump
   ORGANIZATION EXTERNAL
   ( TYPE ORACLE_DATAPUMP DEFAULT DIRECTORY sales_data
     ACCESS PARAMETERS (NOLOGFILE) LOCATION ('sales_2018.dmp')
    )
   AS SELECT * FROM hybrid_datapump_sales PARTITION(sales_2018);
```

Exchange the data in `sales_2018` partition with the data in the external table.

```
ALTER TABLE hybrid_datapump_sales
      EXCHANGE PARTITION(sales_2018) WITH TABLE year_2018_datapump;

SELECT * FROM year_2018_datapump;

   PROD_ID    CUST_ID TIME_ID   CHANNEL_ID   PROMO_ID QUANTITY_SOLD AMOUNT_SOLD
---------- ---------- --------- ---------- ---------- ------------- -----------
      1001        100 10-FEB-18         10         15           500        7500
      1002        110 15-JUN-18         12         18           100        3200
      1002        110 30-MAR-18         10         15           500        6500
      2105        102 21-APR-18         18         12           100        2000
      1200        155 30-APR-18         20         20           300        3600
```

Example 4-50 Exchanging data of an external nonpartitioned table with an internal partition of a hybrid partitioned table

In this example, data of an external table is exchanged with an internal partition of a hybrid partitioned table to add new data to a partition of the hybrid partitioned table. The text data that has been loaded into the external table is first copied to a temporary nonpartitioned internal table. Then the nonpartitioned internal table is exchanged with an internal partition of a hybrid partitioned table.

Create a hybrid partitioned table of TYPE ORACLE_DATAPUMP.

```
CREATE TABLE hybrid_datapump_sales
    ( prod_id  NUMBER     NOT NULL,
      cust_id       NUMBER        NOT NULL,
      time_id       DATE          NOT NULL,
      channel_id    NUMBER        NOT NULL,
      promo_id      NUMBER        NOT NULL,
      quantity_sold NUMBER(10,2)  NOT NULL,
      amount_sold   NUMBER(10,2)  NOT NULL
     )
     EXTERNAL PARTITION ATTRIBUTES
      (TYPE ORACLE_DATAPUMP
       DEFAULT DIRECTORY sales_data
       ACCESS PARAMETERS (NOLOGFILE)
      )
     PARTITION by range (time_id)
     (
      PARTITION sales_old VALUES LESS THAN (TO_DATE('01-01-2018', 'dd-mm-yyyy'))
       EXTERNAL LOCATION ('sales_old.dmp'),
      PARTITION sales_2018 VALUES LESS THAN (TO_DATE('01-01-2019','dd-mm-yyyy')),
      PARTITION sales_2019 VALUES LESS THAN (TO_DATE('01-01-2020','dd-mm-yyyy')),
      PARTITION sales_2020 VALUES LESS THAN (TO_DATE('01-01-2021','dd-mm-yyyy')),
      PARTITION sales_future VALUES LESS THAN (MAXVALUE)
  );
```

Note that the `sales_2020` partition has no records.

```
SELECT * FROM hybrid_datapump_sales PARTITION(sales_2020);

no rows selected
```

For this example, create an external table and load it with a text file that has been generated by some application.

```
CREATE TABLE ext_sales_year_2020
        ( prod_id       NUMBER        NOT NULL,
          cust_id       NUMBER        NOT NULL,
          time_id       DATE          NOT NULL,
          channel_id    NUMBER        NOT NULL,
          promo_id      NUMBER        NOT NULL,
          quantity_sold NUMBER(10,2)  NOT NULL,
          amount_sold   NUMBER(10,2)  NOT NULL
         )
        ORGANIZATION EXTERNAL
        (
         TYPE ORACLE_LOADER DEFAULT DIRECTORY sales_data
         ACCESS PARAMETERS (
         FIELDS TERMINATED BY ','
         (prod_id,cust_id,time_id DATE 'dd-mm-
yyyy',channel_id,promo_id,quantity_sold,amount_sold)
         )
         LOCATION ('sales2020_data.txt')
 );

SELECT * FROM ext_sales_year_2020;

   PROD_ID    CUST_ID TIME_ID   CHANNEL_ID   PROMO_ID QUANTITY_SOLD AMOUNT_SOLD
---------- ---------- --------- ---------- ---------- ------------- -----------
      1001        100 10-JAN-20         10         15           500        7500
      1002        110 15-JAN-20         12         18           100        3200
      1001        100 20-JAN-20         10         15           500        7500
      2105        101 15-FEB-20         12         19            10         300
      2105        102 21-MAR-20         18         12           100        2000
      1200        155 30-MAR-20         20         20           300        3600
      1400        165 05-JUN-20         22         15           100        4000
      2105        125 05-JUN-20         12         16            40        8500
      2105        302 15-SEP-20         10         11            75        4350
      2108        305 18-NOV-20         10         11            70        4250

10 rows selected.
```

Create a temporary internal table for the exchanging of data with the hybrid partitioned table.

Note that this next step is not necessarily a one-to-one copy as shown here.

```
CREATE TABLE sales_year_2020 AS SELECT * FROM ext_sales_year_2020;

SELECT * FROM sales_year_2020;

   PROD_ID    CUST_ID TIME_ID   CHANNEL_ID   PROMO_ID QUANTITY_SOLD AMOUNT_SOLD
---------- ---------- --------- ---------- ---------- ------------- -----------
      1001        100 10-JAN-20         10         15           500        7500
      1002        110 15-JAN-20         12         18           100        3200
      1001        100 20-JAN-20         10         15           500        7500
      2105        101 15-FEB-20         12         19            10         300
      2105        102 21-MAR-20         18         12           100        2000
      1200        155 30-MAR-20         20         20           300        3600
      1400        165 05-JUN-20         22         15           100        4000
      2105        125 05-JUN-20         12         16            40        8500
      2105        302 15-SEP-20         10         11            75        4350
      2108        305 18-NOV-20         10         11            70        4250

10 rows selected.
```

Exchange data in the temporary internal table with the `sales_2020` partition to load the data into the hybrid partitioned table.

```
ALTER TABLE hybrid_datapump_sales
       EXCHANGE PARTITION(sales_2020) WITH TABLE sales_year_2020;

SELECT * FROM hybrid_datapump_sales PARTITION(sales_2020);

   PROD_ID    CUST_ID TIME_ID   CHANNEL_ID   PROMO_ID QUANTITY_SOLD AMOUNT_SOLD
---------- ---------- --------- ---------- ---------- ------------- -----------
      1001        100 10-JAN-20         10         15           500        7500
      1002        110 15-JAN-20         12         18           100        3200
      1001        100 20-JAN-20         10         15           500        7500
      2105        101 15-FEB-20         12         19            10         300
      2105        102 21-MAR-20         18         12           100        2000
      1200        155 30-MAR-20         20         20           300        3600
      1400        165 05-JUN-20         22         15           100        4000
      2105        125 05-JUN-20         12         16            40        8500
      2105        302 15-SEP-20         10         11            75        4350
      2108        305 18-NOV-20         10         11            70        4250

10 rows selected.
```

4.8 Viewing Information About Partitioned Tables and Indexes

You can display information about partitioned tables and indexes with Oracle Database views.

Table 4-4 lists the views that contain information specific to partitioned tables and indexes:

Table 4-4 Views With Information Specific to Partitioned Tables and Indexes

View	Description

`DBA_PART_TABLES`
`ALL_PART_TABLES`
`USER_PART_TABLES`
	`DBA` view displays partitioning information for all partitioned tables in the database. `ALL` view displays partitioning information for all partitioned tables accessible to the user. `USER` view is restricted to partitioning information for partitioned tables owned by the user.

`DBA_TAB_PARTITIONS`
`ALL_TAB_PARTITIONS`
`USER_TAB_PARTITIONS`
	Display partition-level partitioning information, partition storage parameters, and partition statistics generated by the `DBMS_STATS` package or the `ANALYZE` statement.

`DBA_TAB_SUBPARTITIONS`
`ALL_TAB_SUBPARTITIONS`
`USER_TAB_SUBPARTITIONS`
	Display subpartition-level partitioning information, subpartition storage parameters, and subpartition statistics generated by the `DBMS_STATS` package or the `ANALYZE` statement.

`DBA_PART_KEY_COLUMNS`
`ALL_PART_KEY_COLUMNS`
`USER_PART_KEY_COLUMNS`
	Display the partitioning key columns for partitioned tables.

`DBA_SUBPART_KEY_COLUMNS`
`ALL_SUBPART_KEY_COLUMNS`
`USER_SUBPART_KEY_COLUMNS`
	Display the subpartitioning key columns for composite-partitioned tables (and local indexes on composite-partitioned tables).

`DBA_PART_COL_STATISTICS`
`ALL_PART_COL_STATISTICS`
`USER_PART_COL_STATISTICS`
	Display column statistics and histogram information for the partitions of tables.

Table 4-4 (Cont.) Views With Information Specific to Partitioned Tables and Indexes

View	Description
DBA_SUBPART_COL_STATISTICS ALL_SUBPART_COL_STATISTICS USER_SUBPART_COL_STATISTICS	Display column statistics and histogram information for subpartitions of tables.
DBA_PART_HISTOGRAMS ALL_PART_HISTOGRAMS USER_PART_HISTOGRAMS	Display the histogram data (end-points for each histogram) for histograms on table partitions.
DBA_SUBPART_HISTOGRAMS ALL_SUBPART_HISTOGRAMS USER_SUBPART_HISTOGRAMS	Display the histogram data (end-points for each histogram) for histograms on table subpartitions.
DBA_PART_INDEXES ALL_PART_INDEXES USER_PART_INDEXES	Display partitioning information for partitioned indexes.
DBA_IND_PARTITIONS ALL_IND_PARTITIONS USER_IND_PARTITIONS	Display the following for index partitions: partition-level partitioning information, storage parameters for the partition, statistics collected by the DBMS_STATS package or the ANALYZE statement.
DBA_IND_SUBPARTITIONS ALL_IND_SUBPARTITIONS USER_IND_SUBPARTITIONS	Display the following information for index subpartitions: partition-level partitioning information, storage parameters for the partition, statistics collected by the DBMS_STATS package or the ANALYZE statement.
DBA_SUBPARTITION_TEMPLATES ALL_SUBPARTITION_TEMPLATES USER_SUBPARTITION_TEMPLATES	Display information about existing subpartition templates.

See Also:

- *Oracle Database Reference* for descriptions of database views
- *Oracle Database SQL Tuning Guide* for information about histograms and generating statistics for tables
- *Oracle Database Administrator's Guide* for more information about analyzing tables, indexes, and clusters

5

Managing and Maintaining Time-Based Information

Oracle Database provides strategies to manage and maintain data based on time.

This chapter discusses the components in Oracle Database which can build a strategy to manage and maintain data based on time.

Although most organizations have long regarded their stores of data as one of their most valuable corporate assets, how this data is managed and maintained varies enormously from company to company. Originally, data was used to help achieve operational goals, run the business, and help identify the future direction and success of the company.

However, new government regulations and guidelines are a key driving force in how and why data is being retained. Regulations now require organizations to retain and control information for very long periods of time. Consequently, today there are additional objectives that information technology (IT) managers are trying to satisfy:

- To store vast quantities of data for the lowest possible cost
- To meet the new regulatory requirements for data retention and protection
- To improve business opportunities by better analysis based on an increased amount of data

This chapter contains the following sections:

- Managing Data in Oracle Database With ILM
- Implementing an ILM Strategy With Heat Map and ADO
- Controlling the Validity and Visibility of Data in Oracle Database
- Implementing an ILM System Manually Using Partitioning
- Managing ILM Heat Map and ADO with Oracle Enterprise Manager

5.1 Managing Data in Oracle Database With ILM

With Information Lifecycle Management (ILM), you can manage data in Oracle Database using rules and regulations that apply to that data.

Information today comes in a wide variety of types, for example an e-mail message, a photograph, or an order in an Online Transaction Processing (OLTP) System. After you know the type of data and how it is used, you have an understanding of what its evolution and final disposition is likely to be.

One challenge facing each organization is to understand how its data evolves and grows, monitor how its usage changes over time, and decide how long it should survive, while adhering to all the rules and regulations that now apply to that data. Information Lifecycle Management (ILM) is designed to address these issues, with a combination of processes, policies, software, and hardware so that the appropriate technology can be used for each stage in the lifecycle of the data.

This section contains the following topics:

- About Oracle Database for ILM
- Implementing ILM Using Oracle Database

5.1.1 About Oracle Database for ILM

Oracle Database provides the ideal platform for implementing an ILM solution.

The Oracle Database platform offers the following:

- Application Transparency

  Application transparency is very important in ILM because it means that there is no need to customize applications and it also enables various changes to be made to the data without any effect on the applications that are using that data. Data can easily be moved at the different stages of its lifecycle and access to the data can be optimized with the database. Another important benefit is that application transparency offers the flexibility required to quickly adapt to any new regulatory requirements, again without any impact on the existing applications.

- Fine-grained data

  Oracle can view data at a very fine-grained level and group related data, whereas storage devices only see bytes and blocks.

- Low-Cost Storage

  With so much data to retain, using low cost storage is a key factor in implementing ILM. Because Oracle can take advantage of many types of storage devices, the maximum amount of data can be held for the lowest possible cost.

- Enforceable Compliance Policies

  When information is kept for compliance reasons, it is imperative to show to regulatory bodies that data is being retained and managed in accordance with the regulations. Within Oracle Database, it is possible to define security and audit policies, which enforce and log all access to data.

This section contains the following topics:

- Oracle Database Manages All Types of Data
- Regulatory Requirements
- The Benefits of an Online Archive

5.1.1.1 Oracle Database Manages All Types of Data

Information Lifecycle Management is concerned with all data in an organization.

This data includes not just structured data, such as orders in an OLTP system or a history of sales in a data warehouse, but also unstructured data, such as e-mail, documents, and images. Oracle Database supports the storing of unstructured data with BLOBs and Oracle SecureFiles, a sophisticated document management system is available in Oracle Text.

If all of the information in your organization is contained in Oracle Database, then you can take advantage of the features and functionality provided by the database to manage and move the data as it evolves during its lifetime, without having to manage multiple types of data stores.

5.1.1.2 Regulatory Requirements

Many organizations must retain specific data for a specific time period. Failure to follow these regulations could result in organizations having to pay very heavy fines.

Around the world various regulatory requirements, such as Sarbanes-Oxley, HIPAA, DOD5015.2-STD in the US and the European Data Privacy Directive in the European Union, are changing how organizations manage their data. These regulations specify what data must be retained, whether it can be changed, and for how long it must be retained, which could be for a period of 30 years or longer.

These regulations frequently demand that electronic data is secure from unauthorized access and changes, and that there is an audit trail of all changes to data and by whom. Oracle Database can retain huge quantities of data without impacting application performance. It also contains the features required to restrict access and prevent unauthorized changes to data, and can be further enhanced with Oracle Audit Vault and Database Firewall. Oracle Database also provides cryptographic functions that can demonstrate that a highly privileged user has not intentionally modified data. Using Flashback Data Technology, you can store all the versions of a row during its lifetime in a tamper proof historical archive.

5.1.1.3 The Benefits of an Online Archive

There are multiple benefits of an online archive.

There usually comes a point during the lifecycle of the data when it is no longer being regularly accessed and is considered eligible for archiving. Traditionally, the data would have been removed from the database and stored on tape, where you can store vast quantities of information for a very low cost. Today, it is no longer necessary to archive that data to tape, instead it can remain in the database, or be transferred to a central online archive database. All this information can be stored using low-cost storage devices whose cost per gigabyte is very close to that of tape.

There are multiple benefits to keeping all of the data in an Oracle Database for archival purposes. The most important benefit is that the data always be instantly available. Therefore, time is not wasted locating the tapes where the data was archived and determining whether the tape is readable and still in a format that can be loaded into the database.

If the data has been archived for many years, then development time may also be needed to write a program to reload the data into the database from the tape archive. This could prove to be expensive and time consuming, especially if the data is extremely old. If the data is retained in the database, then this is not a problem, because it is online, and in the latest database format.

Holding the historical data in the database no longer impacts the time required to backup the database and the size of the backup. When RMAN is used to back up the database, it only includes in the backup the data that has changed. Because historical data is less likely to change, after that data has been backed up, it is not backed up again.

Another important factor to consider is how the data is to be physically removed from the database, especially if it is to be transferred from a production system to a central database archive. Oracle provides the capability to move this data rapidly between databases by using transportable tablespaces or partitions, which moves the data as a complete unit.

When it is time to remove data from the database, the fastest way is to remove a set of data. This is achieved by keeping the data in its own partition. The partition can be dropped, which is a very fast operation. However, if this approach is not possible because data relationships

must be maintained, then a conventional SQL delete statement must be issued. You should not underestimate the time required to issue the delete statement.

If there is a requirement to remove data from the database and there is a possibility that the data may need to be returned to the database in the future, then consider removing the data in a database format such as a transportable tablespace, or use the XML capability of Oracle Database to extract the information in an open format.

Consider an online archive of your data into Oracle Database for the following reasons:

- The cost of disk is approaching that of tape, so you can eliminate the time to find the tape that contains the data and the cost of restoring that data
- Data remains online when needed, providing you faster access to meet business requirements
- Data online means immediate access, so fines by regulatory body for failing to produce data are less likely
- The current application can be used to access the data, so you do not need to waste resources to build a new application

5.1.2 Implementing ILM Using Oracle Database

Building an Information Lifecycle Management solution using Oracle Database is quite straightforward.

An ILM solution can be completed by following these four simple steps, although Step 4 is optional if ILM is not being implemented for compliance:

- Step 1: Define the Data Classes
- Step 2: Create Storage Tiers for the Data Classes
- Step 3: Create Data Access and Migration Policies
- Step 4: Define and Enforce Compliance Policies

5.1.2.1 Step 1: Define the Data Classes

To make effective use of Information Lifecycle Management, first review all the data in your organization before implementing an Information Lifecycle Management solution.

After reviewing the data, determine the following:

- What data is important, where is it stored, and what must be retained
- How this data flows within the organization
- What happens to this data over time and whether it is still required
- The degree of data availability and protection that is needed
- Data retention for legal and business requirements

After there is an understanding of how the data is used, the data can then be classified on this basis. The most common type of classification is by age or date, but other types are possible, such as by product or privacy. A hybrid classification could also be used, such as by privacy and age.

To treat the data classes differently, the data must be physically separated. When information is first created, the information is often frequently accessed, but then over time it may be referenced very infrequently. For instance, when a customer places an order, they regularly look at the order to see its status and whether the order has been shipped. After the order

arrives, they may never reference that order again. This order would also be included in regular reports that are run to see what goods are being ordered, but, over time, would not figure in any of the reports and may only be referenced in the future if someone does a detailed analysis that involves this data. For example, orders could be classified by the Financial Quarters Q1, Q2, Q3, and Q4, and as Historical Orders.

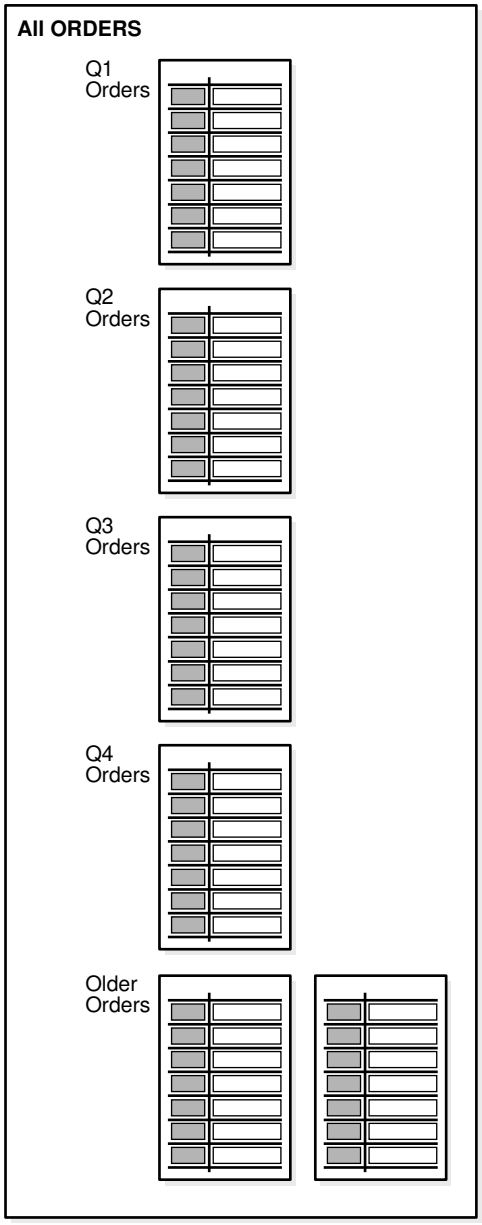
The advantage of using this approach is that when the data is grouped at the row level by its class, which in this example would be the date of the order, all orders for Q1 can be managed as a self contained unit, where as the orders for Q2 would reside in a different class. This can be achieved by using partitioning. Because partitions are transparent to the application, the data is physically separated but the application still locates all the orders.

5.1.2.1.1 Partitioning for ILM

Partitioning involves physically placing data according to a data value, and a frequently used technique is to partition information by date.

Figure 5-1 illustrates a scenario where the orders for Q1, Q2, Q3, and Q4 are stored in individual partitions and the orders for previous years are stored in other partitions.

Figure 5-1 Allocating Data Classes to a Partition



Oracle offers several different partitioning methods. Range partitioning is one frequently used partitioning method for ILM. Interval and reference partitioning are also particularly suited for use in an ILM environment.

There are multiple benefits to partitioning data. Partitioning provides an easy way to distribute the data across appropriate storage devices depending on its usage, while still keeping the data online and stored on the most cost-effective device. Because partitioning is transparent to anyone accessing the data, no application changes are required, thus partitioning can be implemented at any time. When new partitions are required, they are simply added using the `ADD PARTITION` clause or they are created automatically if interval partitioning is being used.

Among other benefits, each partition can have its own local index. When the optimizer uses partition pruning, queries only access the relevant partitions instead of all partitions, thus improving query response times.

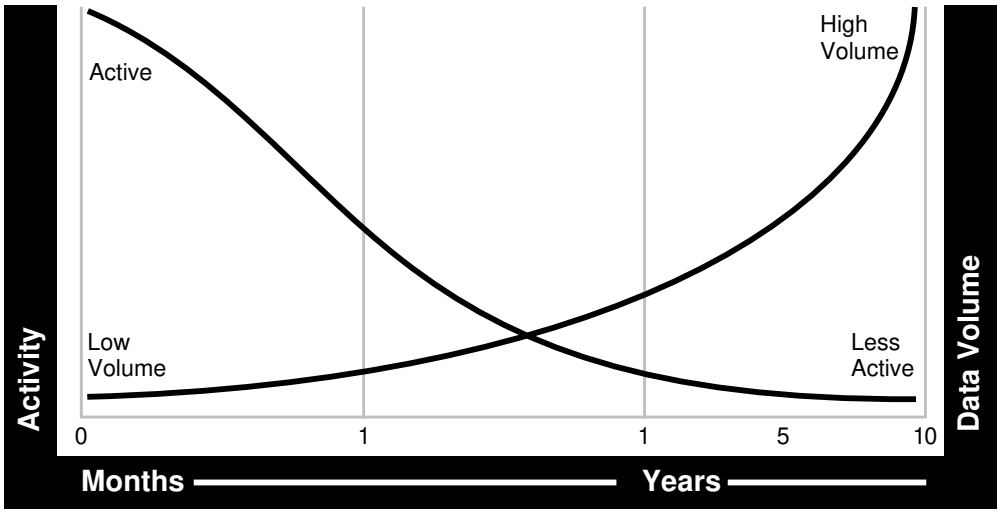
5.1.2.1.2 The Lifecycle of Data

An analysis of your data is likely to reveal that initially, it is accessed and updated on a very frequent basis. As the age of the data increases, its access frequency diminishes to almost negligible, if any.

Most organizations find themselves in the situation where many users are accessing current data while very few users are accessing older data, as illustrated in Figure 5-2. Data is considered to be: active, less active, historical, or ready to be archived.

With so much data being held, during its lifetime the data should be moved to different physical locations. Depending on where the data is in its lifecycle, it must be located on the most appropriate storage device.

Figure 5-2 Data Usage Over Time



5.1.2.2 Step 2: Create Storage Tiers for the Data Classes

Because Oracle Database can take advantage of many different storage options, the second step in implementing an Information Lifecycle Management solution is to establish the required storage tiers.

Although you can create as many storage tiers as you require, a suggested starting point are the following tiers:

- High Performance

  The high performance storage tier is where all the important and frequently accessed data, such as the partition holding our Q1 orders, is stored. This tier uses smaller, faster disks on high performance storage devices.

- Low Cost

  The low cost storage tier is where the less frequently accessed data is stored, such as the partitions holding the orders for Q2, Q3, and Q4. This tier is built using large capacity disks, such as those found in modular storage arrays or low costs ATA disks, which offer the maximum amount of inexpensive storage.

- Online Archive

The online archive storage tier is where all the data that is seldom accessed or modified is stored. This storage tier is likely to be extremely large and to store the maximum quantity of data. You can use various techniques to compress the data. Stored on low cost storage devices, such as ATA drives, the data would still be online and available, for a cost that is only slightly higher than storing this information on tape, without the disadvantages that come with archiving data to tape. If the Online Archive storage tier is identified as read-only, then it would be impossible to change the data and subsequent backups would not be required after the initial database backup.

- Offline Archive (optional)

  The offline archive storage tier is an optional tier because it is only used when there is a requirement to remove data from the database and store it in some other format, such as XML on tape.

Figure 5-2 illustrates how data is used over a time interval. Using this information, it can be determined that to retain all this information, several storage tiers are required to hold all of the data, which also has the benefit of significantly reducing total storage costs.

After the storage tiers have been created, the data classes identified in Step 1: Define the Data Classes are physically implemented inside the database using partitions. This approach provides an easy way to distribute the data across the appropriate storage devices depending on its usage, while still keeping the data online and readily available, and stored on the most cost-effective device.

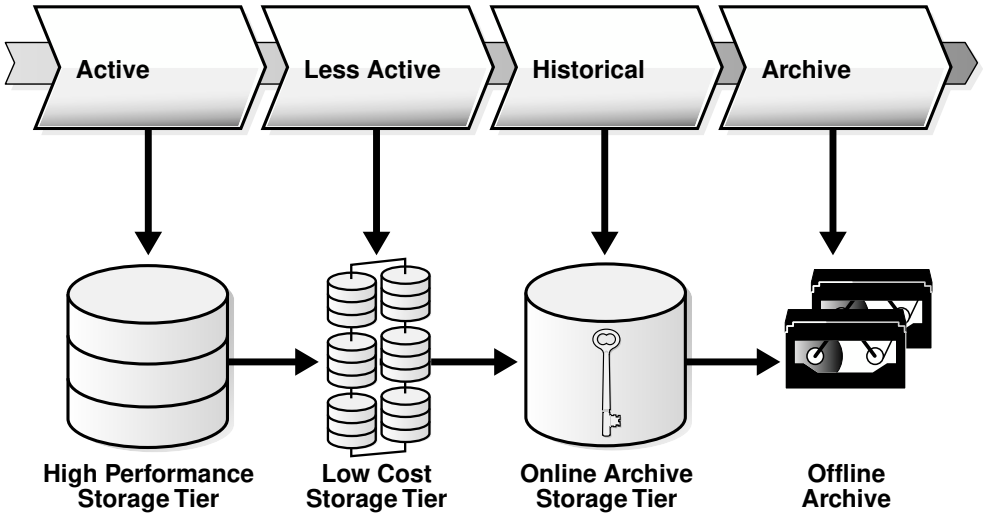
You can also use Oracle Automatic Storage Management (Oracle ASM) to manage the data across the storage tiers. Oracle ASM is a high-performance, ease-of-management storage solution for Oracle Database files. Oracle ASM is a volume manager and provides a file system designed exclusively for use by the database. To use Oracle ASM, you allocate partitioned disks for Oracle Database with preferences for striping and mirroring. Oracle ASM manages the disk space, distributing the I/O load across all available resources to optimize performance while removing the need for manual I/O tuning. For example, you can increase the size of the disk for the database or move parts of the database to new devices without having to shut down the database.

5.1.2.2.1 Assigning Classes to Storage Tiers

After the storage tiers have been defined, the data classes (partitions) identified in Step 1 can be assigned to the appropriate storage tiers.

This assignment provides an easy way to distribute the data across the appropriate storage devices depending on its usage, keeping the data online and available, and stored on the most cost-effective device. In Figure 5-3 data identified to be active, less active, historical, or ready to be archived is assigned to the high performance tier, low cost storage tier, online archive storage tier, and offline archive respectively. Using this approach, no application changes are required because the data is still visible.

Figure 5-3 Data Lifecycle



5.1.2.2.2 The Costs Savings of Using Tiered Storage

One benefit of implementing an ILM strategy is the cost savings that can result from using multiple tiered storage.

Assume that there is 3 TB of data to store, comprising of 200 GB on High Performance, 800 GB on Low Cost, and 2 TB on Online Archive. Assume the cost per GB is $72 on the High Performance tier, $14 on the Low Cost tier, and $7 on the Online Archive tier.

Table 5-1 illustrates the possible cost savings using tiered storage, rather than storing all data on one class of storage. As you can see, the cost savings can be quite significant and, if the data is suitable for OLTP and HCC database compression, then even further cost savings are possible.

Table 5-1 Cost Savings Using Tiered Storage

Storage Tier	Single Tier using High Performance Disks	Multiple Storage Tiers	Multiple Tiers with Database Compression
High Performance (200 GB)	$14,400	$14,400	$14,400
Low Cost (800 GB)	$57,600	$11,200	$11,200
Online Archive (2 TB)	$144,000	$14,000	$5,600
Total of each column	$216,000	$39,600	$31,200

5.1.2.3 Step 3: Create Data Access and Migration Policies

The third step in implementing an Information Lifecycle Management solution is to ensure that only authorized users have access to the data and to specify how to move the data during its lifetime.

As the data ages, there are multiple techniques that can migrate the data between the storage tiers.

5.1.2.3.1 Controlling Access to Data

The security of your data is another very important part of Information Lifecycle Management because the access rights to the data may change during its lifetime.

In addition, there may be regulatory requirements that place exacting demands on how the data can be accessed.

The data in Oracle Database can be secured using database features, such as:

- Database Security
- Views
- Virtual Private Database

Virtual Private Database (VPD) defines a very fine-grained level of access to the database. Security policies determine which rows may be viewed and the columns that are visible. Multiple policies can be defined so that different users and applications see different views of the same data. For example, the majority of users could see the information for Q1, Q2, Q3, and Q4, while only authorized users would be able to view the historical data.

A security policy is defined at the database level and is transparently applied to all database users. The benefit of this approach is that it provides a secure and controlled environment for accessing the data, which cannot be overridden and can be implemented without requiring any application changes. In addition, read-only tablespaces can be defined which ensures that the data does not change.

5.1.2.3.2 Moving Data using Partitioning

During its lifetime, data must be moved and partitioning is a technique that can be used.

Moving data may occur for the following reasons:

- For performance, only a limited number of orders are held on high performance disks
- Data is no longer frequently accessed and is using valuable high performance storage, and must be moved to a low-cost storage device
- Legal requirements demand that the information is always available for a given time interval, and it must be held safely for the lowest possible cost

There are multiple ways that data can be physically moved in Oracle Database to take advantage of the different storage tiers. For example, if the data is partitioned, then a partition containing the orders for Q2 could be moved online from the high performance storage tier to the low cost storage tier. Because the data is being moved within the database, it can be physically moved, without affecting the applications that require it or causing disruption to regular users.

Sometimes individual data items, rather than a group of data, must be moved. For example, suppose data was classified according to a level of privacy and a report, which had been secret, is now to be made available to the public. If the classification changed from secret to public and the data was partitioned on its privacy classification, then the row would automatically move to the partition containing public data.

Whenever data is moved from its original source, it is very important to ensure that the process selected adheres to any regulatory requirements, such as, the data cannot be altered, is secure from unauthorized access, easily readable, and stored in an approved location.

5.1.2.4 Step 4: Define and Enforce Compliance Policies

The fourth step in an Information Lifecycle Management solution is the creation of policies for compliance.

When data is decentralized and fragmented, compliance policies have to be defined and enforced in every data location, which could easily result in a compliance policy being overlooked. However, using Oracle Database to provide a central location for storing data means that it is very easy to enforce compliance policies because they are all managed and enforced from one central location.

When defining compliance policies, consider the following areas:

- Data Retention
- Immutability
- Privacy
- Auditing
- Expiration

5.1.2.4.1 Data Retention

The retention policy describes how the data is to be retained, how long it must be kept, and what happens after data life.

An example of a retention policy is a record must be stored in its original form, no modifications are allowed, it must be kept for seven years, and then it may be deleted. Using Oracle Database security, it is possible to ensure that data remains unchanged and that only authorized processes can remove the data at the appropriate time. Retention policies can also be defined through a lifecycle definition in the ILM Assistant.

5.1.2.4.2 Immutability

Immutability is concerned with proving to an external party that data is complete and has not been modified.

Cryptographic or digital signatures can be generated by Oracle Database and retained either inside or outside of the database, to show that data has not been altered.

5.1.2.4.3 Privacy

Oracle Database provides several ways to ensure data privacy.

Access to data can be strictly controlled with security policies defined using Virtual Private Database (VPD). In addition, individual columns can be encrypted so that anyone looking at the raw data cannot see its contents.

5.1.2.4.4 Auditing

Oracle Database can track all access and changes to data.

These auditing capabilities can be defined either at the table level or through fine-grained auditing, which specifies the criteria for when an audit record is generated. Auditing can be further enhanced using Oracle Audit Vault and Database Firewall.

> **See Also:**
>
> *Oracle Audit Vault and Database Firewall Administrator's Guide* for information about Oracle Audit Vault and Database Firewall

5.1.2.4.5 Expiration

Ultimately, data may expire for business or regulatory reasons and must be removed from the database.

Oracle Database can remove data very quickly and efficiently by simply dropping the partition which contains the information identified for removal.

5.2 Implementing an ILM Strategy With Heat Map and ADO

To implement an Information Lifecycle Management (ILM) strategy for data movement in your database, you can use Heat Map and Automatic Data Optimization (ADO) features.

> **Note:**
>
> Heat Map and ADO are supported in Oracle Database 12*c* Release 2 multitenant environments.

This section contains the following topics:

- Using Heat Map
- Using Automatic Data Optimization
- Limitations and Restrictions With ADO and Heat Map

> **See Also:**
>
> - Managing ILM Heat Map and ADO with Oracle Enterprise Manager for information about using Oracle Enterprise Manager Cloud Control with Heat Map and ADO
> - *Oracle Database Vault Administrator's Guide* for information about using Information Lifecycle Management (ILM) with Oracle Database Vault realms and command rules, including granting the authorization that enables an ADO administrative user to perform ILM operations on Database Vault-protected objects.

5.2.1 Using Heat Map

To implement your ILM strategy, you can use Heat Map in Oracle Database to track data access and modification.

Heat Map provides data access tracking at the segment-level and data modification tracking at the segment and row level. You can enable this functionality with the `HEAT_MAP` initialization parameter.

Heat Map data can assist Automatic Data Optimization (ADO) to manage the contents of the In-Memory column store (IM column store) using ADO policies. Using Heat Map data, which includes column statistics and other relevant statistics, the IM column store can determine when it is almost full (under memory pressure). If the determination is *almost full*, then inactive segments can be evicted if there are more frequently accessed segments that would benefit from population in the IM column store.

This section contains the following topics:

- Enabling and Disabling Heat Map
- Displaying Heat Map Tracking Data With Views
- Managing Heat Map Data With DBMS_HEAT_MAP Subprograms

> **See Also:**
>
> - *Oracle Database In-Memory Guide* for information about enabling and sizing the In-Memory Column Store

5.2.1.1 Enabling and Disabling Heat Map

You can enable and disable heat map tracking at the system or session level with the `ALTER SYSTEM` or `ALTER SESSION` statement using the `HEAT_MAP` clause.

For example, the following SQL statement enables Heat Map tracking for the database instance.

```
ALTER SYSTEM SET HEAT_MAP = ON;
```

When Heat Map is enabled, all accesses are tracked by the in-memory activity tracking module. Objects in the `SYSTEM` and `SYSAUX` tablespaces are not tracked.

The following SQL statement disables heat map tracking.

```
ALTER SYSTEM SET HEAT_MAP = OFF;
```

When Heat Map is disabled, accesses are not tracked by the in-memory activity tracking module. The default value for the `HEAT_MAP` initialization parameter is `OFF`.

The `HEAT_MAP` initialization parameter also enables and disables Automatic Data Optimization (ADO). For ADO, Heat Map must be enabled at the system level.

> **See Also:**
>
> - Using Automatic Data Optimization for more information about ADO
> - *Oracle Database Reference* for information about the `HEAT_MAP` initialization parameter

5.2.1.2 Displaying Heat Map Tracking Data With Views

Heat map tracking data is viewed with `V$*`, `ALL*`, `DBA*`, and `USER*` heat map views.

Example 5-1 shows examples of information provided by heat map views. The `V$HEAT_MAP_SEGMENT` view displays real-time segment access information. The `ALL_`, `DBA_`, and `USER_HEAT_MAP_SEGMENT` views display the latest segment access time for all segments visible to the user. The `ALL_`, `DBA_`, and `USER_HEAT_MAP_SEG_HISTOGRAM` views display segment access information for all segments visible to the user. The `DBA_HEATMAP_TOP_OBJECTS` view displays heat map information for the top most active objects. The `DBA_HEATMAP_TOP_TABLESPACES` view displays heat map information for the top most active tablespaces.

> **See Also:**
>
> *Oracle Database Reference* for information about Heat Map views

Example 5-1 Heat map views

```
/* enable heat map tracking if necessary*/

SELECT SUBSTR(OBJECT_NAME,1,20), SUBSTR(SUBOBJECT_NAME,1,20), TRACK_TIME, SEGMENT_WRITE,
  FULL_SCAN, LOOKUP_SCAN FROM V$HEAT_MAP_SEGMENT;

SUBSTR(OBJECT_NAME,1 SUBSTR(SUBOBJECT_NAM TRACK_TIM SEG FUL LOO
-------------------- -------------------- --------- --- --- ---
SALES                SALES_Q1_1998        01-NOV-12 NO  NO  NO
SALES                SALES_Q3_1998        01-NOV-12 NO  NO  NO
SALES                SALES_Q2_2000        01-NOV-12 NO  NO  NO
SALES                SALES_Q3_1999        01-NOV-12 NO  NO  NO
SALES                SALES_Q2_1998        01-NOV-12 NO  NO  NO
SALES                SALES_Q2_1999        01-NOV-12 NO  NO  NO
SALES                SALES_Q4_2001        01-NOV-12 NO  NO  NO
SALES                SALES_Q1_1999        01-NOV-12 NO  NO  NO
SALES                SALES_Q4_1998        01-NOV-12 NO  NO  NO
SALES                SALES_Q1_2000        01-NOV-12 NO  NO  NO
SALES                SALES_Q1_2001        01-NOV-12 NO  NO  NO
SALES                SALES_Q2_2001        01-NOV-12 NO  NO  NO
SALES                SALES_Q3_2000        01-NOV-12 NO  NO  NO
SALES                SALES_Q4_2000        01-NOV-12 NO  NO  NO
EMPLOYEES                                 01-NOV-12 NO  NO  NO
...

SELECT SUBSTR(OBJECT_NAME,1,20), SUBSTR(SUBOBJECT_NAME,1,20), SEGMENT_WRITE_TIME,
  SEGMENT_READ_TIME, FULL_SCAN, LOOKUP_SCAN FROM USER_HEAT_MAP_SEGMENT;

SUBSTR(OBJECT_NAME,1 SUBSTR(SUBOBJECT_NAM SEGMENT_W SEGMENT_R FULL_SCAN LOOKUP_SC
-------------------- -------------------- --------- --------- --------- ---------
SALES                SALES_Q1_1998                              30-OCT-12 01-NOV-12
SALES                SALES_Q1_1998                              30-OCT-12 01-NOV-12
SALES                SALES_Q1_1998                              30-OCT-12 01-NOV-12
SALES                SALES_Q1_1998                              30-OCT-12 01-NOV-12
SALES                SALES_Q1_1998                              30-OCT-12 01-NOV-12
SALES                SALES_Q1_1998                              30-OCT-12 01-NOV-12
...

SELECT SUBSTR(OBJECT_NAME,1,20), SUBSTR(SUBOBJECT_NAME,1,20), TRACK_TIME, SEGMENT_WRITE, FULL_SCAN,
```

```
  LOOKUP_SCAN FROM USER_HEAT_MAP_SEG_HISTOGRAM;

SUBSTR(OBJECT_NAME,1 SUBSTR(SUBOBJECT_NAM TRACK_TIM SEG FUL LOO
-------------------- -------------------- --------- --- --- ---
SALES                SALES_Q1_1998        31-OCT-12 NO  NO  YES
SALES                SALES_Q1_1998        01-NOV-12 NO  NO  YES
SALES                SALES_Q1_1998        30-OCT-12 NO  YES YES
SALES                SALES_Q2_1998        01-NOV-12 NO  NO  YES
SALES                SALES_Q2_1998        31-OCT-12 NO  NO  YES
SALES                SALES_Q2_1998        30-OCT-12 NO  YES YES
SALES                SALES_Q3_1998        01-NOV-12 NO  NO  YES
SALES                SALES_Q3_1998        30-OCT-12 NO  YES YES
SALES                SALES_Q3_1998        31-OCT-12 NO  NO  YES
SALES                SALES_Q4_1998        01-NOV-12 NO  NO  YES
SALES                SALES_Q4_1998        31-OCT-12 NO  NO  YES
SALES                SALES_Q4_1998        30-OCT-12 NO  YES YES
SALES                SALES_Q1_1999        01-NOV-12 NO  NO  YES
SALES                SALES_Q1_1999        31-OCT-12 NO  NO  YES
...

SELECT SUBSTR(OWNER,1,20), SUBSTR(OBJECT_NAME,1,20), OBJECT_TYPE, SUBSTR(TABLESPACE_NAME,1,20),
    SEGMENT_COUNT FROM DBA_HEATMAP_TOP_OBJECTS ORDER BY SEGMENT_COUNT DESC;

SUBSTR(OWNER,1,20)   SUBSTR(OBJECT_NAME,1 OBJECT_TYPE        SUBSTR(TABLESPACE_NA SEGMENT_COUNT
-------------------- -------------------- ------------------ -------------------- -------------
SH                   SALES                TABLE              EXAMPLE                         96
SH                   COSTS                TABLE              EXAMPLE                         48
PM                   ONLINE_MEDIA         TABLE              EXAMPLE                         22
OE                   PURCHASEORDER        TABLE              EXAMPLE                         18
PM                   PRINT_MEDIA          TABLE              EXAMPLE                         15
OE                   CUSTOMERS            TABLE              EXAMPLE                         10
OE                   WAREHOUSES           TABLE              EXAMPLE                          9
HR                   EMPLOYEES            TABLE              EXAMPLE                          7
OE                   LINEITEM_TABLE       TABLE              EXAMPLE                          6
IX                   STREAMS_QUEUE_TABLE  TABLE              EXAMPLE                          6
SH                   FWEEK_PSCAT_SALES_MV TABLE              EXAMPLE                          5
SH                   CUSTOMERS            TABLE              EXAMPLE                          5
HR                   LOCATIONS            TABLE              EXAMPLE                          5
HR                   JOB_HISTORY          TABLE              EXAMPLE                          5
SH                   PRODUCTS             TABLE              EXAMPLE                          5
...

SELECT SUBSTR(TABLESPACE_NAME,1,20), SEGMENT_COUNT
    FROM DBA_HEATMAP_TOP_TABLESPACES ORDER BY SEGMENT_COUNT DESC;

SUBSTR(TABLESPACE_NA SEGMENT_COUNT
-------------------- -------------
EXAMPLE                        351
USERS                           11

SELECT COUNT(*) FROM DBA_HEATMAP_TOP_OBJECTS;

  COUNT(*)
----------
        64

SELECT COUNT(*) FROM DBA_HEATMAP_TOP_TABLESPACES;

  COUNT(*)
----------
         2
```

5.2.1.3 Managing Heat Map Data With DBMS_HEAT_MAP Subprograms

The `DBMS_HEAT_MAP` package provides additional flexibility for displaying heat map data using `DBMS_HEAT_MAP` subprograms.

`DBMS_HEAT_MAP` includes one set of APIs that externalize heat maps at various levels of storage such as block, extent, segment, object, and tablespace; and a second set of APIs that externalize the heat maps materialized by the background process for the top tablespaces.

Example 5-2 shows examples of the use of `DBMS_HEAT_MAP` package subprograms.

> **See Also:**
>
> *Oracle Database PL/SQL Packages and Types Reference* for information about the `DBMS_HEAT_MAP` package

Example 5-2 Using DBMS_HEAT_MAP package subprograms

```
SELECT SUBSTR(segment_name,1,10) Segment, min_writetime, min_ftstime
  FROM TABLE(DBMS_HEAT_MAP.OBJECT_HEAT_MAP('SH','SALES'));

SELECT SUBSTR(tablespace_name,1,16) Tblspace, min_writetime, min_ftstime
  FROM  TABLE(DBMS_HEAT_MAP.TABLESPACE_HEAT_MAP('EXAMPLE'));

SELECT relative_fno, block_id, blocks, TO_CHAR(min_writetime, 'mm-dd-yy hh-mi-ss') Mintime,
   TO_CHAR(max_writetime, 'mm-dd-yy hh-mi-ss') Maxtime,
   TO_CHAR(avg_writetime, 'mm-dd-yy hh-mi-ss') Avgtime
   FROM TABLE(DBMS_HEAT_MAP.EXTENT_HEAT_MAP('SH','SALES')) WHERE ROWNUM < 10;

SELECT SUBSTR(owner,1,10) Owner, SUBSTR(segment_name,1,10) Segment,
  SUBSTR(partition_name,1,16) Partition, SUBSTR(tablespace_name,1,16) Tblspace,
  segment_type, segment_size FROM TABLE(DBMS_HEAT_MAP.OBJECT_HEAT_MAP('SH','SALES'));

OWNER      SEGMENT    PARTITION        TBLSPACE         SEGMENT_TYPE         SEGMENT_SIZE
---------- ---------- ---------------- ---------------- -------------------- ------------
SH         SALES      SALES_Q1_1998    EXAMPLE          TABLE PARTITION           8388608
SH         SALES      SALES_Q2_1998    EXAMPLE          TABLE PARTITION           8388608
SH         SALES      SALES_Q3_1998    EXAMPLE          TABLE PARTITION           8388608
SH         SALES      SALES_Q4_1998    EXAMPLE          TABLE PARTITION           8388608
SH         SALES      SALES_Q1_1999    EXAMPLE          TABLE PARTITION           8388608
...
```

5.2.2 Using Automatic Data Optimization

To implement your ILM strategy, you can use Automatic Data Optimization (ADO) to automate the compression and movement of data between different tiers of storage within the database.

The functionality includes the ability to create policies that specify different compression levels for each tier, and to control when the data movement takes place.

This section contains the following topics:

- Managing Policies for Automatic Data Optimization
- Creating a Table With an ILM ADO Policy
- Adding ILM ADO Policies

- Disabling and Deleting ILM ADO Policies
- Specifying Segment-Level Compression and Storage Tiering With ADO
- Specifying Row-Level Compression Tiering With ADO
- Managing ILM ADO Parameters
- Using PL/SQL Functions for Policy Management
- Using Views to Monitor Policies for ADO

To use Automatic Data Optimization, you must enable Heat Map at the system level. You enable this functionality with the `HEAT_MAP` initialization parameter. For information about setting the `HEAT_MAP` initialization parameter, refer to Enabling and Disabling Heat Map.

5.2.2.1 Managing Policies for Automatic Data Optimization

You can specify policies for ADO at the row, segment, and tablespace granularity level when creating and altering tables with SQL statements. In addition, ADO policies can perform actions on indexes.

By specifying policies for ADO, you can automate data movement between different tiers of storage within the database. These policies also enable you to specify different compression levels for each tier, control when the data movement takes place, and optimize indexes.

ADO Policies for Tables

The ILM clauses of the SQL `CREATE` and `ALTER TABLE` statements enable you to create, delete, enable or disable a policy for ADO. An ILM policy clause determines the compression or storage tiering policy and contains additional clauses, such as the `AFTER` and `ON` clauses to specify the condition when a policy action should occur. When you create a table, you can add a new policy for ADO. You can alter the table to add more policies or to enable, disable, or delete existing policies. You can add policies to an entire table or a partition of a table. You can specify only one condition type for an `AFTER` clause when adding ADO policies to a table or partition of a table. ILM ADO policies are given a system-generated name, such `P1`, `P2`, and so on to `Pn`.

A segment level policy executes only one time. After the policy executes successfully, it is disabled and is not evaluated again. However, you can explicitly enable the policy again. A row level policy continues to execute and is not disabled after a successful execution.

The scope of an ADO policy can be specified for a group of related objects or at the level of a segment or row, using the keywords `GROUP`, `ROW`, or `SEGMENT`.

The default mappings for compression that can be applied to group policies are:

- `COMPRESS ADVANCED` on a heap table maps to standard compression for indexes and `LOW` for LOB segments.
- `COMPRESS FOR QUERY LOW`/`QUERY HIGH` on a heap table maps to standard compression for indexes and `MEDIUM` for LOB segments.
- `COMPRESS FOR ARCHIVE LOW`/`ARCHIVE HIGH` on a heap table maps to standard compression for indexes and `HIGH` for LOB segments.

The compression mapping cannot be changed. `GROUP` can only be applied to segment level policies. The storage tiering policies are applicable only at the segment level and cannot be specified at the row level.

ADO Policies for Indexes

ADO policies for indexes enable the compression and optimization for indexes using the existing Automatic Data Optimization (ADO) framework.

You can add an ADO index policy with the ILM clause of the `ALTER INDEX` or `CREATE INDEX` SQL statement. An ADO index policy is given a system-generated name, such as P1, P2, ... P*nn*.

For example, you can add an ADO policy when the index is created.

```
CREATE TABLE product_sales
   (PRODUCT_ID NUMBER NOT NULL,
    CUSTOMER_ID NUMBER NOT NULL,
    TIME_ID DATE NOT NULL,
    CHANNEL_ID NUMBER NOT NULL,
    PROMO_ID NUMBER,
    QUANTITY_SOLD NUMBER(10,2) NOT NULL);

CREATE INDEX prod_id_idx ON product_sales(product_id) ILM ADD POLICY OPTIMIZE AFTER 7
DAYS OF NO MODIFICATION;

SELECT POLICY_NAME, POLICY_TYPE, ENABLED FROM USER_ILMPOLICIES;
POLICY_NAME             POLICY_TYPE    ENA
----------------------  -------------  ---
P21                     DATA MOVEMENT  YES
```

You can add an ADO policy to an existing index.

```
ALTER INDEX hr.emp_id_idx ILM ADD POLICY SEGMENT TIER TO LOW_COST_TBS;

ALTER INDEX hr.emp_id_idx ILM ADD POLICY OPTIMIZE AFTER 3 DAYS OF NO ACCESS;
```

The `OPTIMIZE` clause enables ADO to optimize the index whenever the policy condition is met. The optimization process includes actions such as compressing, shrinking, or rebuilding indexes.

- Compress: Compresses portions of the key values in an index segment
- Shrink: Merges the contents of index blocks where possible to free blocks for reuse
- Rebuild: Rebuilds an index to improve space usage and access speed

When the `OPTIMIZE` clause is specified, Oracle automatically determines which action is optimal for the index and implements that action as part of the optimization process. You do not have to specify which action is taken.

When administering ADO policies for indexes, you cannot manually disable policies. An ADO policy for indexes executes only one time. After the policy executes successfully, the policy is disabled and is not evaluated again.

You can delete one policy at a time with the ILM clause of `ALTER INDEX` SQL statement. For example:

```
ALTER INDEX prod_id_idx ILM DELETE POLICY p21;
```

Modifying an ILM ADO policy at the index partition level is not supported. An ADO policy modified at the index level is cascaded to all partitions.

ADO Policies for In-Memory Column Store

Automatic Data Optimization (ADO) supports the In-Memory Column Store (IM column store) with the `INMEMORY`, `INMEMORY MEMCOMPRESS`, and `NO INMEMORY` policy types.

- To enable objects for population in the In-Memory Column Store, include `INMEMORY` in the `ADD POLICY` clause.
- To increase the compression level on objects in an IM column store, include `INMEMORY MEMCOMPRESS` in the `ADD POLICY` clause.
- To explicitly evict objects that benefit the least from the IM column store, include `NO INMEMORY` in the `ADD POLICY` clause. For example:

The following is an example of the use the `NO INMEMORY` clause to evict objects from the IM column store.

```
ALTER TABLE sales_2015 ILM ADD POLICY NO INMEMORY
      AFTER 7 DAYS OF NO ACCESS;
```

An ADO policy with an In-Memory Column Store clause can only be a segment level policy. The `USER/DBA_ILMDATAMOVEMENTPOLICIES` and `V$HEAT_MAP_SEGMENT` views include information about ADO policies for the In-Memory Column Store.

Customizing ADO Policies

You can customize policies with the `ON PL/SQL_function` option which provides the ability to determine when the policy should be executed. The `ON PL/SQL_function` option is available only with segment level policies. For example:

```
CREATE OR REPLACE FUNCTION my_custom_ado_rules (objn IN NUMBER) RETURN BOOLEAN;

ALTER TABLE sales_custom ILM ADD POLICY COMPRESS ADVANCED SEGMENT
      ON my_custom_ado_rules;
```

> **See Also:**
>
> - *Oracle Database In-Memory Guide* for information about In-Memory Column Store and ADO support
> - *Oracle Database SQL Language Reference* for information about the syntax of the ILM clauses in the SQL CREATE TABLE statement
> - *Oracle Database SQL Language Reference* for information about the syntax of the ILM clauses in the SQL CREATE INDEX statement

5.2.2.2 Creating a Table With an ILM ADO Policy

Use the `ILM ADD POLICY` clause with the `CREATE TABLE` statement to create a table with ILM ADO policy.

The SQL statement in Example 5-3 creates a table and adds an ILM policy.

Example 5-3 Creating a table with an ILM ADO policy

```
/* Create an example table with an ILM ADO policy */
CREATE TABLE sales_ado
 (PROD_ID NUMBER NOT NULL,
  CUST_ID NUMBER NOT NULL,
  TIME_ID DATE NOT NULL,
  CHANNEL_ID NUMBER NOT NULL,
  PROMO_ID NUMBER NOT NULL,
  QUANTITY_SOLD NUMBER(10,2) NOT NULL,
```

```
  AMOUNT_SOLD NUMBER(10,2) NOT NULL )
 PARTITION BY RANGE (time_id)
 ( PARTITION sales_q1_2012 VALUES LESS THAN (TO_DATE('01-APR-2012','dd-MON-yyyy')),
   PARTITION sales_q2_2012 VALUES LESS THAN (TO_DATE('01-JUL-2012','dd-MON-yyyy')),
   PARTITION sales_q3_2012 VALUES LESS THAN (TO_DATE('01-OCT-2012','dd-MON-yyyy')),
   PARTITION sales_q4_2012 VALUES LESS THAN (TO_DATE('01-JAN-2013','dd-MON-yyyy')) )
  ILM ADD POLICY COMPRESS FOR ARCHIVE HIGH SEGMENT
      AFTER 12 MONTHS OF NO ACCESS;

/* View the existing ILM ADO polices */
SELECT SUBSTR(policy_name,1,24) POLICY_NAME, policy_type, enabled
       FROM USER_ILMPOLICIES;

POLICY_NAME              POLICY_TYPE   ENABLE
------------------------ ------------- ------
P1                       DATA MOVEMENT YES
```

5.2.2.3 Adding ILM ADO Policies

Use the `ILM ADD POLICY` clause with the `ALTER TABLE` statement to add an ILM ADO policy to a table.

The SQL statements in Example 5-4 provide examples of adding ILM policies to a partition of the `sales` table.

Example 5-4 Adding ILM ADO policies

```
/* Add a row-level compression policy after 30 days of no modifications */
ALTER TABLE sales MODIFY PARTITION sales_q1_2002
  ILM ADD POLICY ROW STORE COMPRESS ADVANCED ROW
  AFTER 30 DAYS OF NO MODIFICATION;

/* Add a segment level compression policy for data after 6 months of no modifications */
ALTER TABLE sales MODIFY PARTITION sales_q1_2001
  ILM ADD POLICY COMPRESS FOR ARCHIVE HIGH SEGMENT
  AFTER 6 MONTHS OF NO MODIFICATION;

/* Add a segment level compression policy for data after 12 months of no access */
ALTER TABLE sales MODIFY PARTITION sales_q1_2000
      ILM ADD POLICY COMPRESS FOR ARCHIVE HIGH SEGMENT
      AFTER 12 MONTHS OF NO ACCESS;

/* Add storage tier policy to move old data to a different tablespace */
/* that is on low cost storage media */
ALTER TABLE sales MODIFY PARTITION sales_q1_1999
  ILM ADD POLICY
  TIER TO my_low_cost_sales_tablespace;

/* View the existing polices */
SELECT SUBSTR(policy_name,1,24) POLICY_NAME, policy_type, enabled
   FROM USER_ILMPOLICIES;

POLICY_NAME              POLICY_TYPE   ENABLE
------------------------ ------------- ------
P1                       DATA MOVEMENT YES
P2                       DATA MOVEMENT YES
P3                       DATA MOVEMENT YES
P4                       DATA MOVEMENT YES
P5                       DATA MOVEMENT YES
```

Note: Adding multiple row-level policies is not supported. Multiple policies are supported at the segment-level and for table partitions. Consider implementing additional polices at either of these levels, if possible.

5.2.2.4 Disabling and Deleting ILM ADO Policies

Use the `ILM DISABLE POLICY` or `ILM DELETE POLICY` clauses with the `ALTER TABLE` statement to disable or delete an ILM ADO policy.

You can disable or delete ILM policies for ADO as shown in the SQL statements in Example 5-5. At times you may need to remove existing ILM policies if those policies conflict with a new policy that you want to add.

Example 5-5 Disabling and deleting ILM ADO policies

```
/* You can disable or delete an ADO policy in a table with the following */
ALTER TABLE sales_ado ILM DISABLE POLICY P1;
ALTER TABLE sales_ado ILM DELETE POLICY P1;

/* You can disable or delete all ADO policies in a table with the following */
ALTER TABLE sales_ado ILM DISABLE_ALL;
ALTER TABLE sales_ado ILM DELETE_ALL;

/* You can disable or delete an ADO policy in a partition with the following */
ALTER TABLE sales MODIFY PARTITION sales_q1_2002 ILM DISABLE POLICY P2;
ALTER TABLE sales MODIFY PARTITION sales_q1_2002 ILM DELETE POLICY P2;

/* You can disable or delete all ADO policies in a partition with the following */
ALTER TABLE sales MODIFY PARTITION sales_q1_2000 ILM DISABLE_all;
ALTER TABLE sales MODIFY PARTITION sales_q1_2000 ILM DELETE_ALL;
```

5.2.2.5 Specifying Segment-Level Compression and Storage Tiering With ADO

You can specify compression at the segment-level within a table using a segment-level compression tiering policy.

In combination with the row-level compression tiering policy, you have fine-grained control over how the data in your database is stored and managed.

Example 5-6 illustrates how to create policies for ADO to enforce a compression and storage tiering policy on the `sales_ado` table, reflecting the following business requirements:

1. Bulk Load Data
2. Run OLTP workloads
3. After six months with no updates, compress for Archive High
4. Move to low cost storage

Example 5-6 Using segment-level compression and storage tiering

```
/* Add a segment level compression policy after 6 months of no changes */
ALTER TABLE sales_ado ILM ADD POLICY
  COMPRESS FOR ARCHIVE HIGH SEGMENT
  AFTER 6 MONTHS OF NO MODIFICATION;

Table altered.

/* Add storage tier policy */
ALTER TABLE sales_ado ILM ADD POLICY
  TIER TO my_low_cost_tablespace;
```

```
SELECT SUBSTR(policy_name,1,24) POLICY_NAME, policy_type, enabled
  FROM USER_ILMPOLICIES;

POLICY_NAME              POLICY_TYPE   ENABLED
------------------------ ------------- -------
...
P6                       DATA MOVEMENT  YES
P7                       DATA MOVEMENT  YES
```

5.2.2.6 Specifying Row-Level Compression Tiering With ADO

Automatic Data Optimization (ADO) policies support Hybrid Columnar Compression (HCC) in addition to basic and advanced compression.

An HCC row level policy can be defined on any table regardless of the compression type of the table. Rows from cold blocks can be compressed with HCC when there is DML activity on other parts of the segment.

> **Note:**
>
> HCC row level policies are supported for ASSM (Automatic Segment Space Management) tablespaces only.

With HCC policies on non-HCC tables, there may be row movement during updates if the row is in a HCC compression unit (CU). Also, similar to other use cases of row movement, index maintenance is necessary to update index entries that referenced the moved row.

Row-level policies are supported in Oracle Database 12*c* Release 1 (12.1): however, the database must be at 12.2 compatibility or greater to use HCC row-level compression policies.

> **See Also:**
>
> *Oracle Database Administrator's Guide* for information about table compression

Example 5-7 Creating an ADO policy using row-level Hybrid Columnar Compression

The SQL statement in Example 5-7 creates a policy using HCC on the rows of the table `employees_ilm`.

```
ALTER TABLE employees_ilm
   ILM ADD POLICY COLUMN STORE COMPRESS FOR QUERY ROW
   AFTER 30 DAYS OF NO MODIFICATION;
```

Example 5-8 Creating an ADO policy using row-level advanced compression

The SQL statement in Example 5-8 creates a policy using advanced compression on the rows of the table `sales_ado`.

```
ALTER TABLE sales_ado
  ILM ADD POLICY ROW STORE COMPRESS ADVANCED ROW
  AFTER 60 DAYS OF NO MODIFICATION;

SELECT policy_name, policy_type, enabled
   FROM USER_ILMPOLICIES;
```

```
POLICY_NAME              POLICY_TYPE   ENABLE
------------------------ ------------- -------
...
P8                       DATA MOVEMENT YES
```

5.2.2.7 Managing ILM ADO Parameters

You can customize your ADO environment with ILM ADO parameters that you set with the `CUSTOMIZE_ILM` procedure in the `DBMS_ILM_ADMIN` PL/SQL package.

Various ILM ADO parameters are described in Table 5-2.

Table 5-2 ILM ADO Parameters

Name	Description
`ABSOLUTEJOB LIMIT`	The value for `ABSOLUTEJOB LIMIT` limits the absolute number of concurrent ADO jobs.
`DEGREEOF PARALLELISM`	The value for `DEGREEOF PARALLELISM` determines the degree of parallelism in which the ADO policy jobs are run.
`ENABLED`	The `ENABLED` parameter controls ADO background evaluation and execution. The default is enabled on (`TRUE` or `1`). The settings of `ENABLED` and the `HEAT_MAP` initialization parameters interact as follows: • If the `HEAT_MAP` initialization parameter is set to `ON` and the `ENABLED` parameter is set to `FALSE` (`0`), then heat map statistics are collected, but ADO does not act on the statistics automatically. • If the `HEAT_MAP` initialization parameter is set to `OFF` and the `ENABLED` parameter is set to `TRUE` (`1`), then heat map statistics are not collected and because ADO cannot rely on the heat map statistics, ADO does nothing. ADO behaves as if `ENABLED` is set to `FALSE`.
`EXECUTION MODE`	The value of `EXECUTION MODE` controls whether ADO executes in online or offline mode. The default is online (`2`).
`EXECUTION INTERVAL`	The value of `EXECUTION INTERVAL` determines the frequency that ADO initiates background evaluation. The default is `15` minutes.
`JOB LIMIT`	The value for `JOB LIMIT` controls the maximum number of ADO jobs at any time. The maximum number of concurrent ADO jobs is calculated as (`JOB LIMIT`)*(number of instances)*(number of CPUs for each instance). The default is `2`.
`POLICY TIME`	The value for `POLICY TIME` determines if ADO policies are specified in seconds or days. Values are `1` for seconds or `0` for days (default).
`RETENTION TIME`	The value for `RETENTION TIME` specifies the length of time that data of completed ADO tasks is kept before that data is purged. The default is `30` days.
`TBS PERCENT USED`	The value for `TBS_PERCENT_USED` parameter specifies the percentage of the tablespace quota when a tablespace is considered full. The default is `85` percent.
`TBS PERCENT FREE`	The value for `TBS_PERCENT_FREE` parameter specifies the targeted free percentage for the tablespace. The default is `25` percent.

For the values of the `TBS_PERCENT*` parameters, ADO makes a best effort, but not a guarantee. When the percentage of the tablespace quota reaches the value of `TBS_PERCENT_USED`, ADO begins to move data so that percent free of the tablespace quota approaches the value of `TBS_PERCENT_FREE`. As an example, assume that `TBS_PERCENT_USED` is set to 85 and `TBS_PERCENT_FREE` is set to `25`, and that a tablespace becomes `90` percent full. ADO then initiates actions to move data so that the tablespace quota has at least `25` percent free, which can also be interpreted as less than `75` percent used of the tablespace quota.

You can display the parameters with the DBA_ILMPARAMETERS view. For example, the following query displays the values of the ADO-related parameters.

```
SQL> SELECT NAME, VALUE FROM DBA_ILMPARAMETERS;

------------------------------------------------------------ ----------
ENABLED                                                               1
RETENTION TIME                                                       30
JOB LIMIT                                                             2
EXECUTION MODE                                                        2
EXECUTION INTERVAL                                                   15
TBS PERCENT USED                                                     85
TBS PERCENT FREE                                                     25
POLICY TIME                                                           0
ABSOLUTE JOB LIMIT                                                   10
DEGREE OF PARALLELISM                                                 4
...
```

See Also:

- Example 5-9 for an example showing how to set ILM ADO parameters with the CUSTOMIZE_ILM procedure in the DBMS_ILM_ADMIN PL/SQL package
- Managing ILM Heat Map and ADO with Oracle Enterprise Manager for information about setting ILM ADO parameters with Oracle Enterprise Manager Cloud Control
- *Oracle Database PL/SQL Packages and Types Reference* for a complete list of ILM ADO parameters
- *Oracle Database PL/SQL Packages and Types Reference* for information about the DBMS_ILM_ADMIN package

5.2.2.8 Using PL/SQL Functions for Policy Management

You can use the PL/SQL DBMS_ILM and DBMS_ILM_ADMIN packages for advanced policy management and customization to implement more complex ADO scenarios and control when policies are actively moving and compressing data.

With the PL/SQL DBMS_ILM and DBMS_ILM_ADMIN packages, you can manage ILM activities for ADO so that they do not negatively impact important production workloads. Database compatibility must be set to a minimum of 12.0 to use these packages.

The EXECUTE_ILM procedure of the DBMS_ILM package creates and schedules jobs to enforce policies for ADO. The EXECUTE_ILM() procedure provides this functionality, regardless of any previously-scheduled ILM jobs. All jobs are created and scheduled to run immediately; however, whether they are run immediately depends on the number of jobs queued with the scheduler.

You can use the EXECUTE_ILM procedure if you want more control when ILM jobs are performed, and do not want to wait until the next maintenance window.

The STOP_ILM procedure of the DBMS_ILM package stops all jobs, all running jobs, jobs based on a task Id, or a specific job.

The CUSTOMIZE_ILM procedure in the DBMS_ILM_ADMIN PL/SQL package enables you to customize settings for ADO, as shown in Example 5-9.

For example, you can set the values for the TBS_PERCENT_USED and TBS_PERCENT_FREE ILM parameters or set the ABS_JOBLIMIT ILM parameter. TBS_PERCENT_USED and TBS_PERCENT_FREE determine when data is moved based on tablespace quotas and ABS_JOBLIMIT sets the absolute number of concurrent ADO jobs.

You can also recreate objects with policies using the DBMS_METADATA PL/SQL package.

> **See Also:**
>
> - Managing ILM ADO Parameters for information about ILM ADO parameters
> - *Oracle Database PL/SQL Packages and Types Reference* for information about the DBMS_ILM, DBMS_ILM_ADMIN, and DBMS_METADATA packages

Example 5-9 Using CUSTOMIZE_ILM to customize ADO settings

```
SQL> BEGIN
  2  DBMS_ILM_ADMIN.CUSTOMIZE_ILM(DBMS_ILM_ADMIN.TBS_PERCENT_USED, 85);
  3  DBMS_ILM_ADMIN.CUSTOMIZE_ILM(DBMS_ILM_ADMIN.TBS_PERCENT_FREE, 25);
  4  END;
  5  /

SQL> BEGIN
  2  DBMS_ILM_ADMIN.CUSTOMIZE_ILM(DBMS_ILM_ADMIN.ABS_JOBLIMIT, 10);
  3  END;
  4  /
```

5.2.2.9 Using Views to Monitor Policies for ADO

You can view and monitor the policies for ADO that are associated with your database objects using the DBA_ILM* and USER_ILM* views, making it easier to change policies as needed.

- The DBA/USER_ILMDATAMOVEMENTPOLICIES view displays information specific to data movement related attributes of an ILM policy for ADO.
- The DBA/USER_ILMTASKS view displays the task Ids of the procedure EXECUTE_ILM. Every time a user invokes the procedure EXECUTE_ILM, a task Id is returned to track this particular invocation. A task Id is also generated to track periodic internal ILM tasks by the database. This view contains information about all ILM tasks for ADO.
- The DBA/USER_ILMEVALUATIONDETAILS view displays details on policies considered for a particular task. It also shows the name of the job that executes the policy in case the policy was selected for evaluation. In case the policy was not executed, this view also provides a reason.
- The DBA/USER_ILMOBJECTS view displays all the objects and policies for ADO in the database. Many objects inherit policies through their parent objects or because they were created in a particular tablespace. This view provides a mapping between the policies and objects. In the case of an inherited policy, this view also indicates the level from which policy is inherited.
- The DBA/USER_ILMPOLICIES view displays details about all the policies for ADO in the database.
- The DBA/USER_ILMRESULTS view displays information about data movement-related jobs for ADO in the database.

- The `DBA_ILMPARAMETERS` view displays information about ADO-related parameters.

> **See Also:**
>
> *Oracle Database Reference* for information about the `ILM` views

5.2.3 Limitations and Restrictions With ADO and Heat Map

The limitations and restrictions associated with ADO and Heat Map are discussed in this topic.

Limitations and restrictions associated with ADO and Heat Map include:

- Partition-level ADO and compression are supported for Temporal Validity except for row-level ADO policies that would compress rows that are past their valid time (access or modification).
- Partition-level ADO and compression are supported for in-database archiving if partitioned on the `ORA_ARCHIVE_STATE` column.
- Custom policies (user-defined functions) for ADO are not supported if the policies default at the tablespace level.
- ADO Row-level policies support the `OF NO MODIFICATION` condition type only.
- ADO does not perform checks for storage space in a target tablespace when using storage tiering.
- ADO is not supported on tables with object types or materialized views.
- ADO is not supported with index-organized tables or clusters.
- ADO concurrency (the number of simultaneous policy jobs for ADO) depends on the concurrency of the Oracle scheduler. If a policy job for ADO fails more than two times, then the job is marked disabled and the job must be manually enabled later.
- ADO has restrictions related to moving tables and table partitions.

> **See Also:**
>
> – *Oracle Database SQL Language Reference* for information about restrictions on moving tables
>
> – *Oracle Database SQL Language Reference* for information about restrictions on moving table partitions

5.3 Controlling the Validity and Visibility of Data in Oracle Database

You can control the validity and visibility of data in Oracle Database with In-Database Archiving and Temporal Validity.

This section contains the following topics:

- Using In-Database Archiving

- Using Temporal Validity
- Creating a Table With Temporal Validity
- Limitations and Restrictions With In-Database Archiving and Temporal Validity

5.3.1 Using In-Database Archiving

In-Database Archiving enables you to mark rows within a table as active or archived.

How In-Database Archiving Helps both Database and Applications Operations

These are the advantages of In-Database Archiving.

- With In-Database Archiving, you can store large volumes of archival data in the database without compromising application performance.
- Archived data can be compressed to help improve backup performance.
- Can be a performance accelerator for data searches.
  It narrows the scope of the search down to what is currently relevant, which may be a minority of the data volume in the targeted tables.
- Can reduce the downtime caused by application upgrades that involve data changes. Upgrade scripts can be written to process active and archived data separately. For example, an application can be made available again immediately after upgrades and data fixes have been applied to active data, since the archived data is invisible to users. Changes to the archived data can be deferred until later.

Active Versus Archival Data

For the purposes of In-Database Archiving, a row in a table exists in one of two states:

- ***Active*** – Rows subject to frequent querying are usually considered active. For example, in a business environment the sales data for the current and last quarter may be considered active.
- ***Archival*** – Rows containing data that is no longer essential to ongoing activity can implicitly be considered archival. They may for example contain data from transactions completed months or years ago which is stored for records retention and/or compliance purposes only. Such data is seldom read or modified. For example, the sales data for the last five fiscal years could be considered archival data.

Often the majority of the data in a database is inactive and therefore archival. Filtering out this data automatically in queries can significantly speed up query performance. Likewise, bulk updates (such as during application upgrades) also benefit by reduced downtime when irrelevant data is excluded. To set up this data filtering, you can enable the table for row archival storage, which adds the column `ORA_ARCHIVE_STATE` to the table. Archived rows are not ordinarily visible to queries, so these rows are not processed and are excluded from query results.

What is ORA_ARCHIVE_STATE?

`ORA_ARCHIVE_STATE` is the system column responsible for designating a given row as `active` or `archived`. It is a hidden column which does not have a user column number (`usrcol#`) in the dictionary. It is not visible when you `DESCRIBE` a table, or issue `SELECT *` queries on the table. However, this column can be explicitly specified in the select-list of a `SELECT` query, the modification list for an `UPDATE` query, and in predicates for both queries and DMLs.

> **Note:**
>
> You and the ILM practices of your organization must determine which rows in a row-archival table should be archived. In-Database Archiving does not perform such an analysis. By default, it sets each row in a row-archival table to active.

Enabling a new or Existing Table for Row Archiving

Include the `ROW ARCHIVAL` clause in a `CREATE TABLE` statement or use `ALTER TABLE` to set `ROW ARCHIVAL` on an existing table. For example:

```
CREATE TABLE scott.emp (EMPNO NOT NULL NUMBER(7), FULLNAME VARCHAR2(100), JOB
VARCHAR2(9), MGR NUMBER(7)) ROW ARCHIVAL;


ALTER TABLE scott.emp ROW ARCHIVAL;
```

You can also remove the `ROW ARCHIVAL` property from a table. This removes the `ORA_ARCHIVE_STATE` column from the table.

```
ALTER TABLE scott.emp NO ROW ARCHIVAL;
```

> **Note:**
>
> You cannot remove the row-archive attribute of a table during online redefinition.

These are some of the characteristics of the `ORA_ARCHIVE_STATE` column:

- It is a hidden column not implicitly reported by `SELECT *` queries and the `DESCRIBE` command.
- You can explicitly update the column.

Querying Row-Archive Enabled Tables

Note how `INSERT`, `UPDATE`, and `CREATE AS SELECT` work when used against row-archive enabled tables.

INSERT
The default value of the `ORA_ARCHIVE_STATE` in each row is “0," which implies that the row is considered active data until you change the column value to archived. Therefore, an INSERT into a row-archival table always sets new rows to active.

UPDATE

Updates can directly set the value of `ORA_ARCHIVE_STATE` in any row. The convention is to use either “0” or “1” to indicate `active` or `archived`, but you may update `ORA_ARCHIVE_STATE` to any valid VARCHAR2(4000) value.

CREATE TABLE AS SELECT
`CREATE TABLE AS SELECT` behavior depends upon whether or not each table in the selection is row-archival enabled and the structure of the columns in the `SELECT` list. (See the table below)

Note that "ORA_ARCHIVE_STATE" is not an Oracle reserved word and therefore is valid as the name of user column if row archival is not already enabled on the respective table.

INSERT INTO SELECT

An `INSERT INTO SELECT` where the source and target tables are row-archival enabled does not populate the target table's ORA_ARCHIVE_STATE column with the value of the corresponding column from the source table unless explicitly mapped in source and target table column lists. Instead, the default active row-archival state is set.

ALTER TABLE MOVE

If you move a table via `ALTER TABLE MOVE`, the row-archival state column of the table is preserved.

How CREATE TABLE AS SELECT Behavior Differs in Data Transfers Between Two Tables, Depending on Whether or not Both Tables are Row Archive Enabled

The table below shows the behavior of `CREATE TABLE AS SELECT` under different scenarios.

Table 5-3 ORA_ARCHIVE_STATE Attributes

Source Table Row-Archive Enabled?	Target Table Row-Archive Enabled?	Select List Structure	ORA_ARCHIVE_STATE is a User Column on Source Table	Behavior
No	No	Either `SELECT *`, or, includes `ORA_ARCHIVE_STATE` is explicitly in the SELECT list	Yes	Copies all rows to the target table with requested columns including `ORA_ARCHIVE_STATE` as a user column.
Yes	No	`SELECT *`	No	Copies selected rows depending on row visibility with all columns except the `ORA_ARCHIVE_STATE` column.
Yes	No	`ORA_ARCHIVE_STATE` is explicitly included in the SELECT list	Yes	Copies selected rows depending on row visibility with requested columns including `ORA_ARCHIVE_STATE` as a user column.
Yes	Yes	`SELECT *`	No	Copies all rows to the target table with the requested columns. Assigns '0' to the `ORA_ARCHIVE_STATE` column in all new rows in the target table.

Table 5-3 (Cont.) ORA_ARCHIVE_STATE Attributes

Source Table Row-Archive Enabled?	Target Table Row-Archive Enabled?	Select List Structure	ORA_ARCHIVE_STATE is a User Column on Source Table	Behavior
Yes	Yes	The `ORA_ARCHIVE_STATE` column is explicitly included in a `SELECT` list	No	Copies selected rows with requested columns (depending on row visibility). Propagates the existing row archival state of each row from the source table to the target table.
Yes	Yes	`ORA_ARCHIVE_STATE` is not explicitly included in the `SELECT`	No	Copies the selected rows with the selected columns (depending on the row visibility). Assigns '0' to the new `ORA_ARCHIVE_STATE` column in all new rows in the target table.
No	Yes	Either `SELECT *`, or, `ORA_ARCHIVE_STATE` is explicitly included in the SELECT list	Yes	**Important:** Returns

Setting `ROW ARCHIVAL VISIBILITY` for Session Level Visibility Control

By default, only active data is visible to users and applications in row-archival enabled tables. But you can modify this per session and make archived rows visible (or back to invisible) within the current session.

The following statement enables session-level visibility only to those rows that are active. Within the session, all user queries against row-archival enabled tables are subject to active data filtering by the system.

```
ALTER SESSION SET ROW ARCHIVAL VISIBILITY = ACTIVE;
```

The statement below enable session level row-archival visibility for all rows. Within the session, all user queries against row-archival enabled tables are affected by the row archival column value.

```
ALTER SESSION SET ROW ARCHIVAL VISIBILITY = ALL;
```

See Also:

- In the *PL/SQL Packages and Types Reference*, the `DBMS_ILM` package contains active and archived state constants for the `ORA_ARCHIVE_STATE` column. These are represented by '`0`' and '`1`' respectively.

```
archive_state_active constant varchar2(1) := '0';
archive_state_archived constant varchar2(1) := '1';
```

  DBMS_ILM also provides the function `ARCHIVESTATENAME` for returning the current value of `ORA_ARCHIVE_STATE`.

```
function archiveStateName(value in varchar2) return varchar2;
```

Input Parameters	Description
value	The string value for which the archive state name is to be returned
return	“archive_state_active” for “0” “archive_state_archived” for others

- The *SQL Language Reference* describes the use of the `ROW_ARCHIVAL` clause in `ALTER TABLE` and `CREATE TABLE` statements.
  - `ALTER TABLE [schema.]table [alter_table_properties | … ] [row_archival_clause]`
  - `CREATE TABLE [schema.]table [(relational properties)] [table properties] table_properties::= [column_properties ] [ table_partitioning_clauses ] … [row_archival_clause] row_archival_clause ::= [NO] ROW ARCHIVAL`

Example 5-10 Using In-Database Archiving

```
/* Set visibility to ACTIVE to display only active rows of a table.*/
ALTER SESSION SET ROW ARCHIVAL VISIBILITY = ACTIVE;

CREATE TABLE employees_indbarch
 (employee_id NUMBER(6) NOT NULL,
  first_name VARCHAR2(20), last_name VARCHAR2(25) NOT NULL,
  email VARCHAR2(25) NOT NULL, phone_number VARCHAR2(20),
  hire_date DATE NOT NULL, job_id VARCHAR2(10) NOT NULL, salary  NUMBER(8,2),
  commission_pct NUMBER(2,2), manager_id NUMBER(6), department_id NUMBER(4)) ROW ARCHIVAL;

/* Show all the columns in the table, including hidden columns */
SELECT SUBSTR(COLUMN_NAME,1,22) NAME, SUBSTR(DATA_TYPE,1,20) DATA_TYPE, COLUMN_ID AS COL_ID,
  SEGMENT_COLUMN_ID AS SEG_COL_ID, INTERNAL_COLUMN_ID AS INT_COL_ID, HIDDEN_COLUMN, CHAR_LENGTH
  FROM USER_TAB_COLS WHERE TABLE_NAME='EMPLOYEES_INDBARCH';

NAME                   DATA_TYPE               COL_ID SEG_COL_ID INT_COL_ID HID CHAR_LENGTH
---------------------- -------------------- ---------- ---------- ---------- --- -----------
ORA_ARCHIVE_STATE      VARCHAR2                                 1          1 YES        4000
EMPLOYEE_ID            NUMBER                        1          2          2 NO            0
FIRST_NAME             VARCHAR2                      2          3          3 NO           20
LAST_NAME              VARCHAR2                      3          4          4 NO           25
```

```
EMAIL                VARCHAR2                4          5          5 NO          25
PHONE_NUMBER         VARCHAR2                5          6          6 NO          20
HIRE_DATE            DATE                    6          7          7 NO           0
JOB_ID               VARCHAR2                7          8          8 NO          10
SALARY               NUMBER                  8          9          9 NO           0
COMMISSION_PCT       NUMBER                  9         10         10 NO           0
MANAGER_ID           NUMBER                 10         11         11 NO           0
DEPARTMENT_ID        NUMBER                 11         12         12 NO           0

/* Insert some data into the table */
INSERT INTO employees_indbarch(employee_id, first_name, last_name, email,
  hire_date, job_id, salary, manager_id, department_id)
  VALUES (251, 'Scott', 'Tiger', 'scott.tiger@example.com', '21-MAY-2009',
 'IT_PROG', 50000, 103, 60);

INSERT INTO employees_indbarch(employee_id, first_name, last_name, email,
  hire_date, job_id, salary, manager_id, department_id)
  VALUES (252, 'Jane', 'Lion', 'jane.lion@example.com', '11-JUN-2009',
  'IT_PROG', 50000, 103, 60);

/* Decrease the ORA_ARCHIVE_STATE column size to improve formatting in queries */
COLUMN ORA_ARCHIVE_STATE FORMAT a18;

/* The default value for ORA_ARCHIVE_STATE is '0', which means active */
SELECT employee_id, ORA_ARCHIVE_STATE FROM employees_indbarch;

EMPLOYEE_ID ORA_ARCHIVE_STATE
----------- ------------------
        251 0
        252 0

/* Insert a value into ORA_ARCHIVE_STATE to set the record to inactive status*/
UPDATE employees_indbarch SET ORA_ARCHIVE_STATE = '1' WHERE employee_id = 252;

/* Only active records are in the following query */
SELECT employee_id, ORA_ARCHIVE_STATE FROM employees_indbarch;

EMPLOYEE_ID ORA_ARCHIVE_STATE
----------- ------------------
        251 0

/* Set visibility to ALL to display all records */
ALTER SESSION SET ROW ARCHIVAL VISIBILITY = ALL;

SELECT employee_id, ORA_ARCHIVE_STATE FROM employees_indbarch;

EMPLOYEE_ID ORA_ARCHIVE_STATE
----------- ------------------
        251 0
        252 1
```

5.3.2 Using Temporal Validity

Temporal Validity enables you to track time periods for real world validity. Valid times can be set by users and applications for data, and data can be selected by a specified valid time, or a valid time range.

Applications often note the validity (or effectivity) of a fact recorded in a database with dates or timestamps that are relevant to the management of a business. For example, the hire-date of an employee in a human resources (HR) application, which determines the effective date of coverage in the insurance industry, is a valid date. This date is in contrast to the date or time at

which the employee record was entered in the database. The former temporal attribute (hire-date) is called the valid time (VT) while the latter (date entered into the database) is called the transaction time (TT). While the valid time is usually controlled by the user, the transaction-time is system-managed.

For ILM, the valid time attributes can signify when a fact is valid in the business world and when it is not. Using valid time attributes, a query could just show rows that are currently valid, while not showing rows that contains facts that are not currently valid, such as a closed order or a future hire.

Concepts that are integral to valid time temporal modeling include:

- Valid time

  This is a user-defined representation of time. Examples of a valid time include project start and finish dates, and employee hire and termination dates.

- Tables with valid-time semantics

  These tables have one or more dimensions of user-defined time, each of which has a start and an end.

- Valid-time flashback queries

  This is the ability to do as-of and versions queries using a valid-time dimension.

A valid-time period consists of two date-time columns specified in the table definition. You can add a valid-time period by explicitly adding columns, or the columns can be created automatically. A valid-time period can be added during the create table or alter table process.

To support session level visibility control for temporal table queries, the `DBMS_FLASHBACK_ARCHIVE` PL/SQL package provides the `ENABLE_AT_VALID_TIME` procedure. To execute the procedure, you need the required system and object privileges.

The following PL/SQL procedure sets the valid time visibility as of the given time.

```
SQL> EXECUTE DBMS_FLASHBACK_ARCHIVE.enable_at_valid_time
          ('ASOF', '31-DEC-12 12.00.01 PM');
```

The following PL/SQL procedure sets the visibility of temporal data to currently valid data within the valid time period at the session level.

```
SQL> EXECUTE DBMS_FLASHBACK_ARCHIVE.enable_at_valid_time('CURRENT');
```

The following procedure sets the visibility of temporal data to the full table, which is the default temporal table visibility.

```
SQL> EXECUTE DBMS_FLASHBACK_ARCHIVE.enable_at_valid_time('ALL');
```

See Also:

- *Oracle Database Development Guide* for information about Oracle Temporal
- *Oracle Database PL/SQL Packages and Types Reference* for information about the `DBMS_FLASHBACK_ARCHIVE` package
- *Oracle Database SQL Language Reference* for information about using the `CREATE TABLE` or `ALTER TABLE` to initiate valid-time temporal modeling
- *Oracle Database Reference* for information about views used to monitor table information

5.3.3 Creating a Table With Temporal Validity

The example in this topic shows how to create a table with temporal validity.

Example 5-11 shows the use of temporal validity.

> **Live SQL:**
>
> View and run a related example on Oracle Live SQL at *Oracle Live SQL: Creating a Table with Temporal Validity*.

Example 5-11 Creating a table with temporal validity

```
/* Create a time with an employee tracking timestamp */
/* using the specified columns*/
CREATE TABLE employees_temp (
       employee_id NUMBER(6) NOT NULL, first_name VARCHAR2(20), last_name VARCHAR2(25) NOT NULL,
       email VARCHAR2(25) NOT NULL, phone_number VARCHAR2(20), hire_date DATE NOT NULL,
       job_id VARCHAR2(10) NOT NULL, salary  NUMBER(8,2), commission_pct NUMBER(2,2),
       manager_id NUMBER(6), department_id NUMBER(4),
       PERIOD FOR emp_track_time);

DESCRIBE employees_temp

 Name                                                      Null?    Type
 --------------------------------------------------------- -------- ---------------
 EMPLOYEE_ID                                               NOT NULL NUMBER(6)
 FIRST_NAME                                                         VARCHAR2(20)
 LAST_NAME                                                 NOT NULL VARCHAR2(25)
 EMAIL                                                     NOT NULL VARCHAR2(25)
 PHONE_NUMBER                                                       VARCHAR2(20)
 HIRE_DATE                                                 NOT NULL DATE
 JOB_ID                                                    NOT NULL VARCHAR2(10)
 SALARY                                                             NUMBER(8,2)
 COMMISSION_PCT                                                     NUMBER(2,2)
 MANAGER_ID                                                         NUMBER(6)
 DEPARTMENT_ID                                                      NUMBER(4)

SQL> SELECT SUBSTR(COLUMN_NAME,1,22) NAME, SUBSTR(DATA_TYPE,1,28) DATA_TYPE, COLUMN_ID AS COL_ID,
     SEGMENT_COLUMN_ID AS SEG_COL_ID, INTERNAL_COLUMN_ID AS INT_COL_ID, HIDDEN_COLUMN
     FROM USER_TAB_COLS WHERE TABLE_NAME='EMPLOYEES_TEMP';

NAME                   DATA_TYPE                    COL_ID SEG_COL_ID INT_COL_ID HID
---------------------- ---------------------------- ------ ---------- ---------- ---
EMP_TRACK_TIME_START   TIMESTAMP(6) WITH TIME ZONE                   1          1 YES
EMP_TRACK_TIME_END     TIMESTAMP(6) WITH TIME ZONE                   2          2 YES
EMP_TRACK_TIME         NUMBER                                                   3 YES
EMPLOYEE_ID            NUMBER                                 1          3          4 NO
FIRST_NAME             VARCHAR2                               2          4          5 NO
LAST_NAME              VARCHAR2                               3          5          6 NO
EMAIL                  VARCHAR2                               4          6          7 NO
PHONE_NUMBER           VARCHAR2                               5          7          8 NO
HIRE_DATE              DATE                                   6          8          9 NO
JOB_ID                 VARCHAR2                               7          9         10 NO
SALARY                 NUMBER                                 8         10         11 NO
COMMISSION_PCT         NUMBER                                 9         11         12 NO
MANAGER_ID             NUMBER                                10         12         13 NO
```

```
DEPARTMENT_ID          NUMBER                          11         13         14 NO

/* Insert/update/delete with specified values for time columns */
INSERT INTO employees_temp(emp_track_time_start, emp_track_time_end, employee_id, first_name,
   last_name, email, hire_date, job_id, salary, manager_id, department_id)
   VALUES (TIMESTAMP '2009-06-01 12:00:01 Europe/Paris',
           TIMESTAMP '2012-11-30 12:00:01 Europe/Paris', 251, 'Scott', 'Tiger',
          'scott.tiger@example.com', DATE '2009-05-21', 'IT_PROG', 50000, 103, 60);

INSERT INTO employees_temp(emp_track_time_start, emp_track_time_end, employee_id, first_name,
     last_name, email, hire_date, job_id, salary, manager_id, department_id)
     VALUES (TIMESTAMP '2009-06-01 12:00:01 Europe/Paris',
             TIMESTAMP '2012-12-31 12:00:01 Europe/Paris', 252, 'Jane', 'Lion',
             'jane.lion@example.com', DATE '2009-06-11', 'IT_PROG', 50000, 103, 60);

UPDATE employees_temp set salary = salary + salary * .05
       WHERE emp_track_time_start <= TIMESTAMP '2009-06-01 12:00:01 Europe/Paris';

SELECT employee_id, SALARY FROM employees_temp;

EMPLOYEE_ID     SALARY
----------- ----------
        251      52500
        252      52500

/* No rows are deleted for the following statement because no records */
/* are in the specified track time. */
DELETE employees_temp WHERE emp_track_time_end < TIMESTAMP '2001-12-31 12:00:01 Europe/Paris';

0 rows deleted.

/* Show rows that are in a specified time period */
SELECT employee_id FROM employees_temp
       WHERE emp_track_time_start > TIMESTAMP '2009-05-31 12:00:01 Europe/Paris' AND
             emp_track_time_end < TIMESTAMP '2012-12-01 12:00:01 Europe/Paris';

EMPLOYEE_ID
-----------
        251

/* Show rows that are in a specified time period */
SELECT employee_id FROM employees_temp AS OF PERIOD FOR
       emp_track_time TIMESTAMP '2012-12-01 12:00:01 Europe/Paris';

EMPLOYEE_ID
-----------
        252
```

5.3.4 Limitations and Restrictions With In-Database Archiving and Temporal Validity

This topic lists the limitations and restrictions associated with In-Database Archiving and Temporal Validity.

The limitations and restrictions include:

- ILM is not supported with OLTP table compression for Temporal Validity. Segment-level ILM and compression is supported if partitioned on the end-time columns.
- ILM is not supported with OLTP table compression for in-database archiving. Segment-level ILM and compression is supported if partitioned on the `ORA_ARCHIVE_STATE` column.

5.4 Implementing an ILM System Manually Using Partitioning

You can manually implement an Information Lifecycle Management (ILM) system using partitioning.

Example 5-12 illustrates how to manually create storage tiers and partition a table across those storage tiers and then setup a virtual private database (VPD) policy on that database to restrict access to the online archive tier data.

> **See Also:**
>
> - *Oracle Database SQL Language Reference* for information about the `CREATE TABLE` SQL statement
> - *Oracle Database PL/SQL Packages and Types Reference* for information about the `DBMS_RLS` package

Example 5-12 Manually implementing an ILM system

```
REM Setup the tablespaces for the data

REM These tablespaces would be placed on a High Performance Tier
CREATE SMALLFILE TABLESPACE q1_orders DATAFILE 'q1_orders'
  SIZE 2M AUTOEXTEND ON NEXT 1M MAXSIZE UNLIMITED LOGGING
  EXTENT MANAGEMENT LOCAL SEGMENT SPACE MANAGEMENT AUTO ;

CREATE SMALLFILE TABLESPACE q2_orders DATAFILE 'q2_orders'
  SIZE 2M AUTOEXTEND ON NEXT 1M MAXSIZE UNLIMITED LOGGING
  EXTENT MANAGEMENT LOCAL SEGMENT SPACE MANAGEMENT AUTO ;

CREATE SMALLFILE TABLESPACE q3_orders DATAFILE 'q3_orders'
  SIZE 2M AUTOEXTEND ON NEXT 1M MAXSIZE UNLIMITED LOGGING
  EXTENT MANAGEMENT LOCAL SEGMENT SPACE MANAGEMENT AUTO ;

CREATE SMALLFILE TABLESPACE q4_orders DATAFILE 'q4_orders'
  SIZE 2M AUTOEXTEND ON NEXT 1M MAXSIZE UNLIMITED LOGGING
  EXTENT MANAGEMENT LOCAL SEGMENT SPACE MANAGEMENT AUTO ;

REM These tablespaces would be placed on a Low Cost Tier
CREATE SMALLFILE TABLESPACE "2006_ORDERS" DATAFILE '2006_orders'
  SIZE 5M AUTOEXTEND ON NEXT 10M MAXSIZE UNLIMITED LOGGING
  EXTENT MANAGEMENT LOCAL SEGMENT SPACE MANAGEMENT AUTO ;

CREATE SMALLFILE TABLESPACE "2005_ORDERS"  DATAFILE '2005_orders'
  SIZE 5M AUTOEXTEND ON NEXT 10M MAXSIZE UNLIMITED LOGGING
  EXTENT MANAGEMENT LOCAL SEGMENT SPACE MANAGEMENT AUTO ;

REM These tablespaces would be placed on the Online Archive Tier
CREATE SMALLFILE TABLESPACE "2004_ORDERS" DATAFILE '2004_orders'
  SIZE 5M AUTOEXTEND ON NEXT 10M MAXSIZE UNLIMITED LOGGING
  EXTENT MANAGEMENT LOCAL SEGMENT SPACE MANAGEMENT AUTO ;

CREATE SMALLFILE TABLESPACE old_orders DATAFILE 'old_orders'
  SIZE 15M AUTOEXTEND ON NEXT 10M MAXSIZE UNLIMITED LOGGING
  EXTENT MANAGEMENT LOCAL SEGMENT SPACE MANAGEMENT AUTO ;
```

```
REM Now create the Partitioned Table
CREATE TABLE allorders (
    prod_id       NUMBER       NOT NULL,
    cust_id       NUMBER       NOT NULL,
    time_id       DATE         NOT NULL,
    channel_id    NUMBER       NOT NULL,
    promo_id      NUMBER       NOT NULL,
    quantity_sold NUMBER(10,2) NOT NULL,
    amount_sold   NUMBER(10,2) NOT NULL)
 --
 -- table wide physical specs
 --
  PCTFREE 5 NOLOGGING
 --
 -- partitions
 --
 PARTITION BY RANGE (time_id)
  ( partition allorders_pre_2004 VALUES LESS THAN
     (TO_DATE('2004-01-01 00:00:00'
             ,'SYYYY-MM-DD HH24:MI:SS'
             ,'NLS_CALENDAR=GREGORIAN'
             )) TABLESPACE old_orders,
    partition allorders_2004 VALUES LESS THAN
     (TO_DATE('2005-01-01 00:00:00'
             ,'SYYYY-MM-DD HH24:MI:SS'
             ,'NLS_CALENDAR=GREGORIAN'
             )) TABLESPACE "2004_ORDERS",
    partition allorders_2005 VALUES LESS THAN
     (TO_DATE('2006-01-01 00:00:00'
             ,'SYYYY-MM-DD HH24:MI:SS'
             ,'NLS_CALENDAR=GREGORIAN'
             )) TABLESPACE "2005_ORDERS",
    partition allorders_2006 VALUES LESS THAN
     (TO_DATE('2007-01-01 00:00:00'
             ,'SYYYY-MM-DD HH24:MI:SS'
             ,'NLS_CALENDAR=GREGORIAN'
             )) TABLESPACE "2006_ORDERS",
    partition allorders_q1_2007 VALUES LESS THAN
     (TO_DATE('2007-04-01 00:00:00'
             ,'SYYYY-MM-DD HH24:MI:SS'
             ,'NLS_CALENDAR=GREGORIAN'
             )) TABLESPACE q1_orders,
    partition allorders_q2_2007 VALUES LESS THAN
     (TO_DATE('2007-07-01 00:00:00'
             ,'SYYYY-MM-DD HH24:MI:SS'
             ,'NLS_CALENDAR=GREGORIAN'
             )) TABLESPACE q2_orders,
    partition allorders_q3_2007 VALUES LESS THAN
     (TO_DATE('2007-10-01 00:00:00'
             ,'SYYYY-MM-DD HH24:MI:SS'
             ,'NLS_CALENDAR=GREGORIAN'
             )) TABLESPACE q3_orders,
    partition allorders_q4_2007 VALUES LESS THAN
     (TO_DATE('2008-01-01 00:00:00'
             ,'SYYYY-MM-DD HH24:MI:SS'
             ,'NLS_CALENDAR=GREGORIAN'
             )) TABLESPACE q4_orders);

ALTER TABLE allorders ENABLE ROW MOVEMENT;

REM Here is another example using INTERVAL partitioning
```

```
REM These tablespaces would be placed on a High Performance Tier
CREATE SMALLFILE TABLESPACE cc_this_month DATAFILE 'cc_this_month'
  SIZE 2M AUTOEXTEND ON NEXT 1M MAXSIZE UNLIMITED LOGGING
  EXTENT MANAGEMENT LOCAL SEGMENT SPACE MANAGEMENT AUTO ;

CREATE SMALLFILE TABLESPACE cc_prev_month DATAFILE 'cc_prev_month'
  SIZE 2M AUTOEXTEND ON NEXT 1M MAXSIZE UNLIMITED LOGGING
  EXTENT MANAGEMENT LOCAL SEGMENT SPACE MANAGEMENT AUTO ;

REM These tablespaces would be placed on a Low Cost Tier
CREATE SMALLFILE TABLESPACE cc_prev_12mth DATAFILE 'cc_prev_12'
  SIZE 2M AUTOEXTEND ON NEXT 1M MAXSIZE UNLIMITED LOGGING
  EXTENT MANAGEMENT LOCAL SEGMENT SPACE MANAGEMENT AUTO ;

REM These tablespaces would be placed on the Online Archive Tier
CREATE SMALLFILE TABLESPACE cc_old_tran DATAFILE 'cc_old_tran'
  SIZE 2M AUTOEXTEND ON NEXT 1M MAXSIZE UNLIMITED LOGGING
  EXTENT MANAGEMENT LOCAL SEGMENT SPACE MANAGEMENT AUTO ;

REM Credit Card Transactions where new partitions
REM are automatically placed on the high performance tier
CREATE TABLE cc_tran (
    cc_no       VARCHAR2(16) NOT NULL,
    tran_dt     DATE         NOT NULL,
    entry_dt    DATE         NOT NULL,
    ref_no      NUMBER       NOT NULL,
    description VARCHAR2(30) NOT NULL,
    tran_amt    NUMBER(10,2) NOT NULL)
 --
 -- table wide physical specs
 --
 PCTFREE 5 NOLOGGING
 --
 -- partitions
 --
 PARTITION BY RANGE (tran_dt)
 INTERVAL (NUMTOYMINTERVAL(1,'month') ) STORE IN (cc_this_month )
  ( partition very_old_cc_trans VALUES LESS THAN
     (TO_DATE('1999-07-01 00:00:00'
             ,'SYYYY-MM-DD HH24:MI:SS'
             ,'NLS_CALENDAR=GREGORIAN'
             )) TABLESPACE cc_old_tran ,
    partition old_cc_trans VALUES LESS THAN
     (TO_DATE('2006-07-01 00:00:00'
             ,'SYYYY-MM-DD HH24:MI:SS'
             ,'NLS_CALENDAR=GREGORIAN'
             )) TABLESPACE cc_old_tran ,
    partition last_12_mths VALUES LESS THAN
     (TO_DATE('2007-06-01 00:00:00'
             ,'SYYYY-MM-DD HH24:MI:SS'
             ,'NLS_CALENDAR=GREGORIAN'
             )) TABLESPACE cc_prev_12mth,
    partition recent_cc_trans VALUES LESS THAN
    (TO_DATE('2007-07-01 00:00:00'
            ,'SYYYY-MM-DD HH24:MI:SS'
            ,'NLS_CALENDAR=GREGORIAN'
            )) TABLESPACE cc_prev_month,
    partition new_cc_tran VALUES LESS THAN
     (TO_DATE('2007-08-01 00:00:00'
             ,'SYYYY-MM-DD HH24:MI:SS'
             ,'NLS_CALENDAR=GREGORIAN'
             )) TABLESPACE cc_this_month);
```

```
REM Create a Security Policy to allow user SH to see all credit card data,
REM PM only sees this years data,
REM and all other uses cannot see the credit card data

CREATE OR REPLACE FUNCTION ilm_seehist
  (oowner IN VARCHAR2, ojname IN VARCHAR2)
   RETURN VARCHAR2 AS con VARCHAR2 (200);
BEGIN
  IF SYS_CONTEXT('USERENV','CLIENT_INFO') = 'SH'
  THEN -- sees all data
    con:= '1=1';
  ELSIF SYS_CONTEXT('USERENV','CLIENT_INFO') = 'PM'
  THEN -- sees only data for 2007
    con := 'time_id > ''31-Dec-2006''';
  ELSE
    -- others nothing
    con:= '1=2';
  END IF;
  RETURN (con);
END ilm_seehist;
/
```

5.5 Managing ILM Heat Map and ADO with Oracle Enterprise Manager

You can manage Heat Map and Automatic Data Optimization with Oracle Enterprise Manager Cloud Control.

This section contains the following topics:

- Accessing the Database Administration Menu
- Viewing Automatic Data Optimization Activity at the Tablespace Level
- Viewing the Segment Activity Details of Any Tablespace
- Viewing the Segment Activity Details of Any Object
- Viewing the Segment Activity History of Any Object
- Searching Segment Activity in Automatic Data Optimization
- Viewing Policies for a Segment
- Disabling Background Activity
- Changing Execution Frequency of Background Automatic Data Optimization
- Viewing Policy Executions In the Last 24 Hours
- Viewing Objects Moved In Last 24 Hours
- Viewing Policy Details
- Viewing Objects Associated With a Policy
- Evaluating a Policy Before Execution
- Executing a Single Policy
- Stopping a Policy Execution
- Viewing Policy Execution History

> **See Also:**
>
> - Displaying Heat Map Tracking Data With Views and Using Views to Monitor Policies for ADO for information about views available for displaying Heat Map and ADO policy details
> - *Oracle Enterprise Manager Cloud Control Administrator's Guide* for information about managing Oracle Enterprise Manager Cloud Control

5.5.1 Accessing the Database Administration Menu

To access the **Administration** menu for a database:

1. Log in to Oracle Enterprise Manager Cloud Control.
2. Select **Databases** under the **Targets** menu.
3. Click the database name in the list.
4. The **Administration** menu appears on the database home page.

5.5.2 Viewing Automatic Data Optimization Activity at the Tablespace Level

To monitor the activity of the top 100 tablespaces selected by size in the database, follow these steps:

1. From the **Administration** menu, choose **Storage**, then select **Information Lifecycle Management**.

   Enterprise Manager displays the Information Lifecycle Management page.
2. On the Information Lifecycle Management page, you can view the Top 100 Tablespace Activity Heat Map based on the last access time, last write time, last full scan time, and last look up scan time.

   By default, the size of each box in the heat map represents a tablespace within a heat map and is determined by tablespace size. You can then use Information Lifecycle Management to drill down from tablespace to object to segment level heat maps.

5.5.3 Viewing the Segment Activity Details of Any Tablespace

To view segment activity details of any tablespace, follow these steps:

1. From the **Administration** menu, choose **Storage**, then select I**nformation Lifecycle Management**.

   Enterprise Manager displays the Information Lifecycle Management page.
2. On the Information Lifecycle Management page, click the **Show Additional** button.

   Enterprise Manager displays a dialog box that you can use to search for the segment activity details of any tablespace.
3. On the dialog box, enter the **Tablespace Name** and click the **Search** button.

   Enterprise Manager displays the Segment Activity details for the tablespace.

4. Click the **Edit Tablespace Policy** button to display the Edit Tablespace page with the ADO tab selected. You can create a policy for the tablespace that allows it to be compressed or moved.

5.5.4 Viewing the Segment Activity Details of Any Object

To view segment activity details of any object, follow these steps:

1. From the **Administration** menu, choose **Storage**, then select **Information Lifecycle Management**.

   Enterprise Manager displays the Information Lifecycle Management page.

2. Move the cursor over any of the boxes within the Database level heat map where each box represents a tablespace. Click the **Tablespace** to which the object you want to view belongs.

   Enterprise Manager displays the Tablespace level heat map for the 100 largest objects belonging to the tablespace. The Segment Activity tables displays the Segment Activity details for the 100 largest objects.

3. On the Information Lifecycle Management page, click the **Show Additional** button.

   Enterprise Manager displays a dialog box that you can use to search for the segment activity details of any object belonging to the tablespace.

4. On the dialog box, enter the **Schema Name** and **Object Name** and click **Search**.

   Enterprise Manager displays the Segment Activity details for the object.

5. Click the **Edit Object Policy** button to display the Edit Object page with the ADO tab selected. You can create a policy for the object that allows it to be compressed or moved.

5.5.5 Viewing the Segment Activity History of Any Object

To view the segment activity history of any object, follow these steps:

1. From the **Administration** menu, choose **Storage**, then select **Information Lifecycle Management**.

   Enterprise Manager displays the Information Lifecycle Management page.

2. Move the cursor over any of the boxes within the Database level heat map where each box represents a tablespace. Click the **Tablespace** to which the object you want to view belongs.

   Enterprise Manager displays the Tablespace level heat map for the 100 largest objects belonging to the tablespace. The Segment Activity tables displays the Segment Activity details for the 100 largest objects.

3. Select the object in the Segment Activity details table and click the **Activity History** button.

   Enterprise Manager displays the Edit Object page with the ADO tab selected. The ADO tab displays a list of policies and the Segment Access history.

4. You can select a segment, change the date range to be the last 60 days, select the **Daily** option, and clicks the **Refresh** button to display the Segment Access History for the object for the last 60 days.

5.5.6 Searching Segment Activity in Automatic Data Optimization

To search for segment activities during different time durations in Automatic Data Optimization, follow these steps:

1. From the **Administration** menu, choose **Storage**, then select **Information Lifecycle Management**.

   Enterprise Manager displays the Information Lifecycle Management page.

2. On the Information Lifecycle Management page, click any of the boxes in a heat map. From the initial display which is the database level, click a box in the heat map to display the 100 largest objects in that tablespace. You can then click a heat map box to see the heat map of the 100 largest segments in that object.

3. Enter a timestamp that is one year ago for the Last Access Time and then click **Go**. A list of segments that have not been accessed or modified in the last year are displayed. The segments are sorted in descending order by segment size.

   On the object level heat map, you can search for a specific segment based on Tablespace, Name, Partition, Type, Last Access Time, Last Write Time, Last Full Scan Time, and Last Look Up Scan Time.

   You can select a row (segment) and view the row activity for that segment clicking the **Policies** column to view the policies associated with the segment.

5.5.7 Viewing Policies for a Segment

To view the policies associated with a segment, follow these steps:

1. From the **Administration** menu, choose **Storage**, then select **Information Lifecycle Management**.

   Enterprise Manager displays the Information Lifecycle Management page.

2. On the Information Lifecycle Management page, click **Go**. If the segments in the search results have policies associated with them, the count is displayed in the Policies column and is non-zero. Move your mouse over the count to view the associated policies with the segment, including inherited policies. If the count is zero then no policies are associated with the segment.

   From the Database level heat map drill down to Tablespace level heat map. On the Tablespace level heat map Enterprise Manager displays the top 100 Objects belonging to the Tablespace. For each object, Enterprise Manager displays the count of policies in the column.

   From Tablespace level heat map select an object and drill down to the object level heat map. Enterprise Manager displays the Top 100 largest Segments belonging to the Object. For each segment, Enterprise Manager displays a count of policies in the Policies column.

5.5.8 Disabling Background Activity

To disable the Automatic Data Optimization background evaluation and scheduler, follow these steps:

For more information about the ILM ADO `ENABLED` parameter, refer to Managing ILM ADO Parameters.

1. From the **Administration** menu, choose **Storage**, then select **Information Lifecycle Management**.

Enterprise Manager displays the Information Lifecycle Management page.

2. On the Information Lifecycle Management page, click the **Policy** tab.
3. Click **Configure** in the Policy Execution Settings section.

   The Configure Policy Execution Settings dialog box is displayed.
4. Change the **Status** drop-down to **Disabled** and click **OK**.

   Enterprise Manager displays the Information Lifecycle Management page where the Status now shows Disabled on the Policy tab in the Policy Execution Settings section.

5.5.9 Changing Execution Frequency of Background Automatic Data Optimization

To change the execution frequency of the Information Lifecycle Management background evaluation and scheduler, follow these steps:

For more information about the ILM ADO `EXECUTION INTERVAL` parameter, refer to Managing ILM ADO Parameters.

1. From the **Administration** menu, choose **Storage**, then select **Information Lifecycle Management**.

   Enterprise Manager displays the Information Lifecycle Management page.
2. On the Information Lifecycle Management page, click the **Policy** tab.
3. Click **Configure** in the Policy Execution Settings section.

   The Configure Policy Execution Settings dialog box is displayed.
4. Change the value of the **Execution Interval (mins)** to a lower or higher number than what is currently displayed and then click **OK**.

   Enterprise Manager displays the Information Lifecycle Management page where the Execution Interval now shows the new value on the Policy tab under Policy Execution Settings.

5.5.10 Viewing Policy Executions In the Last 24 Hours

To view policies that were executed in the last 24 hours and to view what objects were moved or compressed with the execution of the policies, follow these steps:

1. From the **Administration** menu, choose **Storage**, then select **Information Lifecycle Management**.

   Enterprise Manager displays the Information Lifecycle Management page.
2. On the Information Lifecycle Management page, click the **Policy** tab.
3. Click the **Policies Completed** or the **Policies Failed** link on the **Policy execution summary for last 24 hours** row. Clicking either displays the execution history for the past 24 hours.

   The Policy Execution Details dialog box display, showing the execution details for the policies in the last 24 hours.

5.5.11 Viewing Objects Moved In Last 24 Hours

To view which objects were moved in the last 24 hours and which policies/jobs moved those objects, follow these steps:

1. From the **Administration** menu, choose **Storage**, then select **Information Lifecycle Management**.

   Enterprise Manager displays the Information Lifecycle Management page.
2. On the Information Lifecycle Management page, click the **Policy** tab.
3. On the **Policies execution summary for last 24 hours** row, click the **Objects Moved** link.

   The Policy Execution History dialog box displays, showing the execution history for the jobs and policies executed and the objects moved in the last 24 hours.

5.5.12 Viewing Policy Details

To view the details of a specific ADO policy, follow these steps:

1. From the **Administration** menu, choose **Storage**, then select **Information Lifecycle Management**.

   Enterprise Manager displays the Information Lifecycle Management page.
2. On the Information Lifecycle Management page, click the **Policy** tab.
3. To view policy details, click the policy name link in the policy table, or select any row in the policies table and then click the **View Policy Details** button.

5.5.13 Viewing Objects Associated With a Policy

To view the objects associated with a specific policy, follow these steps:

1. From the **Administration** menu, choose **Storage**, then select **Information Lifecycle Management**.

   Enterprise Manager displays the Information Lifecycle Management page.
2. On the Information Lifecycle Management page, click the **Policy** tab.
3. Click the count in the Objects column.

   The Objects associated with the Policy are displayed

5.5.14 Evaluating a Policy Before Execution

To evaluate a policy before executing the policy, follow these steps:

1. From the **Administration** menu, choose **Storage**, then select **Information Lifecycle Management**.

   Enterprise Manager displays the Information Lifecycle Management page.
2. On the Information Lifecycle Management page, click the **Policy** tab.
3. In the Evaluations region, click **Evaluate**.

   A dialog box displays giving you the choice of evaluating all policies in the database, or all policies affecting objects in a particular schema.
4. Enter a **Schema Name** and click **OK** to initiate the evaluation.

The evaluation dialog box closes and the evaluation is submitted. You can refresh the page or perform other Enterprise Manager tasks and then revisit the Policy Tab later. When you do, the **Completed** count in the **Evaluations** region is increased by 1.

5. When you click the **Completed** count link in the Evaluations region, a dialog box is displayed that lists all completed evaluations.
6. When you click the **Task ID** of the most recent evaluation, the Evaluation Details dialog box is displayed listing all objects currently included in the evaluation task that will either be compressed or moved if this evaluation is executed.
7. Click **OK** to include the list of objects in the execution. The Evaluation Details dialog box closes.
8. Select the row in the Evaluations table containing the most recent evaluation (top most row), then click **Execute**.

   The Execute Evaluation dialog box is displayed, again listing the objects that will be affected during execution.
9. Click **OK** to begin the execution.

   The Execute Evaluation dialog box closes. Eventually, execution results can be seen by clicking the **Completed** or **Failed** links in the Jobs or Policies regions under Policy Execution Summary for Last 24 Hours. Also, eventually the Evaluations Completed count is decreased by 1, as the task changes state from **INACTIVE** to **ACTIVE** to **COMPLETED**.

5.5.15 Executing a Single Policy

To execute a policy immediately on the objects associated with the Policy, follow these steps:

1. From the **Administration** menu, choose **Storage**, then select **Information Lifecycle Management**.

   Enterprise Manager displays the Information Lifecycle Management page.
2. On the Information Lifecycle Management page, click the **Policy** tab.
3. Select the policy and click **Execute Policy**.

   The Execute Policy dialog box is displayed, listing all objects that are evaluated by the selected policy. The dialog box also includes a **Hide**/**Show** button to display the `EXECUTE_ILM` commands to be executed. Only objects with this policy enabled are included.

5.5.16 Stopping a Policy Execution

To stop a policy execution, follow these steps:

1. From the **Administration** menu, choose **Storage**, then select **Information Lifecycle Management**.

   Enterprise Manager displays the Information Lifecycle Management page.
2. On the Information Lifecycle Management page, click the **Policy** tab.
3. In the Jobs region under Policy Execution Summary for Last 24 Hours, click the **In Progress** link. This step assumes there is at least one task or job in progress.

   A dialog box displays listing all currently executing tasks.
4. Click the **Jobs** link for one of the tasks listed in the table.

   A dialog box displays listing details about the job(s) running as part of the task.

5. Clicks **OK**.

   The Jobs Details dialog box closes.

6. Select a row in the table and click **Stop Execution**.

   A dialog box displays confirmation of the stop execution process.

7. Click **OK**.

   The confirmation dialog box is dismissed.

5.5.17 Viewing Policy Execution History

To view the execution history for a specific policy, follow these steps:

1. From the **Administration** menu, choose **Storage**, then select **Information Lifecycle Management**.

   Enterprise Manager displays the Information Lifecycle Management page.

2. On the Information Lifecycle Management page, click the **Policy** tab.

3. Select the policy and click **Execution History**.

   The Policy Execution History dialog box displays, showing the execution history for the selected policy. The details include the job information and the objects that were moved or compressed.

5.6 Using Automatic Storage Compression

Use Automatic Storage Compression to speed up initial direct loads and DMLs, and scans on the final dataset.

Automatic Storage Compression allows loads and post-load DMLs to operate on uncompressed format. When DML activity has subsided, compression is performed gradually in a background AutoTask. The final compressed data will not have fragmentation caused by DMLs and will be ideal for scans.

Sorting Behavior in Automatic Storage Compression

- If clustering directive is from the `CLUSTERING` clause in `CREATE TABLE` or `ALTER TABLE`:
  - Each direct load performs a sort at the statement-level.
  - Automatic Storage Compression preserves the sort-order when it moves the 1 GB chunks.
- If clustering has been enabled using the `DBMS_AUTO_CLUSTERING` PL/SQL package:
  - The direct load does not sort.
  - Auto Storage Compression sorts the rows when moving each 1 GB chunk.

Prerequisites for Automatic Storage Compression

- The table must be in a tablespace with these properties:
  - `SEGMENT SPACE MANAGEMENT AUTO`
  - `AUTOALLOCATE`
- In the PDB, set `HEAT_MAP=ON`.
- The table must be an HCC table.

To Enable Automatic Storage Compression

1. Execute `DBMS_ILM_ADMIN.ENABLE_AUTO_OPTIMIZE` at the PDB level.
2. Direct load the uncompressed data into a HCC table.
3. Check that all rows are uncompressed and note the segment size.
4. After the time specified in minutes by `AUTO_OPTIMIZE_INACTIVITY_THRESHOLD` with no DML activity, Automatic Storage Compression starts running as a background AutoTask.
5. Rows are compressed incrementally.
   You can monitor the incremental progress by checking the "`Auto compression data moved`" system statistic, which increases over time.
6. Eventually, "`Auto compression data moved`" matches the initial uncompressed segment size. This indicates that Automatic Storage Compression has completed. Check the segment size against the original to confirm that it has been compressed.

> **Note:**
>
> If clustering was specified using the attribute clustering DDL:
>
> - Sorting occurs during the direct load.
> - Automatic Storage Compression preserves the sort order
>
> If clustering was enabled using `DBMS_AUTO_CLUSTERING`:
>
> - Sorting does not occur during the direct load.
> - Automatic Storage Compression performs a sort during data movement.

> **See Also:**
>
> - DML and Hybrid Columnar Compression is explained in the *Oracle Database Concepts Guide*.
> - You can find the ENABLE_AUTO_OPTIMIZE procedure in the *Database PL/SQL Packages and Types Reference*.

5.6.1 Example: Using Automatic Storage Compression

1. Enable Automatic Storage Compression. Set `DBMS_ILM_ADMIN.ENABLE_AUTO_OPTIMIZE` at the PDB level

   ```
   EXEC DBMS_ILM_ADMIN.ENABLE_AUTO_OPTIMIZE;
   ```

2. Create a table without HCC compression.
   This example uses the table `MYTAB`, which was created with no compression.

3. Check to verify that the table is not compressed.

```
SELECT UNIQUE DBMS_COMPRESSION.GET_COMPRESSION_TYPE('SCOTT', 'MYTAB',
<ROWID>) FROM SCOTT.MYTAB;

DBMS_COMPRESSION.GET_COMPRESSION_TYPE('SCOTT', 'MYTAB', <ROWID>)
------------------------------------------------------------------------

1
```

> **Note:**
>
> `DBMS_COMPRESSION.GET_COMPRESSION_TYPE` uses constants that can determine compression type. The value “1” indicates that the table is not compressed. See DBMS_COMPRESSION Constants in the *Oracle PL/SQL Packages and Types Reference* for more information.

4. Set `AUTO_OPTIMIZE_INACTIVITY_THRESHOLD`. `AUTO_OPTIMIZE_INACTIVITY_THRESHOLD` specifies how long to wait with no DML activity before starting automatic compression. Wait time is set by `DBMS_ILM_ADMIN.CUSTOMIZE_ILM`. The default is 1440 minutes (one day). You can change the value:

```
EXEC
DBMS_ILM_ADMIN.CUSTOMIZE_ILM(DBMS_ILM_ADMIN.AUTO_OPTIMIZE_INACTIVITY_THRESH
OLD, <NEW_VALUE_IN_MINUTES>);
```

After the defined wait time, Automatic Storage Compression starts up as a background AutoTask and rows are moved and compressed incrementally. Automatic Compression can be paused and restarted. For example, if you want to pause it to determine the size of the table before compression, use `DBMS_ILM_ADMIN.PAUSE_AUTO_OPTIMIZE`.

```
EXEC DBMS_ILM_ADMIN.PAUSE_AUTO_OPTIMIZE;
```

5. Run `ALTER TABLE` to add HCC compression and load data using direct path. Here we add HCC Query LOW compression to the table and insert /*+ append */ to perform the direct-path load. For this example, we will determine the uncompressed size of the table while auto compression is paused.

```
 SELECT BYTES/1024/1024 MB FROM DBA_SEGMENTS WHERE OWNER = 'SCOTT' AND
 SEGMENT_NAME = 'MYTAB';

MB
-----
5.625
```

The query returns the uncompressed table size.

6. Monitor the incremental progress of the automatic compression by checking the `Auto compression data moved` system statistic, which increases over time as data is moved and

compressed. Since we previously paused compression, the value of `Auto compression data moved` is 0. Auto Compression has not started.

```
SELECT NAME, VALUE FROM V$SYSSTAT WHERE NAME LIKE 'AUTO COMPRESSION DATA%';

NAME                                     VALUE
---------------------------------------- --------
Auto compression data movement success      0
Auto compression data movement failure      0
Auto compression data moved                 0
```

Note that `V$SYSSTAT` show the sum of values, across all tables that are using auto compression. If you are compressing more than one table, then the value of `Auto compression data moved` would include the data moved by all of those tables. Also, due to rounding up, the value may not exactly match the actual size of uncompressed data over time.

7. Now let's resume automatic compression so that it begins when `auto_optimize_inactivity_threshold` is met.

```
EXEC DBMS_ILM_ADMIN.RESUME_AUTO_OPTIMIZE;
```

8. You can monitor the compression operation. As it continues, the value of "`Auto compression data moved`" increases.

```
SELECT NAME, VALUE FROM V$SYSSTAT WHERE NAME LIKE 'AUTO COMPRESSION DATA%';

NAME                                     VALUE
---------------------------------------- --------
Auto compression data movement success      1
Auto compression data movement failure      0
Auto compression data moved                 6
```

The value of "Auto compression data moved" indicates that approximately 6 MB of uncompressed data was moved to compression. Note that when auto compression started, the uncompressed size of the data was 5.625 MB.

As this example demonstrates, the segment size before compression, and the amount of data moved during compression, may not be an exact match. `V$SYSSTAT` shows the sum of values across all tables using automatic compression. As noted, if you are loading more than one table, then `Auto compression data moved` includes the data for those tables as well.

6

Using Partitioning in a Data Warehouse Environment

Partitioning features can improve performance in a data warehouse environment.

This chapter describes the partitioning features that significantly enhance data access and improve overall application performance. Improvements with partitioning are especially true for applications that access tables and indexes with millions of rows and many gigabytes of data, as found in a data warehouse environment. Data warehouses often contain large tables and require techniques for managing these large tables and for providing good query performance across these large tables.

This chapter contains the following sections:

- What Is a Data Warehouse?
- Scalability in a Data Warehouse
- Partitioning for Performance in a Data Warehouse
- Manageability in a Data Warehouse

6.1 What Is a Data Warehouse?

A data warehouse is a relational database that is designed for query and analysis rather than for transaction processing.

A data warehouse usually contains historical data derived from transaction data, but can include data from other sources. Data warehouses separate analysis workload from transaction workload and enable an organization to consolidate data from several sources.

In addition to a relational database, a data warehouse environment can include an extraction, transformation, and loading (ETL) solution, analytical processing and data mining capabilities, client analysis tools, and other applications that manage the process of gathering data and delivering it to business users.

> **See Also:**
>
> *Oracle Database Data Warehousing Guide*

6.2 Scalability in a Data Warehouse

Partitioning helps to scale a data warehouse by dividing database objects into smaller pieces, enabling access to smaller, more manageable objects. Having direct access to smaller objects addresses the scalability requirements of data warehouses.

This section contains the following topics:

- Bigger Databases

- Bigger Individual Tables: More Rows in Tables
- More Users Querying the System
- More Complex Queries

6.2.1 Bigger Databases

The ability to split a large database object into smaller pieces transparently simplifies efficient management of very large databases.

You can identify and manipulate individual partitions and subpartitions to manage large database objects. Consider the following advantages of partitioned objects:

- Backup and recovery can be performed on a low level of granularity to manage the size of the database.
- Part of a database object can be placed in compressed storage while other parts can remain uncompressed.
- Partitioning can store data transparently on different storage tiers to lower the cost of retaining vast amounts of data. For more information, refer to Managing and Maintaining Time-Based Information.

6.2.2 Bigger Individual Tables: More Rows in Tables

It takes longer to scan a big table than it takes to scan a small table. Queries against partitioned tables may access one or more partitions that are small in contrast to the total size of the table.

Similarly, queries may take advantage of partition elimination on indexes. It takes less time to read a smaller portion of an index from disk than to read the entire index. Index structures that share the partitioning strategy with the table, such as local partitioned indexes, can be accessed and maintained on a partition-by-partition basis.

The database can take advantage of the distinct data sets in separate partitions if you use parallel execution to speed up queries, DML, and DDL statements. Individual parallel execution servers can work on their own data sets, identified by the partition boundaries.

6.2.3 More Users Querying the System

With partitioning, users are more likely to query on isolated and smaller data sets.

Consequently, the database can return results faster than if all users queried the same and much larger data sets. Data contention is less likely.

6.2.4 More Complex Queries

You can perform complex queries faster using smaller data sets.

If smaller data sets are being accessed, then complex calculations are more likely to be processed in memory, which is beneficial from a performance perspective and reduces the application's I/O requirements. A larger data set may have to be written to the temporary tablespace to complete a query, in which case additional I/O operations against the database storage occurs.

6.3 Partitioning for Performance in a Data Warehouse

Good performance is a requirement for a successful data warehouse.

Analyses run against the database should return within a reasonable amount of time, even if the queries access large amounts of data in tables that are terabytes in size. Partitioning increases the speed of data access and application processing, which results in successful data warehouses that are not prohibitively expensive.

This section contains the following topics:

- Partition Pruning in a Data Warehouse
- Partition-Wise Joins in a Data Warehouse
- Indexes and Partitioned Indexes in a Data Warehouse
- Materialized Views and Partitioning in a Data Warehouse

6.3.1 Partition Pruning in a Data Warehouse

Partition pruning is an essential performance feature for data warehouses.

In partition pruning, the optimizer analyzes `FROM` and `WHERE` clauses in SQL statements to eliminate unneeded partitions when building the partition access list. As a result, Oracle Database performs operations only on those partitions that are relevant to the SQL statement.

Partition pruning dramatically reduces the amount of data retrieved from disk and shortens processing time, thus improving query performance and optimizing resource utilization.

This section contains the following topics:

- Basic Partition Pruning Techniques
- Advanced Partition Pruning Techniques

For more information about partition pruning and the difference between static and dynamic partition pruning, refer to Partitioning for Availability, Manageability, and Performance.

6.3.1.1 Basic Partition Pruning Techniques

The optimizer uses a wide variety of predicates for pruning.

The three predicate types, equality, range, and `IN`-list, are the predicates most commonly used for partition pruning. As an example, consider the following query:

```
SELECT SUM(amount_sold) day_sales
FROM sales
WHERE time_id = TO_DATE('02-JAN-1998', 'DD-MON-YYYY');
```

Because there is an equality predicate on the partitioning column of `sales`, the query is pruned down to a single predicate and this is reflected in the following execution plan:

```
-------------------------------------------------------------------------------------------
|  Id | Operation               | Name  | Rows| Bytes | Cost (%CPU)| Time     |Pstart| Pstop |
-------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT        |       |     |       | 21 (100)   |          |      |       |
|   1 |  SORT AGGREGATE         |       | 1   | 13    |            |          |      |       |
|   2 |   PARTITION RANGE SINGLE|       | 485 | 6305  | 21 (10)    | 00:00:01 | 5    | 5     |
| * 3 |    TABLE ACCESS FULL    | SALES | 485 | 6305  | 21 (10)    | 00:00:01 | 5    | 5     |
```

```
-----------------------------------------------------------------------------------------------
Predicate Information (identified by operation id):
---------------------------------------------------
  3 - filter("TIME_ID"=TO_DATE('1998-01-02 00:00:00', 'yyyy-mm-dd hh24:mi:ss'))
```

Similarly, a range or an `IN`-list predicate on the `time_id` column and the optimizer would be used to prune to a set of partitions. The partitioning type plays a role in which predicates can be used. Range predicates cannot be used for pruning on hash partitioned tables, but they can be used for all other partitioning strategies. However, on list-partitioned tables, range predicates may not map to a contiguous set of partitions. Equality and `IN`-list predicates can prune with all the partitioning methods.

6.3.1.2 Advanced Partition Pruning Techniques

Oracle Database pruning feature effectively handles more complex predicates or SQL statements that involve partitioned tables.

A common situation is when a partitioned table is joined to the subset of another table, limited by a `WHERE` condition. For example, consider the following query:

```
SELECT t.day_number_in_month, SUM(s.amount_sold)
  FROM sales s, times t
  WHERE s.time_id = t.time_id
    AND t.calendar_month_desc='2000-12'
  GROUP BY t.day_number_in_month;
```

If the database performed a nested loop join with `times` table on the right-hand side, then the query would access only the partition corresponding to this row from the `times` table, so pruning would implicitly take place. But, if the database performed a hash or sort merge join, this would not be possible. If the table with the `WHERE` predicate is relatively small compared to the partitioned table, and the expected reduction of records or partitions for the partitioned table is significant, then the database performs dynamic partition pruning using a recursive subquery. The decision whether to invoke subquery pruning is an internal cost-based decision of the optimizer.

A sample execution plan using a hash join operation would look like the following:

```
-----------------------------------------------------------------------------------------------
| Id| Operation                    |  Name |  Rows | Bytes| Cost (%CPU)|  Time  | Pstart | Pstop |
-----------------------------------------------------------------------------------------------
|  0| SELECT STATEMENT             |       |       |      | 761 (100)  |        |        |       |
|  1|  HASH GROUP BY               |       |    20 | 640  | 761 (41)   |00:00:10|        |       |
|* 2|   HASH JOIN                  |       | 19153 | 598K | 749 (40)   |00:00:09|        |       |
|* 3|    TABLE ACCESS FULL         | TIMES |    30 |  570 |  17 (6)    |00:00:01|        |       |
|  4|     PARTITION RANGE SUBQUERY |       |  918K | 11M  |   655 (33) |00:00:08| KEY(SQ)|KEY(SQ)|
|  5|      TABLE ACCESS FULL       | SALES |   918 | 11M  |   655 (33) |00:00:08| KEY(SQ)|KEY(SQ)|
-----------------------------------------------------------------------------------------------
Predicate Information (identified by operation id):
---------------------------------------------------
PLAN_TABLE_OUTPUT
-----------------------------------------------------------------------------------------------
  2 - access("S"."TIME_ID"="T"."TIME_ID")
  3 - filter("T"."CALENDAR_MONTH_DESC"='2000-12')
```

This execution plan shows that dynamic partition pruning occurred on the `sales` table using a subquery, as shown by the `KEY(SQ)` value in the `PSTART` and `PSTOP` columns.

The following is an example of advanced pruning using an `OR` predicate.

```
SELECT p.promo_name promo_name, (s.profit - p.promo_cost) profit
FROM
   promotions p,
```

```
      ( SELECT
         sales.promo_id,
         SUM(sales.QUANTITY_SOLD * (costs.UNIT_PRICE - costs.UNIT_COST)) profit
      FROM
         sales, costs
      WHERE
         ((sales.time_id BETWEEN TO_DATE('01-JAN-1998','DD-MON-YYYY',
                     'NLS_DATE_LANGUAGE = American') AND
         TO_DATE('01-JAN-1999','DD-MON-YYYY', 'NLS_DATE_LANGUAGE = American')
      OR
         (sales.time_id BETWEEN TO_DATE('01-JAN-2001','DD-MON-YYYY',
                     'NLS_DATE_LANGUAGE = American') AND
         TO_DATE('01-JAN-2002','DD-MON-YYYY', 'NLS_DATE_LANGUAGE = American')))
         AND sales.time_id = costs.time_id
         AND sales.prod_id = costs.prod_id)
      GROUP BY
         sales.promo_id) s
   WHERE s.promo_id = p.promo_id
   ORDER BY profit
   DESC;
```

This query joins the `sales` and `costs` tables. The `sales` table is partitioned by range on the column `time_id`. One condition in the query is two predicates on `time_id`, which are combined with an `OR` operator. This `OR` predicate is used to prune the partitions in the `sales` table and a single join between the `sales` and `costs` table is performed. The execution plan is as follows:

```
-------------------------------------------------------------------------------------------------
| Id| Operation               |  Name    |Rows |Bytes |TmpSp|Cost(%CPU)| Time     | Pstart| Pstop |
-------------------------------------------------------------------------------------------------
|  0| SELECT STATEMENT        |          | 4   |  200 |     | 3556 (14)| 00:00:43|       |       |
|  1|  SORT ORDER BY          |          | 4   |  200 |     | 3556 (14)| 00:00:43|       |       |
|* 2|   HASH JOIN             |          | 4   |  200 |     | 3555 (14)| 00:00:43|       |       |
|  3|    TABLE ACCESS FULL    |PROMOTIONS| 503 | 16599|     |    16 (0)| 00:00:01|       |       |
|  4|    VIEW                 |          |   4 |   68 |     | 3538 (14)| 00:00:43|       |       |
|  5|     HASH GROUP BY       |          |   4 |  164 |     | 3538 (14)| 00:00:43|       |       |
|  6|      PARTITION RANGE OR |          | 314K|   12M|     |  3321 (9)| 00:00:40|KEY(OR)|KEY(OR)|
|* 7|       HASH JOIN         |          | 314K|   12M| 440K|  3321 (9)| 00:00:40|       |       |
|* 8|        TABLE ACCESS FULL| SALES    | 402K| 7467K|     |  400 (39)| 00:00:05|KEY(OR)|KEY(OR)|
|  9| TABLE ACCESS FULL       | COSTS    |82112| 1764K|     |   77 (24)| 00:00:01|KEY(OR)|KEY(OR)|
-------------------------------------------------------------------------------------------------
Predicate Information (identified by operation id):
---------------------------------------------------
  2 - access("S"."PROMO_ID"="P"."PROMO_ID")
  7 - access("SALES"."TIME_ID"="COSTS"."TIME_ID" AND "SALES"."PROD_ID"="COSTS"."PROD_ID")
  8 - filter("SALES"."TIME_ID"<=TO_DATE('1999-01-01 00:00:00', 'yyyy-mm-dd hh24:mi:ss') AND
      "SALES"."TIME_ID">=TO_DATE('1998-01-01 00:00:00', 'yyyy-mm-dd hh24:mi:ss') OR
      "SALES"."TIME_ID">=TO_DATE('2001-01-01 00:00:00', 'yyyy-mm-dd hh24:mi:ss') AND
      "SALES"."TIME_ID"<=TO_DATE('2002-01-01 00:00:00', 'yyyy-mm-dd hh24:mi:ss'))
```

The database also does additional pruning when a column is range-partitioned on multiple columns. As long as the database can guarantee that a particular predicate cannot be satisfied in a particular partition, the partition is skipped. This allows the database to optimize cases where there are range predicates on multiple columns or in the case where there are no predicates on a prefix of the partitioning columns.

For tips on partition pruning, refer to Partition Pruning Tips.

6.3.2 Partition-Wise Joins in a Data Warehouse

Partition-wise joins reduce query response time by minimizing the amount of data exchanged among parallel execution servers when joins execute in parallel.

Using partition-wise joins significantly reduces response time and improves the use of both CPU and memory resources. Parallel partition-wise joins are used commonly for processing large joins efficiently and fast. Partition-wise joins can be full or partial. Oracle Database decides which type of join to use.

In addition to parallel partition-wise joins, queries using the `SELECT DISTINCT` clause and SQL window functions can perform parallel partition-wise operations.

This section contains the following topics:

- Full Partition-Wise Joins
- Partial Partition-Wise Joins
- Benefits of Partition-Wise Joins
- Performance Considerations for Parallel Partition-Wise Joins

> **See Also:**
>
> - Partition-Wise Operations for additional information about partition-wise operations
> - *Oracle Database Data Warehousing Guide* for information about data warehousing and optimization techniques

6.3.2.1 Full Partition-Wise Joins

Full partition-wise joins can occur if two tables that are co-partitioned on the same key are joined in a query.

The tables can be co-partitioned at the partition level, or at the subpartition level, or at a combination of partition and subpartition levels. Reference partitioning is an easy way to guarantee co-partitioning. Full partition-wise joins can be executed serially and in parallel.

For more information about partition-wise joins, refer to Partitioning for Availability, Manageability, and Performance.

The following example shows a full partition-wise join on the `orders` and `order_items` tables, in which the `order_items` table is reference-partitioned.

```
CREATE TABLE orders
( order_id     NUMBER(12) NOT NULL
, order_date   DATE NOT NULL
, order_mode   VARCHAR2(8)
, order_status VARCHAR2(1)
, CONSTRAINT orders_pk PRIMARY KEY (order_id)
)
PARTITION BY RANGE (order_date)
( PARTITION p_before_jan_2006 VALUES LESS THAN (TO_DATE('01-JAN-2006','dd-MON-yyyy'))
, PARTITION p_2006_jan VALUES LESS THAN (TO_DATE('01-FEB-2006','dd-MON-yyyy'))
, PARTITION p_2006_feb VALUES LESS THAN (TO_DATE('01-MAR-2006','dd-MON-yyyy'))
, PARTITION p_2006_mar VALUES LESS THAN (TO_DATE('01-APR-2006','dd-MON-yyyy'))
, PARTITION p_2006_apr VALUES LESS THAN (TO_DATE('01-MAY-2006','dd-MON-yyyy'))
, PARTITION p_2006_may VALUES LESS THAN (TO_DATE('01-JUN-2006','dd-MON-yyyy'))
, PARTITION p_2006_jun VALUES LESS THAN (TO_DATE('01-JUL-2006','dd-MON-yyyy'))
, PARTITION p_2006_jul VALUES LESS THAN (TO_DATE('01-AUG-2006','dd-MON-yyyy'))
, PARTITION p_2006_aug VALUES LESS THAN (TO_DATE('01-SEP-2006','dd-MON-yyyy'))
, PARTITION p_2006_sep VALUES LESS THAN (TO_DATE('01-OCT-2006','dd-MON-yyyy'))
```

```
, PARTITION p_2006_oct VALUES LESS THAN (TO_DATE('01-NOV-2006','dd-MON-yyyy'))
, PARTITION p_2006_nov VALUES LESS THAN (TO_DATE('01-DEC-2006','dd-MON-yyyy'))
, PARTITION p_2006_dec VALUES LESS THAN (TO_DATE('01-JAN-2007','dd-MON-yyyy'))
)
PARALLEL;

CREATE TABLE order_items
( order_id NUMBER(12) NOT NULL
, product_id NUMBER NOT NULL
, quantity NUMBER NOT NULL
, sales_amount NUMBER NOT NULL
, CONSTRAINT order_items_orders_fk FOREIGN KEY (order_id) REFERENCES
orders(order_id)
)
PARTITION BY REFERENCE (order_items_orders_fk)
PARALLEL;
```

A typical data warehouse query would scan a large amount of data. In the underlying execution plan, the columns `Rows`, `Bytes`, `Cost (%CPU)`, `Time`, and `TQ` have been removed.

```
EXPLAIN PLAN FOR
SELECT o.order_date
, sum(oi.sales_amount) sum_sales
FROM orders o
, order_items oi
WHERE o.order_id = oi.order_id
AND o.order_date BETWEEN TO_DATE('01-FEB-2006','DD-MON-YYYY')
                     AND TO_DATE('31-MAY-2006','DD-MON-YYYY')
GROUP BY o.order_id
, o.order_date
ORDER BY o.order_date;

-----------------------------------------------------------------------------------------------
| Id  | Operation                          | Name        | Pstart| Pstop |IN-OUT| PQ Distrib |
-----------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT                   |             |       |       |      |            |
|   1 |  PX COORDINATOR                    |             |       |       |      |            |
|   2 |   PX SEND QC (ORDER)               | :TQ10001    |       |       | P->S | QC (ORDER) |
|   3 |    SORT GROUP BY                   |             |       |       | PCWP |            |
|   4 |     PX RECEIVE                     |             |       |       | PCWP |            |
|   5 |      PX SEND RANGE                 | :TQ10000    |       |       | P->P | RANGE      |
|   6 |       SORT GROUP BY                |             |       |       | PCWP |            |
|   7 |        PX PARTITION RANGE ITERATOR|             |     3 |     6 | PCWC |            |
|*  8 |         HASH JOIN                  |             |       |       | PCWP |            |
|*  9 |          TABLE ACCESS FULL         | ORDERS      |     3 |     6 | PCWP |            |
|  10 |          TABLE ACCESS FULL         | ORDER_ITEMS |     3 |     6 | PCWP |            |
-----------------------------------------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------

   8 - access("O"."ORDER_ID"="OI"."ORDER_ID")
   9 - filter("O"."ORDER_DATE"<=TO_DATE(' 2006-05-31 00:00:00', 'syyyy-mm-dd hh24:mi:ss'))
```

6.3.2.2 Partial Partition-Wise Joins

Oracle Database can perform partial partition-wise joins only in parallel.

Unlike full partition-wise joins, partial partition-wise joins require you to partition only one table on the join key, not both tables. The partitioned table is referred to as the reference table. The

other table may or may not be partitioned. Partial partition-wise joins are more common than full partition-wise joins.

To execute a partial partition-wise join, the database dynamically partitions or repartitions the other table based on the partitioning of the reference table. After the other table is repartitioned, the execution is similar to a full partition-wise join.

The following example shows a call detail records table, `cdrs`, in a typical data warehouse scenario. The table is interval-hash partitioned.

```
CREATE TABLE cdrs
( id                 NUMBER
, cust_id            NUMBER
, from_number        VARCHAR2(20)
, to_number          VARCHAR2(20)
, date_of_call       DATE
, distance           VARCHAR2(1)
, call_duration_in_s NUMBER(4)
) PARTITION BY RANGE(date_of_call)
INTERVAL (NUMTODSINTERVAL(1,'DAY'))
SUBPARTITION BY HASH(cust_id)
SUBPARTITIONS 16
(PARTITION p0 VALUES LESS THAN (TO_DATE('01-JAN-2005','dd-MON-yyyy')))
PARALLEL;
```

The `cdrs` table is joined with the nonpartitioned `callers` table on the `cust_id` column to rank the customers who spent the most time making calls.

```
EXPLAIN PLAN FOR
SELECT c.cust_id
,      c.cust_last_name
,      c.cust_first_name
,      AVG(call_duration_in_s)
,      COUNT(1)
,      DENSE_RANK() OVER
       (ORDER BY (AVG(call_duration_in_s) * COUNT(1)) DESC) ranking
FROM   callers c
,      cdrs    cdr
WHERE cdr.cust_id = c.cust_id
AND cdr.date_of_call BETWEEN TO_DATE('01-JAN-2006','dd-MON-yyyy')
                         AND TO_DATE('31-DEC-2006','dd-MON-yyyy')
GROUP BY c.cust_id
, c.cust_last_name
, c.cust_first_name
ORDER BY ranking;
```

The execution plans shows a partial partition-wise join. In the plan, the columns `Rows`, `Bytes`, `Cost (%CPU)`, `Time`, and `TQ` have been removed.

```
-------------------------------------------------------------------------------------------------
| Id  | Operation                          | Name     | Pstart| Pstop |IN-OUT| PQ Distrib |
-------------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT                   |          |       |       |      |            |
|   1 |  WINDOW NOSORT                     |          |       |       |      |            |
|   2 |   PX COORDINATOR                   |          |       |       |      |            |
|   3 |    PX SEND QC (ORDER)              | :TQ10002 |       |       | P->S | QC (ORDER) |
|   4 |     SORT ORDER BY                  |          |       |       | PCWP |            |
|   5 |      PX RECEIVE                    |          |       |       | PCWP |            |
|   6 |       PX SEND RANGE                | :TQ10001 |       |       | P->P | RANGE      |
|   7 |        HASH GROUP BY               |          |       |       | PCWP |            |
|*  8 |         HASH JOIN                  |          |       |       | PCWP |            |
|   9 |          PART JOIN FILTER CREATE   | :BF0000  |       |       | PCWP |            |
```

```
|  10 |           BUFFER SORT                |          |       |       | PCWC |            |
|  11 |            PX RECEIVE                |          |       |       | PCWP |            |
|  12 |             PX SEND PARTITION (KEY)  | :TQ10000 |       |       | S->P | PART (KEY) |
|  13 |              TABLE ACCESS FULL       | CALLERS  |       |       |      |            |
|  14 |           PX PARTITION RANGE ITERATOR|          |   367 |   731 | PCWC |            |
|  15 |            PX PARTITION HASH ALL     |          |     1 |    16 | PCWC |            |
|* 16 |            TABLE ACCESS FULL         | CDRS     |  5857 | 11696 | PCWP |            |
-------------------------------------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------

   8 - access("CDR"."CUST_ID"="C"."CUST_ID")
  16 - filter("CDR"."DATE_OF_CALL">=TO_DATE(' 2006-01-01 00:00:00', 'syyyy-mm-dd
hh24:mi:ss') AND "CDR"."DATE_OF_CALL"<=TO_DATE('
              2006-12-31 00:00:00', 'syyyy-mm-dd hh24:mi:ss'))
```

6.3.2.3 Benefits of Partition-Wise Joins

Partition-wise joins offers several benefits.

These benefits are described in the following topics:

- Reduction of Communications Overhead
- Reduction of Memory Requirements

6.3.2.3.1 Reduction of Communications Overhead

When executed in parallel, partition-wise joins reduce communications overhead.

This reduction, in the default case, occurs because parallel execution of a join operation by a set of parallel execution servers requires the redistribution of each table on the join column into disjoint subsets of rows. These disjoint subsets of rows are then joined pair-wise by a single parallel execution server.

The database can avoid redistributing the partitions because the two tables are partitioned on the join column. This functionality enables each parallel execution server to join a pair of matching partitions. This improved performance from using parallel execution is even more noticeable in Oracle Real Application Clusters configurations with internode parallel execution.

Partition-wise joins dramatically reduce interconnect traffic. Using this feature is key for large decision support systems (DSS) configurations that use Oracle Real Application Clusters. Currently, most Oracle Real Application Clusters platforms, such as massively parallel processing (MPP) and symmetric multiprocessing (SMP) clusters, provide limited interconnect bandwidths compared to their processing powers. Ideally, interconnect bandwidth should be comparable to disk bandwidth, but this is seldom the case. Consequently, most join operations in Oracle Real Application Clusters experience high interconnect latencies without parallel execution of partition-wise joins.

6.3.2.3.2 Reduction of Memory Requirements

Partition-wise joins require less memory than the equivalent join operation of the complete data set of the tables being joined.

For serial joins, the join is performed at the same time on a pair of matching partitions. If data is evenly distributed across partitions, then the memory requirement is divided by the number of partitions and there is no skew to the data distribution among the parallel servers.

For parallel joins, memory requirements depend on the number of partition pairs that are joined in parallel. For example, if the degree of parallelism is 20 and the number of partitions is 100, then 5 times less memory is required because only 20 joins of two partitions each are performed at the same time. The fact that partition-wise joins require less memory has a direct beneficial effect on performance. For example, the join probably does not need to write blocks to disk during the build phase of a hash join.

6.3.2.4 Performance Considerations for Parallel Partition-Wise Joins

The optimizer weighs the advantages and disadvantages when deciding whether to use partition-wise joins.

The optimizer chooses whether to use partition-wise joins based on the following:

- In range partitioning where partition sizes differ, data skew increases response time; some parallel execution servers take longer than others to finish their joins. Oracle recommends the use of hash partitioning and subpartitioning to enable partition-wise joins because hash partitioning, if the number of partitions is a power of two, limits the risk of skew. Ideally, the hash partitioning key is unique to minimize the risk of skew.
- The number of partitions used for partition-wise joins should, if possible, be a multiple of the number of query servers. With a degree of parallelism of 16, for example, you can have 16, 32, or even 64 partitions. If there is an odd number of partitions, then some parallel execution servers are used less than others. For example, if there are 17 evenly distributed partition pairs, only one pair works on the last join, while the other pair has to wait. This is because, in the beginning of the execution, each parallel execution server works on a different partition pair. After this first phase, only one pair remains. Thus, a single parallel execution server joins this remaining pair while all other parallel execution servers are idle.

In some situations, parallel joins can cause remote I/O operations. For example, on Oracle Real Application Clusters environments running on MPP configurations, if a pair of matching partitions is not collocated on the same node, a partition-wise join requires extra internode communication due to remote I/O. This is because Oracle Database must transfer at least one partition to the node where the join is performed. In this case, it is better to explicitly redistribute the data than to use a partition-wise join.

6.3.3 Indexes and Partitioned Indexes in a Data Warehouse

Indexes are optional structures associated with tables that allow SQL statements to execute more quickly against a table.

Even though table scans are very common in many data warehouses, indexes can often speed up queries.

Both B-tree and bitmap indexes can be created as local indexes on a partitioned table, in which case they inherit the table's partitioning strategy. B-tree indexes can be created as global partitioned indexes on partitioned and nonpartitioned tables.

This section contains the following topics:

- Local Partitioned Indexes
- Nonpartitioned Indexes
- Global Partitioned Indexes

For more information about partitioned indexes, refer to Partitioning for Availability, Manageability, and Performance.

6.3.3.1 Local Partitioned Indexes

In a local index, all keys in a particular index partition refer only to rows stored in a single underlying table partition.

A local index is equipartitioned with the underlying table. Oracle Database partitions the index on the same columns as the underlying table, creates the same number of partitions or subpartitions, and gives them the same partition boundaries as corresponding partitions of the underlying table.

Oracle Database also maintains the index partitioning automatically when partitions in the underlying table are added, dropped, merged, or split, or when hash partitions or subpartitions are added or coalesced. This ensures that the index remains equipartitioned with the table.

For data warehouse applications, local nonprefixed indexes can improve performance because many index partitions can be scanned in parallel by range queries on the index key. The following example creates a local B-tree index on a partitioned `customers_dw` table:

```
CREATE INDEX cust_last_name_ix
 ON customers_dw(last_name) LOCAL
 PARALLEL NOLOGGING ;
```

Bitmap indexes use a very efficient storage mechanism for low cardinality columns. Bitmap indexes are used in data warehouses, and especially common in data warehouses that implement *star schemas*. A single star schema consists of a central large fact table and multiple smaller dimension tables that describe the data in the fact table.

For example, consider a `sales` table that is a fact table, described by dimension tables `customers`, `products`, `promotions`, `times`, and `channels`. Bitmap indexes enable the star transformation, an optimization for fast query retrieval against star or star look-a-like schemas.

Fact table foreign key columns are ideal candidates for bitmap indexes, because generally there are few distinct values relative to the total number of rows. Fact tables are often range or range-* partitioned, in which case you must create local bitmap indexes. Global bitmap indexes on partitioned tables are not supported.

The following example creates a local partitioned bitmap index on the `sales` table:

```
CREATE BITMAP INDEX prod_id_ix
ON sales(prod_id) LOCAL
PARALLEL NOLOGGING;
```

> **See Also:**
>
> *Oracle Database Data Warehousing Guide* for more information about the star transformation

6.3.3.2 Nonpartitioned Indexes

You can create nonpartitioned indexes on nonpartitioned and partitioned tables.

Nonpartitioned indexes are primarily used on nonpartitioned tables in data warehouse environments and in general to enforce uniqueness if the status of a unique constraint is required to be enforced in a data warehousing environment. You can use a nonpartitioned global index on a partitioned table to enforce a primary or unique key. A nonpartitioned (global)

index can be useful for queries that commonly retrieve very few rows based on equality predicates or `IN`-list on a column or set of columns that is not included in the partitioning key. In those cases, it can be faster to scan a single index than to scan many index partitions to find all matching rows.

Unique indexes on columns other than the partitioning columns must be global because unique local nonprefixed indexes whose keys do not contain the partitioning keys are not supported. Unique keys are not always enforced in data warehouses due to the controlled data load processes and the performance cost of enforcing the unique constraint. Global indexes can grow very large on tables with billions of rows.

The following example creates a global unique index on the `sales` table:

```
CREATE UNIQUE INDEX sales_unique_ix
  ON sales(cust_id, prod_id, promo_id, channel_id, time_id)
  PARALLEL NOLOGGING;
```

Very few queries benefit from this index. In systems with a very limited data load window, consider not creating and maintaining it.

6.3.3.3 Global Partitioned Indexes

You can create global partitioned indexes on nonpartitioned and partitioned tables.

In a global partitioned index, the keys in a particular index partition may refer to rows stored in multiple underlying table partitions or subpartitions. A global index can be range or hash partitioned, though it can be defined on any type of partitioned table.

A global index is created by specifying the `GLOBAL` attribute. The database administrator is responsible for defining the initial partitioning of a global index at creation and for maintaining the partitioning over time. Index partitions can be merged or split as necessary.

Global indexes can be useful if there is a class of queries that uses an access path to the table to retrieve a few rows through an index, and by partitioning the index you can eliminate large portions of the index for the majority of its queries. On a partitioned table, you would consider a global partitioned index if the column or columns included to achieve partition pruning do not include the table partitioning key.

The following example creates a global hash partitioned index on the `sales` table:

```
CREATE INDEX cust_id_prod_id_global_ix
ON sales(cust_id,prod_id)
GLOBAL PARTITION BY HASH (cust_id)
( PARTITION p1 TABLESPACE tbs1
, PARTITION p2 TABLESPACE tbs2
, PARTITION p3 TABLESPACE tbs3
, PARTITION p4 TABLESPACE tbs4
)
PARALLEL NOLOGGING;
```

6.3.4 Materialized Views and Partitioning in a Data Warehouse

One technique employed in data warehouses to improve performance is the creation of summaries. Summaries are special types of aggregate views that improve query execution times by precalculating expensive joins and aggregation operations before execution and storing the results in a table in the database.

For example, you can create a summary table to contain the sums of sales by region and by product.

The summaries or aggregates that are referred to in this guide and in literature on data warehousing are created in Oracle Database using a schema object called a materialized view. Materialized views in a data warehouse speed up query performance.

The database supports transparent rewrites against materialized views, so that you do not need to modify the original queries to take advantage of precalculated results in materialized views. Instead of executing the query, the database retrieves precalculated results from one or more materialized views, performs any necessary additional operations on the data, and returns the query results.

> **See Also:**
>
> *Oracle Database Data Warehousing Guide* for information about data warehousing and materialized views

6.3.4.1 Partitioned Materialized Views

The underlying storage for a materialized view is a table structure. You can partition materialized views like you can partition tables.

When the database rewrites a query to run against materialized views, the query can take advantage of the same performance features from which queries running against tables directly benefit. The rewritten query may eliminate materialized view partitions. If joins back to tables or with other materialized views are necessary to retrieve the query result, then the rewritten query can take advantage of partition-wise joins.

Example 6-1 shows how to create a compressed partitioned materialized view that aggregates sales results to country level. This materialized view benefits from queries that summarize sales numbers by country level or higher to subregion or region level.

Example 6-1 Creating a compressed partitioned materialized view

```
CREATE MATERIALIZED VIEW country_sales
PARTITION BY HASH (country_id)
PARTITIONS 16
COMPRESS FOR OLTP
PARALLEL NOLOGGING
ENABLE QUERY REWRITE
AS SELECT co.country_id
, co.country_name
, co.country_subregion
, co.country_region
, sum(sa.quantity_sold) country_quantity_sold
, sum(sa.amount_sold) country_amount_sold
FROM sales sa
, customers cu
, countries co
WHERE sa.cust_id = cu.cust_id
AND cu.country_id = co.country_id
GROUP BY co.country_id
, co.country_name
, co.country_subregion
, co.country_region;
```

> **See Also:**
>
> *Oracle Database Data Warehousing Guide* for information about data warehousing and materialized views

6.4 Manageability in a Data Warehouse

Data warehouses store historical data. Important parts of a data warehouse are the data loading and purging. Partitioning is powerful technology that can help data management for data warehouses.

This section contains the following topics:

- Partition Exchange Load
- Partitioning and Indexes
- Removing Data from Tables
- Partitioning and Data Compression

> **See Also:**
>
> *Oracle Database SQL Tuning Guide* for information about collecting and managing statistics on partitioned indexes, exchanges, and tables

6.4.1 Partition Exchange Load

Partitions can be added using partition exchange load (PEL).

When you use PEL, you create a separate table that looks exactly like a single partition, including the same indexes and constraints, if any. If you use a composite partitioned table, then your separate table must use a partitioning strategy that matches the subpartitioning strategy of your composite partitioned table. You can then exchange an existing table partition with this separate table. In a data load scenario, data can be loaded into the separate table. Build indexes and implement constraints on the separate table, without impacting the table users query. Then perform the PEL, which is a very low-impact transaction compared to the data load. Daily loads, with a range partition strategy by day, are common in data warehouse environments.

The following example shows a partition exchange load for the `sales` table:

```
ALTER TABLE sales ADD PARTITION p_sales_jun_2007
VALUES LESS THAN (TO_DATE('01-FEB-2007','dd-MON-yyyy'));

CREATE TABLE sales_jun_2007 COMPRESS FOR OLTP
AS SELECT * FROM sales WHERE 1=0;
```

Next, populate table `sales_jun_2007` with sales numbers for June 2007, and create the equivalent bitmap indexes and constraints that have been implemented on the `sales` table:

```
CREATE BITMAP INDEX time_id_jun_2007_bix ON sales_jun_2007(time_id) NOLOGGING;
CREATE BITMAP INDEX cust_id_jun_2007_bix ON sales_jun_2007(cust_id) NOLOGGING;
CREATE BITMAP INDEX prod_id_jun_2007_bix ON sales_jun_2007(prod_id) NOLOGGING;
```

```
CREATE BITMAP INDEX promo_id_jun_2007_bix ON sales_jun_2007(promo_id) NOLOGGING;
CREATE BITMAP INDEX channel_id_jun_2007_bix ON sales_jun_2007(channel_id) NOLOGGING;

ALTER TABLE sales_jun_2007 ADD CONSTRAINT prod_id_fk FOREIGN KEY (prod_id) REFERENCES products(prod_id);
ALTER TABLE sales_jun_2007 ADD CONSTRAINT cust_id_fk FOREIGN KEY (cust_id) REFERENCES customers(cust_id);
ALTER TABLE sales_jun_2007 ADD CONSTRAINT promo_id_fk FOREIGN KEY (promo_id) REFERENCES
promotions(promo_id);
ALTER TABLE sales_jun_2007 ADD CONSTRAINT time_id_fk FOREIGN KEY (time_id) REFERENCES times(time_id);
ALTER TABLE sales_jun_2007 ADD CONSTRAINT channel_id_fk FOREIGN KEY (channel_id) REFERENCES
channels(channel_id);
```

Next, exchange the partition:

```
ALTER TABLE sales
EXCHANGE PARTITION p_sales_jun_2007
WITH TABLE sales_jun_2007
INCLUDING INDEXES;
```

For more information about partition exchange load, refer to Partition Administration.

6.4.2 Partitioning and Indexes

Partition maintenance operations are most easily performed on local indexes.

Local indexes do not invalidate a global index when partition management takes place. Use `INCLUDING INDEXES` in the PEL statement to exchange the local indexes with the equivalent indexes on the separate table so that no index partitions get invalidated. For PEL, you can update global indexes as part of the load. Use the `UPDATE GLOBAL INDEXES` extension to the PEL statement. If an index requires updating, then the PEL takes much longer.

6.4.3 Removing Data from Tables

Data warehouses commonly keep a time window of data. For example, three years of historical data is stored.

Partitioning makes it very easy to purge data from a table. You can use the `DROP PARTITION` or `TRUNCATE PARTITION` statements to purge data. Common strategies also include using a partition exchange load to unload the data from the table and replacing the partition with an empty table before dropping the partition. Archive the separate table you exchanged before emptying or dropping it.

A drop or truncate operation would invalidate a global index or a global partitioned index. Local indexes remain valid. The local index partition is dropped when you drop the table partition.

The following example shows how to drop partition `sales_1995` from the `sales` table:

```
ALTER TABLE sales
 DROP PARTITION sales_1995
 UPDATE GLOBAL INDEXES PARALLEL;
```

6.4.4 Partitioning and Data Compression

Data in a partitioned table can be compressed on a partition-by-partition basis.

Using compressed data is most efficient for data that does not change frequently. Common data warehouse scenarios often see few data changes as data ages and other scenarios only insert data. Using the partition management features, you can compress data on a partition-by-partition basis. Although Oracle Database supports compression for all DML operations, it is still more efficient to modify data in a noncompressed table.

Altering a partition to enable compression applies only to future data to be inserted into the partition. To compress the existing data in the partition, you must move the partition. Enabling compression and moving a partition can be done in a single operation.

To use table compression on partitioned tables with bitmap indexes, you must do the following before you introduce the compression attribute for the first time:

1. Mark bitmap indexes `UNUSABLE`.
2. Set the compression attribute.
3. Rebuild the indexes.

The first time you make a compressed partition part of an existing, fully uncompressed partitioned table, you must either drop all existing bitmap indexes or mark them `UNUSABLE` before adding a compressed partition. This must be done regardless of whether any partition contains data. It is also independent of the operation that causes one or more compressed partitions to become part of the table. This does not apply to a partitioned table having only B-tree indexes.

The following example shows how to compress the `SALES_1995` partition in the `sales` table:

```
ALTER TABLE sales
 MOVE PARTITION sales_1995
 COMPRESS FOR OLTP
 PARALLEL NOLOGGING;
```

If a table or a partition takes less space on disk, then the performance of large table scans in an I/O-constraint environment may improve.

7

Using Partitioning in an Online Transaction Processing Environment

Partitioning features are very useful for OLTP systems.

Due to the explosive growth of online transaction processing (OLTP) systems and their user populations, partitioning is particularly useful for OLTP systems in addition to data warehousing environments

Partitioning is often used for OLTP systems to reduce contention while supporting a very large user population. It also helps to address regulatory requirements facing OLTP systems, including storing larger amounts of data in a cost-effective manner.

This chapter contains the following sections:

- What Is an Online Transaction Processing System?
- Performance in an Online Transaction Processing Environment
- Manageability in an Online Transaction Processing Environment

7.1 What Is an Online Transaction Processing System?

An Online Transaction Processing (OLTP) system is a common data processing system in today's enterprises. Classic examples of OLTP systems are order entry, retail sales, and financial transaction systems.

OLTP systems are primarily characterized through a specific data usage that is different from data warehouse environments, yet some characteristics, such as having large volumes of data and lifecycle-related data usage and importance, are identical.

The main characteristics of an OLTP environment are:

- Short response time

  The nature of OLTP environments is predominantly any kind of interactive ad hoc usage, such as telemarketers entering telephone survey results. OLTP systems require short response times in order for users to remain productive.

- Small transactions

  OLTP systems typically read and manipulate highly selective, small amounts of data; the data processing is mostly simple and complex joins are relatively rare. There is always a mix of queries and DML workload. For example, one of many call center employees retrieves customer details for every call and enters customer complaints while reviewing past communications with the customer.

- Data maintenance operations

  It is not uncommon to have reporting programs and data updating programs that must run either periodically or on an ad hoc basis. These programs, which run in the background while users continue to work on other tasks, may require a large number of data-intensive computations. For example, a university may start batch jobs assigning students to classes while students can still sign up online for classes themselves.

- Large user populations

  OLTP systems can have enormously large user populations where many users are trying to access the same data at the same time. For example, an online auction website can have hundreds of thousands (if not millions) of users accessing data on its website at the same time.

- High concurrency

  Due to the large user population, the short response times, and small transactions, the concurrency in OLTP environments is very high. A requirement for thousands of concurrent users is not uncommon.

- Large data volumes

  Depending on the application type, the user population, and the data retention time, OLTP systems can become very large. For example, every customer of a bank could have access to the online banking system which shows all their transactions for the last 12 months.

- High availability

  The availability requirements for OLTP systems are often extremely high. An unavailable OLTP system can impact a very large user population, and organizations can suffer major losses if OLTP systems are unavailable. For example, a stock exchange system has extremely high availability requirements during trading hours.

- Lifecycle-related data usage

  Similar to data warehousing environments, OLTP systems often experience different data access patterns over time. For example, at the end of the month, monthly interest is calculated for every active account.

The following are benefits of partitioning for OLTP environments:

- Support for bigger databases

  Backup and recovery, as part of a high availability strategy, can be performed on a low level of granularity to efficiently manage the size of the database. OLTP systems usually remain online during backups and users may continue to access the system while the backup is running. The backup process should not introduce major performance degradation for the online users.

  Partitioning helps to reduce the space requirements for the OLTP system because part of a database object can be stored compressed while other parts can remain uncompressed. Update transactions against uncompressed rows are more efficient than updates on compressed data.

  Partitioning can store data transparently on different storage tiers to lower the cost of retaining vast amounts of data.

- Partition maintenance operations for data maintenance (instead of DML)

  For data maintenance operations (purging being the most common operation), you can leverage partition maintenance operations with the Oracle Database capability of online index maintenance. A partition management operation generates less redo than the equivalent DML operations.

- Potential higher concurrency through elimination of hot spots

  A common scenario for OLTP environments is to have monotonically increasing index values that are used to enforce primary key constraints, thus creating areas of high concurrency and potential contention: every new insert tries to update the same set of index blocks. Partitioned indexes, in particular hash partitioned indexes, can help alleviate this situation.

7.2 Performance in an Online Transaction Processing Environment

Performance in OLTP environments heavily relies on efficient index access, thus the choice of the most appropriate index strategy becomes crucial.

The following sections discuss best practices for deciding whether to partition indexes in an OLTP environment.

- Deciding Whether to Partition Indexes
- How to Use Partitioning on Index-Organized Tables

7.2.1 Deciding Whether to Partition Indexes

Due to the selectivity of queries and high concurrency of OLTP applications, the choice of the right index strategy is indisputably an important decision for the use of partitioning in an OLTP environment. With less contention, the application can support a larger user population.

The following basic rules explain the main benefits and trade-offs for the various possible index structures:

- A nonpartitioned index, while larger than individual partitioned index segments, always leads to a single index probe (or scan) if an index access path is chosen; there is only one segment for a table. The data access time and number of blocks being accessed are identical for both a partitioned and a nonpartitioned table.

  A nonpartitioned index does not provide partition autonomy and requires an index maintenance operation for every partition maintenance operation that affects rowids (for example, drop, truncate, move, merge, coalesce, or split operations).

- With partitioned indexes, there are always multiple segments. Whenever Oracle Database cannot prune down to a single index segment, the database has to access multiple segments. This potentially leads to higher I/O requirements (*n* index segment probes compared with one probe for a nonpartitioned index) and can have an impact (measurable or not) on the run-time performance. This is true for all partitioned indexes.

  Partitioned indexes can either be local partitioned indexes or global partitioned indexes. Local partitioned indexes always inherit the partitioning key from the table and are fully aligned with the table partitions. Consequently, any kind of partition maintenance operation requires little to no index maintenance work. For example, dropping or truncating a partition does not incur any measurable overhead for index maintenance; the local index partitions are either dropped or truncated.

  Partitioned indexes that are not aligned with the table are called global partitioned indexes. Unlike local indexes, there is no relation between a table and an index partition. Global partitioned indexes give the flexibility to choose a partitioning key that is optimal for an efficient partition index access. Partition maintenance operations normally affect more (if not all) partitions of a global partitioned index, depending on the operation and partitioning key of the index.

- Under some circumstances, having multiple segments for an index can be beneficial for performance. It is very common in OLTP environments to use sequences to create artificial keys. Consequently, you create key values that are monotonically increasing, which results in many insertion processes competing for the same index blocks. Introducing a global partitioned index (for example, using global hash partitioning on the key column) can alleviate this situation. If you have, for example, four hash partitions for such an index, then

you now have four index segments into which you are inserting data, reducing the concurrency on these segments by a factor of four for the insertion processes.

Enforcing uniqueness is important database functionality for OLTP environments. Uniqueness can be enforced with nonpartitioned and partitioned indexes. However, because partitioned indexes provide partition autonomy, the following requirements must be met to implement unique indexes:

- A nonpartitioned index can enforce uniqueness for any given column or combination of columns. The behavior of a nonpartitioned index is no different for a partitioned table compared to a nonpartitioned table.
- Each partition of a partitioned index is considered an autonomous segment. To enforce the autonomy of these segments, you always have to include the partitioning key columns as a subset of the unique key definition.
  - Global partitioned indexes must always be prefixed with at least the first leading column of the index column (the partitioning column of the partitioned global index).
  - Unique local indexes must have the partitioning key of the table as a subset of the unique key definition.

Example 7-1 shows the creation of a unique index on the `order_id` column of the `orders_oltp` table. The `order_id` in the OLTP application is filled using a sequence number. The unique index uses hash partitioning to reduce contention for the monotonically increasing `order_id` values. The unique key is then used to create the primary key constraint.

Example 7-1 Creating a unique index and primary key constraint

```
CREATE UNIQUE INDEX orders_pk
  ON orders_oltp(order_id)
  GLOBAL PARTITION BY HASH (order_id)
  ( PARTITION p1 TABLESPACE tbs1
   , PARTITION p2 TABLESPACE tbs2
   , PARTITION p3 TABLESPACE tbs3
   , PARTITION p4 TABLESPACE tbs4
  ) NOLOGGING;

ALTER TABLE orders_oltp ADD CONSTRAINT orders_pk
  PRIMARY KEY (order_id)
  USING INDEX;
```

7.2.2 How to Use Partitioning on Index-Organized Tables

When your workload fits the use of index-organized tables, then you must consider how to use partitioning on your index-organized table and on any secondary indexes.

You must decide whether to partition secondary indexes on index-organized tables based on the same considerations as indexes on regular heap tables. You can partition an index-organized table, but the partitioning key must be a subset of the primary key. A common reason to partition an index-organized table is to reduce contention; this is typically achieved using hash partitioning.

Another reason to partition an index-organized table is to be able to physically separate data sets based on a primary key column. For example, an application-hosting company can physically separate application instances for different customers by list partitioning on the company identifier. Queries in such a scenario can often take advantage of index partition pruning, shortening the time for the index scan. ILM scenarios with index-organized tables and partitioning are less common because they require a date column to be part of the primary key.

For more information about how to create partitioned index-organized tables, refer to Partition Administration.

> **See Also:**
>
> *Oracle Database Administrator's Guide* for more information about index-organized tables

7.3 Manageability in an Online Transaction Processing Environment

In addition to the performance benefits, partitioning also enables the optimal data management for large objects in an OLTP environment.

Every partition maintenance operation in Oracle Database can be extended to atomically include global and local index maintenance, enabling the execution of any partition maintenance operation without affecting the 24x7 availability of an OLTP environment.

Partition maintenance operations in OLTP systems occur often because of ILM scenarios. In these scenarios, [range | interval] partitioned tables, or [range | interval]-* composite partitioned tables, are common.

Some business cases for partition maintenance operations include scenarios surrounding the separation of application data. For example, a retail company runs the same application for multiple branches in a single schema. Depending on the branch revenues, the application (as separate partitions) is stored on more efficient storage. List partitioning, or list-* composite partitioning, is a common partitioning strategy for this type of business case.

You can use hash partitioning, or hash subpartitioning for tables, in OLTP systems to obtain similar performance benefits to the performance benefits achieved in data warehouse environments. The majority of the daily OLTP workload consists of relatively small operations, executed serially. Periodic batch operations, however, may execute in parallel and benefit from the distribution improvements that hash partitioning and subpartitioning can provide for partition-wise joins. For example, end-of-the-month interest calculations may be executed in parallel to complete within a nightly batch window.

This section contains the following topics:

- Impact of a Partition Maintenance Operation on a Partitioned Table with Local Indexes
- Impact of a Partition Maintenance Operation on Global Indexes
- Common Partition Maintenance Operations in OLTP Environments

For more information about the performance benefits of partitioning, refer to Partitioning for Availability, Manageability, and Performance.

7.3.1 Impact of a Partition Maintenance Operation on a Partitioned Table with Local Indexes

When a partition maintenance operation takes place, Oracle Database locks the affected table partitions for any DML operation, except in the case of an `ONLINE MOVE`.

Data in the affected partitions, except a `DROP` or `TRUNCATE` operation, is still fully accessible for any `SELECT` operation. Because local indexes are logically coupled with the table (data) partitions, only the local index partitions of the affected table partitions have to be maintained as part of a partition maintenance operation, enabling optimal processing for the index maintenance.

For example, when you move an older partition from a high-end storage tier to a low-cost storage tier using the `ALTER TABLE MOVE ONLINE` functionality, the data and the index are always available for `SELECT` and DML operations; the necessary index maintenance is to update the existing index partition to reflect the new physical location of the data. If you drop an older partition after you have archived it, then its local index partitions get dropped as well, and for global indexes the orphaned entries for the removed partitions get marked, enabling a split-second partition maintenance operation that affects only the data dictionary.

7.3.2 Impact of a Partition Maintenance Operation on Global Indexes

Whenever a global index is defined on a partitioned or nonpartitioned table, there is no correlation between a distinct table partition and the index. Consequently, any partition maintenance operation affects all global indexes or index partitions.

The partitions of tables containing local indexes are locked to prevent DML operations against the affect table partitions, except for an `ONLINE MOVE` operation. However, unlike the index maintenance for local indexes, any global index is still fully available for DML operations and does not affect the online availability of the OLTP system.

Conceptually and technically, the index maintenance for global indexes for a partition maintenance operation is comparable to the index maintenance that would become necessary for a semantically identical DML operation, except for `DROP` and `TRUNCATE` where the global index maintenance gets delayed to a later point in time. For more information about managing global indexes, refer to Management of Global Partitioned Indexes.

For example, dropping an old partition is semantically equivalent to deleting all the records of the old partition using the SQL `DELETE` statement. In the DML case, all index entries of the deleted data set have to be removed from any global index as a standard index maintenance operation, which does not affect the availability of an index for `SELECT` and DML operations.

The `DROP PARTITION` also does not affect the index availability, but enables you to decouple the necessary index maintenance from the initial data removal without affecting the availability of the global indexes. In this scenario, a drop operation represents the optimal approach: data is removed without the overhead of a conventional `DELETE` operation and the global indexes are maintained in a nonintrusive manner.

7.3.3 Common Partition Maintenance Operations in OLTP Environments

The two most common partition maintenance operations are the removal of data and the relocation of data onto lower-cost storage tier devices.

- Removing (Purging) Old Data
- Moving or Merging Older Partitions to a Low-Cost Storage Tier Device

7.3.3.1 Removing (Purging) Old Data

Using either a `DROP` or `TRUNCATE` operation removes older data based on the partitioning key criteria.

The drop operation removes the data and the partition metadata, while a `TRUNCATE` operation removes only the data but preserve the metadata. All local index partitions are dropped respectively, and truncated. Asynchronous global index maintenance is done for partitioned or nonpartitioned global indexes and is fully available for select and DML operations.

The following example drops all data older than January 2006 from the `orders_oltp` table. As part of the drop statement, an `UPDATE GLOBAL INDEXES` statement is executed, so that the global index remains usable throughout the maintenance operation. Any local index partitions are dropped as part of this operation.

```
ALTER TABLE orders_oltp DROP PARTITION p_before_jan_2006
 UPDATE GLOBAL INDEXES;
```

7.3.3.2 Moving or Merging Older Partitions to a Low-Cost Storage Tier Device

Using a `MOVE` or `MERGE` operation as part of an Information Lifecycle Management (ILM) strategy, you can relocate older partitions to the most cost-effective storage tier.

Using the `ALTER TABLE ONLINE MOVE` functionality enables the data to be available for both queries and DML operations. Local indexes are maintained and likely relocated as part of the merge or move operation. The standard index maintenance is done for partitioned or nonpartitioned global indexes and is fully available for select and DML operations.

The following example shows how to merge the January 2006 and February 2006 partitions in the `orders_oltp` table, and store them in a different tablespace. Any local index partitions are also moved to the `ts_low_cost` tablespace as part of this operation. The `UPDATE INDEXES` clause ensures that all indexes remain usable throughout and after the operation, without additional rebuilds.

```
ALTER TABLE orders_oltp
  MERGE PARTITIONS p_2006_jan,p_2006_feb
  INTO PARTITION p_before_mar_2006 COMPRESS
  TABLESPACE ts_low_cost
  UPDATE INDEXES;
```

For more information about the benefits of partition maintenance operations for Information Lifecycle Management, see Managing and Maintaining Time-Based Information.

8
Using Parallel Execution

Parallel execution is the ability to apply multiple CPU and I/O resources to the execution of a single SQL statement by using multiple processes.

This chapter explains how parallel execution works, and how to control, manage, and monitor parallel execution in the Oracle Database.

This chapter contains the following sections:

- Parallel Execution Concepts
- Setting the Degree of Parallelism
- In-Memory Parallel Execution
- Parallel Statement Queuing
- Types of Parallelism
- About Initializing and Tuning Parameters for Parallel Execution
- Monitoring Parallel Execution Performance
- Tips for Tuning Parallel Execution

> **See Also:**
>
> `http://www.oracle.com/technetwork/database/database-technologies/parallel-execution/overview/index.html` for information about the parallel execution with Oracle Database

8.1 Parallel Execution Concepts

Parallel execution enables the application of multiple CPU and I/O resources to the execution of a single SQL statement.

Parallel execution dramatically reduces response time for data-intensive operations on large databases typically associated with a decision support system (DSS) and data warehouses. You can also implement parallel execution on an online transaction processing (OLTP) system for batch processing or schema maintenance operations, such as index creations.

Parallel execution is sometimes called parallelism. Parallelism is the idea of breaking down a task so that, instead of one process doing all of the work in a query, many processes do part of the work at the same time. An example of this is when four processes combine to calculate the total sales for a year, each process handles one quarter of the year instead of a single process handling all four quarters by itself. The improvement in performance can be quite significant.

Parallel execution improves processing for:

- Queries requiring large table scans, joins, or partitioned index scans
- Creation of large indexes

- Creation of large tables, including materialized views
- Bulk insertions, updates, merges, and deletions

This section contains the following topics:

- When to Implement Parallel Execution
- When Not to Implement Parallel Execution
- Fundamental Hardware Requirements
- How Parallel Execution Works
- Parallel Execution Server Pool
- Balancing the Workload to Optimize Performance
- Multiple Parallelizers
- Parallel Execution on Oracle RAC

8.1.1 When to Implement Parallel Execution

Parallel execution is used to reduce the execution time of queries by exploiting the CPU and I/O capabilities in the hardware.

Parallel execution is a better choice than serial execution when:

- The query references a large data set.
- There is low concurrency.
- Elapsed time is important.

Parallel execution enables many processes working together to execute single operation, such as a SQL query. Parallel execution benefits systems with all of the following characteristics:

- Symmetric multiprocessors (SMPs), clusters, or massively parallel systems
- Sufficient I/O bandwidth
- Underutilized or intermittently used CPUs (for example, systems where CPU usage is typically less than 30%)
- Sufficient memory to support additional memory-intensive processes, such as sorting, hashing, and I/O buffers

If your system lacks any of these characteristics, parallel execution might not significantly improve performance. In fact, parallel execution may reduce system performance on overutilized systems or systems with small I/O bandwidth.

The benefits of parallel execution can be observed in DSS and data warehouse environments. OLTP systems can also benefit from parallel execution during batch processing and during schema maintenance operations such as creation of indexes. The average simple DML or `SELECT` statements that characterize OLTP applications would not experience any benefit from being executed in parallel.

8.1.2 When Not to Implement Parallel Execution

Serial execution is different than parallel execution in that only one process executes a single database operation, such as a SQL query.

Serial execution is a better choice than parallel execution when:

- The query references a small data set.
- There is high concurrency.
- Efficiency is important.

Parallel execution is not typically useful for:

- Environments in which the typical query or transaction is very short (a few seconds or less).

  This includes most online transaction systems. Parallel execution is not useful in these environments because there is a cost associated with coordinating the parallel execution servers; for short transactions, the cost of this coordination may outweigh the benefits of parallelism.
- Environments in which the CPU, memory, or I/O resources are heavily used.

  Parallel execution is designed to exploit additional available hardware resources; if no such resources are available, then parallel execution does not yield any benefits and indeed may be detrimental to performance.

8.1.3 Fundamental Hardware Requirements

Parallel execution is designed to effectively use multiple CPUs and disks to answer queries quickly.

It is very I/O intensive by nature. To achieve optimal performance, each component in the hardware configuration must be sized to sustain the same level of throughput: from the CPUs and the Host Bus Adapters (HBAs) in the compute nodes, to the switches, and on into the I/O subsystem, including the storage controllers and the physical disks. If the system is an Oracle Real Application Clusters (Oracle RAC) system, then the interconnection also has to be sized appropriately. The weakest link is going to limit the performance and scalability of operations in a configuration.

It is recommended to measure the maximum I/O performance that a hardware configuration can achieve without Oracle Database. You can use this measurement as a baseline for the future system performance evaluations. Remember, it is not possible for parallel execution to achieve better I/O throughput than the underlying hardware can sustain. Oracle Database provides a free calibration tool called Orion, which is designed to measure the I/O performance of a system by simulating Oracle I/O workloads. A parallel execution typically performs large random I/Os.

> **See Also:**
>
> *Oracle Database Performance Tuning Guide* for information about I/O configuration and design

8.1.4 How Parallel Execution Works

Parallel execution breaks down a task so that, instead of one process doing all of the work in a query, many processes do part of the work at the same time.

This section contains the following topics:

- Parallel Execution of SQL Statements

- Producer/Consumer Model
- Granules of Parallelism
- Distribution Methods Between Producers and Consumers
- How Parallel Execution Servers Communicate

8.1.4.1 Parallel Execution of SQL Statements

Each SQL statement undergoes an optimization and parallelization process when it is parsed.

If the statement is determined to be executed in parallel, then the following steps occur in the execution plan:

1. The user session or shadow process takes on the role of a coordinator, often called the query coordinator (QC) or the parallel execution (PX) coordinator. The QC is the session that initiates the parallel SQL statement.
2. The PX coordinator obtains the necessary number of processes called parallel execution (PX) servers. The PX servers are the individual processes that perform work in parallel on behalf of the initiating session.
3. The SQL statement is executed as a sequence of operations, such as a full table scan or an `ORDER BY` clause. Each operation is performed in parallel if possible.
4. When the PX servers are finished executing the statement, the PX coordinator performs any portion of the work that cannot be executed in parallel. For example, a parallel query with a `SUM()` operation requires adding the individual subtotals calculated by each PX server.
5. Finally, the PX coordinator returns the results to the user.

8.1.4.2 Producer/Consumer Model

Parallel execution uses the producer/consumer model.

A parallel execution plan is carried out as a series of producer/consumer operations. Parallel execution (PX) servers that produce data for subsequent operations are called producers, PX servers that require the output of other operations are called consumers. Each producer or consumer parallel operation is performed by a set of PX servers called PX server sets. The number of PX servers in PX server set is called Degree of Parallelism (DOP). The basic unit of work for a PX server set is called a data flow operation (DFO).

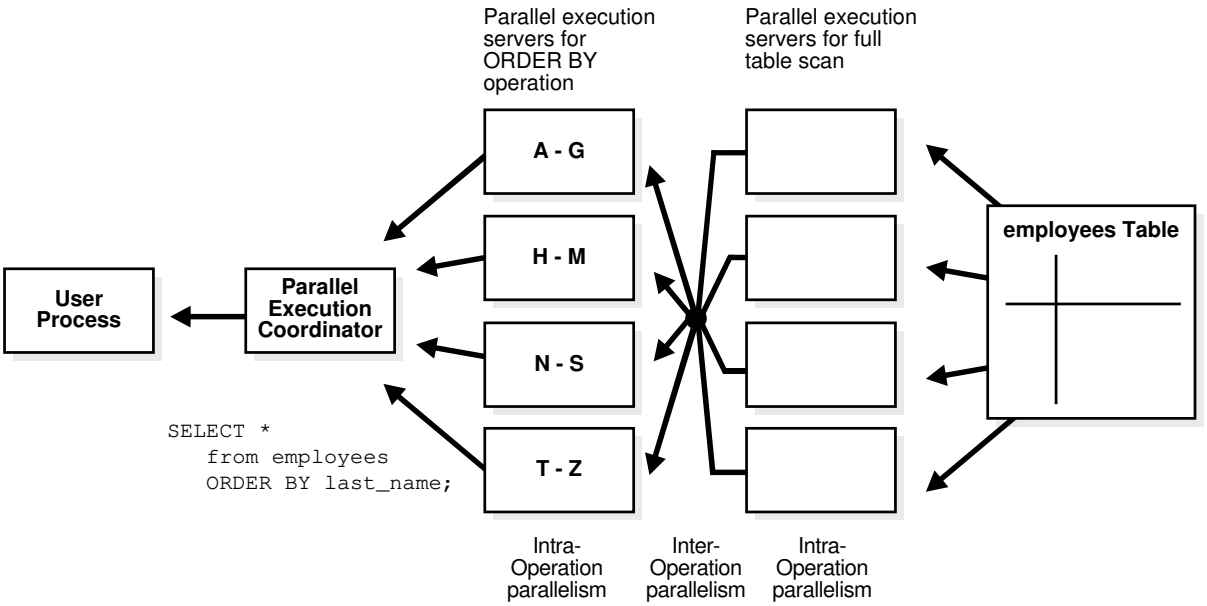
A PX coordinator can have multiple levels of producer/consumer operations (multiple DFOs), but the number of PX servers sets for a PX coordinator is limited to two. Therefore, at one point in time only two PX server sets can be active for a PX coordinator. As a result, there is parallelism in both the operations in a DFO and between DFOs. The parallelism of an individual DFO is called intra-operation parallelism and the parallelism between DFOs is called inter-operation parallelism. To illustrate intra- and inter-operation parallelism, consider the following statement:

```
SELECT * FROM employees ORDER BY last_name;
```

The execution plan implements a full scan of the `employees` table. This operation is followed by a sorting of the retrieved rows, based on the value of the `last_name` column. For the sake of this example, assume the `last_name` column is not indexed. Also assume that the DOP for the query is set to `4`, which means that four parallel execution servers can be active for any given operation.

Figure 8-1 illustrates the parallel execution of the example query.

Figure 8-1 Inter-operation Parallelism and Dynamic Partitioning



As illustrated in Figure 8-1, there are actually eight PX servers involved in the query even though the DOP is `4`. This is because a producer and consumer operator can be performed at the same time (inter-operation parallelism).

Also all of the PX servers involved in the scan operation send rows to the appropriate PX server performing the `SORT` operation. If a row scanned by a PX server contains a value for the `last_name` column between `A` and `G`, that row is sent to the first `ORDER BY` parallel execution server. When the scan operation is complete, the sorting processes can return the sorted results to the query coordinator, which returns the complete query results to the user.

8.1.4.3 Granules of Parallelism

The basic unit of work in parallelism is a called a granule.

Oracle Database divides the operation executed in parallel, such as a table scan or index creation, into granules. Parallel execution (PX) servers execute the operation one granule at a time. The number of granules and their sizes correlate with the degree of parallelism (DOP). The number of granules also affect how well the work is balanced across PX servers.

8.1.4.3.1 Block Range Granules

Block range granules are the basic unit of most parallel operations, even on partitioned tables. From an Oracle Database perspective, the degree of parallelism is not related to the number of partitions.

Block range granules are ranges of physical blocks from a table. Oracle Database computes the number and the size of the granules during run-time to optimize and balance the work distribution for all affected parallel execution (PX) servers. The number and size of granules are dependent upon the size of the object and the DOP. Block range granules do not depend on static preallocation of tables or indexes. During the computation of the granules, Oracle Database takes the DOP into account and tries to assign granules from different data files to each of the PX servers to avoid contention whenever possible. Additionally, Oracle Database

considers the disk affinity of the granules on massive parallel processing (MPP) systems to take advantage of the physical proximity between PX servers and disks.

8.1.4.3.2 Partition Granules

When partition granules are used, a parallel execution (PX) server works on an entire partition or subpartition of a table or index.

Because partition granules are statically determined by the structure of the table or index when a table or index is created, partition granules do not give you the flexibility in executing an operation in parallel that block granules do. The maximum allowable degree of parallelism (DOP) is the number of partitions. This might limit the utilization of the system and the load balancing across PX servers.

When partition granules are used for parallel access to a table or index, you should use a relatively large number of partitions, ideally three times the DOP, so that Oracle Database can effectively balance work across the PX servers.

Partition granules are the basic unit of parallel index range scans, joins between two equipartitioned tables where the query optimizer has chosen to use partition-wise joins, and parallel operations that modify multiple partitions of a partitioned object. These operations include parallel creation of partitioned indexes, and parallel creation of partitioned tables.

You can tell which types of granules were used by looking at the execution plan of a statement. The line `PX BLOCK ITERATOR` above the table or index access indicates that block range granules have been used. In the following example, you can see this on line 7 of the explain plan output just above the `TABLE FULL ACCESS` on the `SALES` table.

```
-------------------------------------------------------------------------------------------------
|Id|      Operation          |  Name  |Rows|Bytes|Cost%CPU|  Time  |Pst|Pst|  TQ |INOUT|PQDistri|
-------------------------------------------------------------------------------------------------
| 0|SELECT STATEMENT         |        |  17| 153 |565(100)|00:00:07|   |   |     |     |        |
| 1| PX COORDINATOR          |        |    |     |        |        |   |   |     |     |        |
| 2|  PX SEND QC(RANDOM)     |:TQ10001|  17| 153 |565(100)|00:00:07|   |   |Q1,01|P->S |QC(RAND)|
| 3|   HASH GROUP BY         |        |  17| 153 |565(100)|00:00:07|   |   |Q1,01|PCWP |        |
| 4|    PX RECEIVE           |        |  17| 153 |565(100)|00:00:07|   |   |Q1,01|PCWP |        |
| 5|     PX SEND HASH        |:TQ10000|  17| 153 |565(100)|00:00:07|   |   |Q1,00|P->P | HASH   |
| 6|      HASH GROUP BY      |        |  17| 153 |565(100)|00:00:07|   |   |Q1,00|PCWP |        |
| 7|       PX BLOCK ITERATOR |        | 10M| 85M | 60(97) |00:00:01| 1 | 16|Q1,00|PCWC |        |
|*8|        TABLE ACCESS FULL|  SALES | 10M| 85M | 60(97) |00:00:01| 1 | 16|Q1,00|PCWP |        |
-------------------------------------------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------
8 - filter("CUST_ID"<=22810 AND "CUST_ID">=22300)
```

When partition granules are used, you see the line `PX PARTITION RANGE` above the table or index access in the explain plan output. On line 6 of the example that follows, the plan has `PX PARTITION RANGE ALL` because this statement accesses all of the 16 partitions in the table. If not all of the partitions are accessed, it simply shows `PX PARTITION RANGE`.

```
-------------------------------------------------------------------------------------------------
|Id|      Operation                   |  Name    |Rows|Byte|Cost%CPU|  Time  |Ps|Ps|  TQ |INOU|PQDistri|
-------------------------------------------------------------------------------------------------
| 0|SELECT STATEMENT                  |          |  17| 153|   2(50)|00:00:01|  |  |     |    |        |
| 1| PX COORDINATOR                   |          |    |    |        |        |  |  |     |    |        |
| 2|  PX SEND QC(RANDOM)              |:TQ10001  |  17| 153|   2(50)|00:00:01|  |  |Q1,01|P->S|QC(RAND)|
| 3|   HASH GROUP BY                  |          |  17| 153|   2(50)|00:00:01|  |  |Q1,01|PCWP|        |
| 4|    PX RECEIVE                    |          |  26| 234|    1(0)|00:00:01|  |  |Q1,01|PCWP|        |
| 5|     PX SEND HASH                 |:TQ10000  |  26| 234|    1(0)|00:00:01|  |  |Q1,00|P->P| HASH   |
| 6|      PX PARTITION RANGE ALL      |          |  26| 234|    1(0)|00:00:01|  |  |Q1,00|PCWP|        |
| 7|       TABLEACCESSLOCAL INDEX ROWID|SALES     |  26| 234|    1(0)|00:00:01| 1|16|Q1,00|PCWC|        |
|*8|        INDEX RANGE SCAN          |SALES_CUST|  26|    |    1(0)|00:00:01| 1|16|Q1,00|PCWP|        |
-------------------------------------------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------
8 - access("CUST_ID"<=22810 AND "CUST_ID">=22300)
```

8.1.4.4 Distribution Methods Between Producers and Consumers

A distribution method is the method by which data is sent (or redistributed) from one parallel execution (PX) server set to another.

The following are the most commonly used distribution methods in parallel execution.

- Hash Distribution

  The hash distribution method uses a hash function on one or more columns in the row which then determines the consumer where the producer should send the row. This distribution attempts to divide the work equally among consumers based on hash values.

- Broadcast Distribution

  In the broadcast distribution method, each producer sends all rows to all consumers. This method is used when the result set of the left side in a join operation is small and the cost of broadcasting all rows is not high. The result set from the right side of the join does not need to be distributed in this case; consumer PX servers assigned to the join operation can scan the right side and perform the join.

- Range Distribution

  Range distribution is mostly used in parallel sort operations. In this method each producer sends rows that have a range of values to the same consumer. This is the method used in Figure 8-1.

- Hybrid Hash Distribution

  Hybrid hash is an adaptive distribution method used in join operations. The actual distribution method is decided at runtime by the optimizer depending on the size of the result set of the left side of the join. The number of rows returned from the left side is counted and checked against a threshold value. When the number of rows is less than or equal to the threshold value, broadcast distribution is used for the left side of the join, and the right side is not distributed as the same consumer PX servers assigned to the join operation scan the right side and perform the join. When the number of rows returned from the left side is higher than the threshold value, hash distribution is used for both sides of the join.

To determine the distribution method, the parallel execution (PX) coordinator examines each operation in a SQL statement's execution plan and then determines the way in which the rows operated on by the operation must be redistributed among the PX servers. As an example of parallel query, consider the query in Example 8-1. Figure 8-2 illustrates the data flow or query plan for the query in Example 8-1, and Example 8-2 shows the explain plan output for the same query.

The query plan shows that an adaptive distribution methods was picked by the PX coordinator. Assuming the optimizer picks hash distribution at runtime, the execution proceeds as follows: two sets of PX servers, SS1 and SS2, are allocated for the query, each server set has four PX servers because of the `PARALLEL` hint that specifies the DOP of the statement.

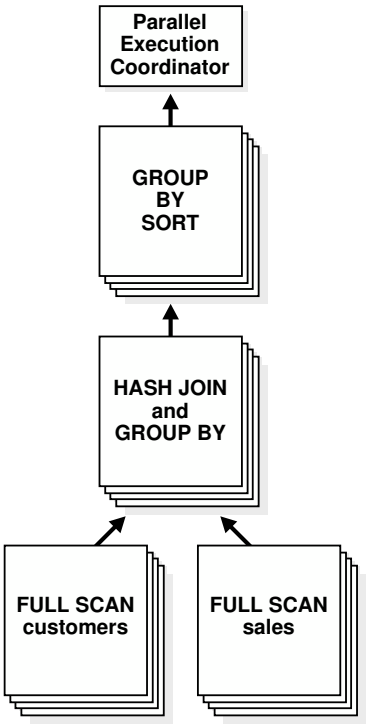
PX set SS1 first scans the table `customers` and sends rows to SS2, which builds a hash table on the rows. In other words, the consumers in SS2 and the producers in SS1 work concurrently: one in scanning `customers` in parallel, the other is consuming rows and building the hash table to enable the hash join in parallel. This is an example of inter-operation parallelism.

After a PX server process in SS1 scans a row from the customers table, which PX server process in SS2 should it send it to? In this case, the redistribution of rows flowing up from SS1 performing the parallel scan of customers into SS2 performing the parallel hash-join is done by

hash distribution on the join column. That is, a PX server process scanning customers computes a hash function on the value of the column customers.cust_id to decide which PX server process in SS2 to send it to. The redistribution method used is explicitly shown in the Distrib column in the `EXPLAIN PLAN` of the query. In Figure 8-2, this can be seen on lines 5, 9, and 14 of the `EXPLAIN PLAN`.

After SS1 has finished scanning the entire `customers` table, it scans the `sales` table in parallel. It sends its rows to PX servers in SS2, which then perform the probes to finish the hash join in parallel. These PX servers also perform a `GROUP BY` operation after the join. After SS1 has scanned the `sales` table in parallel and sent the rows to SS2, it switches to performing the final group by operation in parallel. At this point the PX servers in SS2 send their rows using hash distribution to PX servers on SS1 for the group by operation. This is how two server sets run concurrently to achieve inter-operation parallelism across various operators in the query tree.

Figure 8-2 Data Flow Diagram for Joining Tables



Example 8-1 Running an Explain Plan for a Query on Customers and Sales

```
EXPLAIN PLAN FOR
SELECT /*+ PARALLEL(4) */ customers.cust_first_name, customers.cust_last_name,
  MAX(QUANTITY_SOLD), AVG(QUANTITY_SOLD)
  FROM sales, customers
  WHERE sales.cust_id=customers.cust_id
  GROUP BY customers.cust_first_name, customers.cust_last_name;

Explained.
```

Example 8-2 Explain Plan Output for a Query on Customers and Sales

```
PLAN_TABLE_OUTPUT
-------------------------------------------------------------------------------------------------------------------------------------------
Plan hash value: 3260900439
-------------------------------------------------------------------------------------------------------------------------------------------
|Id  |Operation                    |Name     |Rows  | Bytes |TempSpc|Cost (%CPU)| Time     |Pstart|Pstop |   TQ  |IN-OUT|PQ Distrib |
```

```
-----------------------------------------------------------------------------------------------------------------------------------
|  0 |SELECT STATEMENT               |          |  960 | 26880 |       |    6  (34)| 00:00:01 |      |      |       |      |           |
|  1 | PX COORDINATOR                |          |      |       |       |           |          |      |      |       |      |           |
|  2 |  PX SEND QC (RANDOM)          |:TQ10003  |  960 | 26880 |       |    6  (34)| 00:00:01 |      |      | Q1,03 | P->S |QC (RAND)  |
|  3 |   HASH GROUP BY               |          |  960 | 26880 | 50000 |    6  (34)| 00:00:01 |      |      | Q1,03 | PCWP |           |
|  4 |    PX RECEIVE                 |          |  960 | 26880 |       |    6  (34)| 00:00:01 |      |      | Q1,03 | PCWP |           |
|  5 |     PX SEND HASH              |:TQ10002  |  960 | 26880 |       |    6  (34)| 00:00:01 |      |      | Q1,02 | P->P |HASH       |
|  6 |      HASH GROUP BY            |          |  960 | 26880 | 50000 |    6  (34)| 00:00:01 |      |      | Q1,02 | PCWP |           |
|* 7 |       HASH JOIN               |          |  960 | 26880 |       |    5  (20)| 00:00:01 |      |      | Q1,02 | PCWP |           |
|  8 |        PX RECEIVE             |          |  630 | 12600 |       |    2   (0)| 00:00:01 |      |      | Q1,02 | PCWP |           |
|  9 |         PX SEND HYBRID HASH   |:TQ10000  |  630 | 12600 |       |    2   (0)| 00:00:01 |      |      | Q1,00 | P->P |HYBRID HASH|
| 10 |          STATISTICS COLLECTOR |          |      |       |       |           |          |      |      | Q1,00 | PCWC |           |
| 11 |           PX BLOCK ITERATOR   |          |  630 | 12600 |       |    2   (0)| 00:00:01 |      |      | Q1,00 | PCWC |           |
| 12 |            TABLE ACCESS FULL  |CUSTOMERS |  630 | 12600 |       |    2   (0)| 00:00:01 |      |      | Q1,00 | PCWP |           |
| 13 |        PX RECEIVE             |          |  960 |  7680 |       |    2   (0)| 00:00:01 |      |      | Q1,02 | PCWP |           |
| 14 |         PX SEND HYBRID HASH   |:TQ10001  |  960 |  7680 |       |    2   (0)| 00:00:01 |      |      | Q1,01 | P->P |HYBRID HASH|
| 15 |          PX BLOCK ITERATOR    |          |  960 |  7680 |       |    2   (0)| 00:00:01 |    1 |   16 | Q1,01 | PCWC |           |
| 16 |           TABLE ACCESS FULL   |SALES     |  960 |  7680 |       |    2   (0)| 00:00:01 |    1 |   16 | Q1,01 | PCWP |           |
-----------------------------------------------------------------------------------------------------------------------------------
Predicate Information (identified by operation id):
---------------------------------------------------
7 - access("SALES"."CUST_ID"="CUSTOMERS"."CUST_ID")
Note
-----
   - Degree of Parallelism is 4 because of hint
```

8.1.4.5 How Parallel Execution Servers Communicate

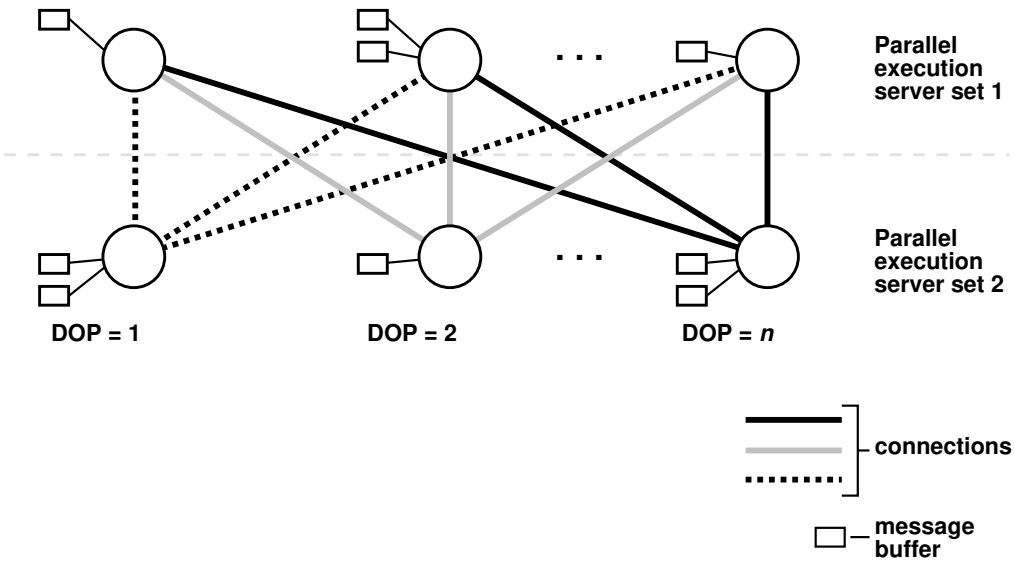
To execute a query in parallel, Oracle Database generally creates a set of producer parallel execution servers and a set of consumer parallel execution servers.

The producer server retrieves rows from tables and the consumer server performs operations such as join, sort, DML, and DDL on these rows. Each server in the producer set has a connection to each server in the consumer set. The number of virtual connections between parallel execution servers increases as the square of the degree of parallelism.

Each communication channel has at least one, and sometimes up to four memory buffers, which are allocated from the shared pool. Multiple memory buffers facilitate asynchronous communication among the parallel execution servers.

A single-instance environment uses at most three buffers for each communication channel. An Oracle Real Application Clusters environment uses at most four buffers for each channel. Figure 8-3 illustrates message buffers and how producer parallel execution servers connect to consumer parallel execution servers.

Figure 8-3 Parallel Execution Server Connections and Buffers



When a connection is between two processes on the same instance, the servers communicate by passing the buffers back and forth in memory (in the shared pool). When the connection is between processes in different instances, the messages are sent using external high-speed network protocols over the interconnect. In Figure 8-3, the DOP equals the number of parallel execution servers, which in this case is n. Figure 8-3 does not show the parallel execution coordinator. Each parallel execution server actually has an additional connection to the parallel execution coordinator. It is important to size the shared pool adequately when using parallel execution. If there is not enough free space in the shared pool to allocate the necessary memory buffers for a parallel server, it fails to start.

8.1.5 Parallel Execution Server Pool

When an instance starts, Oracle Database creates a pool of parallel execution servers, which are available for any parallel operation.

The initialization parameter `PARALLEL_MIN_SERVERS` specifies the number of parallel execution servers that Oracle Database creates at instance startup.

When executing a parallel operation, the parallel execution coordinator obtains parallel execution servers from the pool and assigns them to the operation. If necessary, Oracle Database can create additional parallel execution servers for the operation. These parallel execution servers remain with the operation throughout execution. After the statement has been processed, the parallel execution servers return to the pool.

> **Note:**
>
> Parallel processes can be released pro-actively before individual statements using parallelism are finished. For example, an uncommitted parallel DML operation or a partially fetched parallel `SELECT` statement with two Parallel Server Sets (Producer-Consumer) will release one of the Parallel Server Sets to the server pool as soon as it has finished working, freeing half of the parallel process for use by other statements. You can use the `V$PQ_SESSTAT` view to monitor the number of servers released proactively in this way.

If the number of parallel operations increases, Oracle Database creates additional parallel execution servers to handle incoming requests. However, Oracle Database never creates more parallel execution servers for an instance than the value specified by the initialization parameter `PARALLEL_MAX_SERVERS`.

If the number of parallel operations decreases, Oracle Database terminates any parallel execution servers that have been idle for a threshold interval. Oracle Database does not reduce the size of the pool less than the value of `PARALLEL_MIN_SERVERS`, no matter how long the parallel execution servers have been idle.

> **See Also:**
>
> V$PQ_SESSTAT in the *Oracle Database Reference*.

8.1.5.1 Processing without Enough Parallel Execution Servers

Oracle Database can process a parallel operation with fewer than the requested number of processes.

If all parallel execution servers in the pool are occupied and the maximum number of parallel execution servers has been started, the parallel execution coordinator switches to serial processing.

See Also:

- Tuning General Parameters for Parallel Execution for information about the `PARALLEL_MIN_PERCENT` and `PARALLEL_MAX_SERVERS` initialization parameters
- *Oracle Database Reference* for information about using the initialization parameter `PARALLEL_MIN_PERCENT`

8.1.6 Balancing the Workload to Optimize Performance

To optimize performance, all parallel execution servers should have equal workloads.

For SQL statements run in parallel by block range or by parallel execution servers, the workload is dynamically divided among the parallel execution servers. This minimizes workload skewing, which occurs when some parallel execution servers perform significantly more work than the other processes.

For the relatively few SQL statements executed in parallel by partitions, if the workload is evenly distributed among the partitions, you can optimize performance by matching the number of parallel execution servers to the number of partitions or by choosing a DOP in which the number of partitions is a multiple of the number of processes. This applies to partition-wise joins and parallel DML on tables created before Oracle9*i*. Refer to Limitation on the Degree of Parallelism for more information.

For example, suppose a table has 16 partitions, and a parallel operation divides the work evenly among them. You can use 16 parallel execution servers (DOP equals 16) to do the work in approximately one-tenth the time that one process would take. You might also use five processes to do the work in one-fifth the time, or two processes to do the work in one-half the time.

If, however, you use 15 processes to work on 16 partitions, the first process to finish its work on one partition then begins work on the 16th partition; and as the other processes finish their work, they become idle. This configuration does not provide good performance when the work is evenly divided among partitions. When the work is unevenly divided, the performance varies depending on whether the partition that is left for last has more or less work than the other partitions.

Similarly, suppose you use six processes to work on 16 partitions and the work is evenly divided. In this case, each process works on a second partition after finishing its first partition, but only four of the processes work on a third partition while the other two remain idle.

In general, you cannot assume that the time taken to perform a parallel operation on a given number of partitions (N) with a given number of parallel execution servers (P) equals N divided by P. This formula does not consider the possibility that some processes might have to wait while others finish working on the last partitions. By choosing an appropriate DOP, however, you can minimize the workload skew and optimize performance.

8.1.7 Multiple Parallelizers

Each parallel execution (PX) coordinator in an execution plan is called a parallelizer.

The number of PX servers used by a SQL statement is determined by the statement degree of parallelism (DOP) and the number of parallelizers. Because the number of PX server sets for a parallelizer is limited to two, the number of PX servers for most statements is DOP*2. Some statements can have more than one parallelizer. Because each parallelizer can use two PX server sets, the number of PX servers for these statements can be more than DOP*2. You can identify these statements by looking at the `EXPLAIN PLAN`. If the plan has multiple PX coordinators it means the statement has multiple parallelizers.

A few example cases where SQL statements use multiple parallelizers are subquery factoring, grouping sets, star queries, in-memory aggregation, and noncorrelated subqueries.

Multiple parallelizers in a SQL statement can be active concurrently or one after the other depending on the execution plan.

A statement with a single parallelizer allocates the required number of PX servers at the start of execution and holds these allocated PX servers without releasing until the statement completes. This ensures that the number of PX servers throughout the execution is constant. Statements with multiple parallelizers are different as they allocate PX servers when each parallelizer starts. Because parallelizers can start at different times during the execution, each parallelizer may be running with a different number of PX servers based on the number of available processes in the system.

If multiple parallelizers are executed concurrently the statement can use more PX servers than DOP*2.

The view `V$PQ_SESSTAT` shows the number of parallelizers in the `STATISTIC` column. The data flow operation statistic,`DFO Trees` , shows the number of parallelizers. The `Server Threads` statistic shows the maximum number of PX servers used concurrently for a SQL statement.

> **See Also:**
>
> *Oracle Database Reference* for information about `V$PQ_SESSTAT` and other dynamic views

8.1.8 Parallel Execution on Oracle RAC

By default in an Oracle RAC environment, a SQL statement executed in parallel can run across all the nodes in the cluster.

For this cross-node or inter-node parallel execution to perform, the interconnect in the Oracle RAC environment must be sized appropriately because inter-node parallel execution may result in heavy interconnect traffic. Inter-node parallel execution does not scale with an undersized interconnect.

Limiting the Number of Available Instances

In an Oracle RAC environment, you can use services to limit the number of instances that participate in the execution of a parallel SQL statement. The default service includes all available instances. You can create any number of services, each consisting of one or more instances. When a user connects to the database using a service, only PX servers on the instances that are members of the service can participate in the execution of a parallel statement.

To limit parallel execution to a single node, you can set the `PARALLEL_FORCE_LOCAL` initialization parameter to `TRUE`. In this case, only PX servers on the instance that a session connects to is

used to execute parallel statements from that session. Note that when this parameter is set to `TRUE`, all parallel statements running on that instance are executed locally, whether the session connects to the instance directly or connects using a service.

Parallel Execution on Flex Clusters

Parallel statements executed on flex clusters can use both hub and leaf nodes. As user sessions are only allowed to connect to the hub nodes, the coordinator process (Query Coordinator or PX Coordinator) resides on hub nodes and can use PX server processes from any node in the cluster. For parallel queries any PX server on any node can participate in the execution of the statement. For parallel DML operations only PX servers on hub nodes can participate in the execution of the DML part of the statement as only hub nodes are allowed to perform DML operations.

When there is data distribution from the leaf nodes to the hub nodes for DML operations, the execution plan indicates this distribution. In the following example, data is distributed to hub nodes in line `Id 5`, indicating the load operation in line `Id 3` is executed only on hub nodes.

```
-----------------------------------------------------
| Id  | Operation                          | Name      |
-----------------------------------------------------
|   0 | CREATE TABLE STATEMENT             |           |
|   1 |  PX COORDINATOR                    |           |
|   2 |   PX SEND QC (RANDOM)              | :TQ10001  |
|   3 |    LOAD AS SELECT (HYBRID TSM/HWMB)| SALESTEMP |
|   4 |     PX RECEIVE                     |           |
|   5 |      PX SEND ROUND-ROBIN (HUB)     | :TQ10000  |
|   6 |       PX BLOCK ITERATOR            |           |
|   7 |        TABLE ACCESS FULL           | SALES     |
-----------------------------------------------------
```

> **See Also:**
>
> - *Oracle Clusterware Administration and Deployment Guide* for information about nodes in hub, leaf, and flex cluster architecture
> - *Oracle Grid Infrastructure Installation and Upgrade Guide for Linux* for information about cluster installation options for Grid Infrastructure
> - *Oracle Real Application Clusters Administration and Deployment Guide* for more information about instance groups

8.2 Setting the Degree of Parallelism

The **degree of parallelism** (DOP) is the number of parallel execution servers associated with a single operation.

Parallel execution is designed to effectively use multiple CPUs. Oracle Database parallel execution framework enables you to either explicitly choose a specific degree of parallelism or to rely on Oracle Database to automatically control it.

This section contains the following topics:

- Manually Specifying the Degree of Parallelism
- Default Degree of Parallelism

- Automatic Degree of Parallelism
- Determining Degree of Parallelism in Auto DOP
- Controlling Automatic Degree of Parallelism
- Adaptive Parallelism

8.2.1 Manually Specifying the Degree of Parallelism

A specific degree of parallelism (DOP) can be requested from Oracle Database for both tables and indexes.

For example, you can set a fixed DOP at a table level with the following:

```
ALTER TABLE sales PARALLEL 8;
ALTER TABLE customers PARALLEL 4;
```

In this example, queries accessing just the `sales` table request a DOP of 8 and queries accessing the `customers` table request a DOP of 4. A query accessing both the `sales` and the `customers` tables is processed with a DOP of 8 and potentially allocates 16 parallel execution servers (because of the producer/consumer model). Whenever different DOPs are specified, Oracle Database uses the higher DOP.

You can also request a specific DOP by using statement level or object level parallel hints.

The DOP specified in the `PARALLEL` clause of a table or an index takes effect only when `PARALLEL_DEGREE_POLICY` is set to `MANUAL` or `LIMITED`.

The actual runtime DOP of a statement can be limited by Oracle Database Resource Manager.

> **See Also:**
>
> - *Oracle Database SQL Language Reference* for information about hints for parallel processing
> - *Oracle Database Administrator's Guide* for more information about Oracle Database Resource Manager

8.2.2 Default Degree of Parallelism

If the `PARALLEL` clause is specified but no degree of parallelism (DOP) is listed, then the object gets the default DOP.

For example, you can set a table to the default DOP with the following SQL statement.

```
ALTER TABLE sales PARALLEL;
```

Default parallelism uses a formula to determine the DOP based on the system configuration, as in the following:

- For a single instance, DOP = `PARALLEL_THREADS_PER_CPU x CPU_COUNT`
- For an Oracle RAC configuration, DOP = `PARALLEL_THREADS_PER_CPU x sum(CPU_COUNT)`

By default, `sum(CPU_COUNT)` is the total number of CPUs in the cluster. However, if you have used Oracle RAC services to limit the number of nodes across which a parallel operation can execute, then `sum(CPU_COUNT)` is the total number of CPUs across the nodes belonging to that service. For example, on a 4-node Oracle RAC cluster, with each node having 8 CPU cores and no Oracle RAC services, the default DOP would be 2 x (8+8+8+8) = 64.

You can also request the default DOP by using statement level or object level parallel hints.

The default DOP specified in the `PARALLEL` clause of a table or an index takes effect only when `PARALLEL_DEGREE_POLICY` is set to `MANUAL`.

The default DOP algorithm is designed to use maximum resources and assumes that the operation finishes faster if it can use more resources. Default DOP targets the single-user workload and it is not recommended in a multiuser environment.

The actual runtime DOP of a SQL statement can be limited by Oracle Database Resource Manager.

See Also:

- *Oracle Database SQL Language Reference* for information about hints for parallel processing
- *Oracle Database Administrator's Guide* for more information about Oracle Database Resource Manager

8.2.3 Automatic Degree of Parallelism

Automatic Degree of Parallelism (Auto DOP) enables Oracle Database to automatically decide if a statement should execute in parallel and what DOP it should use.

The following is a summary of parallel statement processing when Auto DOP is enabled.

1. A SQL statement is issued.
2. The statement is parsed and the optimizer determines the execution plan.
3. The threshold limit specified by the `PARALLEL_MIN_TIME_THRESHOLD` initialization parameter is checked.
   a. If the expected execution time is less than the threshold limit, the SQL statement is run serially.
   b. If the expected execution time is greater than the threshold limit, the statement is run in parallel based on the DOP that the optimizer calculates, including factoring for any defined resource limitations.

8.2.4 Determining Degree of Parallelism in Auto DOP

With automatic degree of parallelism (DOP), the optimizer automatically determines the DOP for a statement based on the resource requirements of that statement.

The optimizer uses the cost of all scan operations, such as a full table scan or index fast full scan, and the cost of all CPU operations in the execution plan to determine the necessary DOP.

However, the optimizer limits the actual maximum DOP to ensure parallel execution servers do not overwhelm the system. This limit is set by the parameter `PARALLEL_DEGREE_LIMIT`. The default value for this parameter is `CPU`, which means the DOP is limited by the number of CPUs on the system (`PARALLEL_THREADS_PER_CPU` * `sum(CPU_COUNT)`) also known as the default DOP. This default DOP ensures that a single user operation cannot overwhelm the system. By adjusting this parameter setting, you can control the maximum DOP the optimizer can choose for a SQL statement. The optimizer can further limit the maximum DOP that can be chosen if Oracle Database Resource Manager is used to limit the DOP.

> **Note:**
>
> The value `AUTO` for `PARALLEL_DEGREE_LIMIT` has the same functionality as the value `CPU`.

To calculate the cost of operations for a SQL statement, Auto DOP uses information about the hardware characteristics of the system. The hardware characteristics include I/O calibration statistics so these statistics should be gathered.

If I/O calibration is not run to gather the required statistics, a default calibration value is used to calculate the cost of operations and the DOP.

I/O calibration statistics can be gathered with the PL/SQL DBMS_RESOURCE_MANAGER.CALIBRATE_IO procedure. I/O calibration is a one-time action if the physical hardware does not change.

The DOP determined by the optimizer is shown in the notes section of an explain plan output, as shown in the following explain plan output, visible either using the explain plan statement or `V$SQL_PLAN`.

```
EXPLAIN PLAN FOR
SELECT SUM(AMOUNT_SOLD) FROM SH.SALES;

PLAN TABLE OUTPUT
Plan hash value: 1763145153

-----------------------------------------------------------------------------------------------------------------------
|Id| Operation              | Name     | Rows | Bytes| Cost (%CPU)| Time     | Pstart| Pstop|    TQ |IN-OUT| PQ Distrib |
-----------------------------------------------------------------------------------------------------------------------
| 0| SELECT STATEMENT       |          |    1|     4|     2   (0)| 00:00:01 |       |      |       |      |            |
| 1|  SORT AGGREGATE        |          |    1|     4|            |          |       |      |       |      |            |
| 2|   PX COORDINATOR       |          |     |      |            |          |       |      |       |      |            |
| 3|    PX SEND QC (RANDOM) | :TQ10000 |    1|     4|            |          |       |      | Q1,00| P->S | QC (RAND)  |
| 4|     SORT AGGREGATE     |          |    1|     4|            |          |       |      | Q1,00| PCWP |            |
| 5|      PX BLOCK ITERATOR |          |   960|  3840|     2   (0)| 00:00:01 |     1 |    16| Q1,00| PCWC |            |
| 6|       TABLE ACCESS FULL| SALES    |   960|  3840|     2   (0)| 00:00:01 |     1 |    16| Q1,00| PCWP |            |
-----------------------------------------------------------------------------------------------------------------------
Note
-----
   - automatic DOP: Computed Degree of Parallelism is 4
```

> **See Also:**
>
> *Oracle Database PL/SQL Packages and Types Reference* for information about the DBMS_RESOURCE_MANAGER package

8.2.5 Controlling Automatic Degree of Parallelism

There are several initialization parameters that control automatic degree of parallelism (auto DOP). These initialization parameters include `PARALLEL_DEGREE_POLICY`, `PARALLEL_DEGREE_LIMIT`, `PARALLEL_MIN_TIME_THRESHOLD`, and `PARALLEL_MIN_DEGREE`.

The initialization parameter `PARALLEL_DEGREE_POLICY` controls whether Auto DOP, parallel statement queuing, and in-memory parallel execution are enabled. This parameter has the following possible values:

- `MANUAL`

  This setting disables Auto DOP, parallel statement queuing and in-memory parallel execution. It reverts the behavior of parallel execution to what it was previous to Oracle Database 11g Release 2 (11.2), which is the default.

  With the default setting of `MANUAL` for `PARALLEL_DEGREE_POLICY`, the system only uses parallel execution when a DOP has been explicitly set on an object or if a parallel hint is specified in the SQL statement. The DOP used is exactly what was specified. No parallel statement queuing and in-memory parallel execution occurs.

- `LIMITED`

  This setting enables Auto DOP for some statements but parallel statement queuing and in-memory parallel execution are disabled. Automatic DOP is applied only to statements that access tables or indexes declared explicitly with the `PARALLEL` clause without an explicit DOP specified. Tables and indexes that have a DOP specified use that explicit DOP setting.

  If you want Oracle Database to automatically decide the DOP only for a subset of SQL statements that touch a specific subset of objects, then set `PARALLEL_DEGREE_POLICY` to `LIMITED` and set the parallel property without specifying an explicit DOP on that subset of objects.

- `AUTO`

  This setting enables Auto DOP for all statements, also enables parallel statement queuing and in-memory parallel execution.

  If you want Oracle Database to automatically decide the DOP for all SQL statements, then set `PARALLEL_DEGREE_POLICY` to `AUTO`.

- `ADAPTIVE`

  This setting enables Auto DOP, parallel statement queuing, and in-memory parallel execution, similar to the `AUTO` value. In addition, performance feedback is enabled.

The `PARALLEL_DEGREE_LIMIT` initialization parameter specifies the maximum DOP that Auto DOP can use systemwide. For a more fine grained control of the maximum DOP, you can use Oracle Database Resource Manager.

The `PARALLEL_MIN_TIME_THRESHOLD` initialization parameter specifies the minimum estimated execution time for a statement to be considered for Auto DOP. First the optimizer calculates a serial execution plan for the SQL statement. If the estimated execution time is greater than the value of `PARALLEL_MIN_TIME_THRESHOLD`, the statement becomes a candidate for Auto DOP.

The `PARALLEL_MIN_ DEGREE` initialization parameter controls the minimum degree of parallelism computed by automatic degree of parallelism. However, `PARALLEL_MIN_DEGREE` has no impact if the value of `PARALLEL_MIN_DEGREE` is greater than the value of `CPU_COUNT` or if an object is Oracle-owned, such as a dictionary table or view created on a dictionary table.

You can also request Auto DOP by specifying the appropriate statement level or object level SQL hints.

See Also:

- Automatic Degree of Parallelism for additional information about Auto DOP
- Parallel Statement Queuing for information about parallel statement queuing
- In-Memory Parallel Execution for information about in-memory parallel execution
- Tips for Tuning Parallel Execution for information about other techniques that you can use to control parallelism
- *Oracle Database Reference* for information about settings for the `PARALLEL_DEGREE_POLICY` initialization parameter
- *Oracle Database SQL Language Reference* for information about the `PARALLEL` hint

8.2.6 Adaptive Parallelism

The adaptive multiuser algorithm reduces the degree of parallelism as the load on the system increases.

When using Oracle Database adaptive parallelism capabilities, the database uses an algorithm at SQL execution time to determine whether a parallel operation should receive the requested DOP or have its DOP lower to ensure the system is not overloaded.

In a system that makes aggressive use of parallel execution by using a high DOP, the adaptive algorithm adjusts the DOP down when only a few operations are running in parallel. While the algorithm still ensures optimal resource utilization, users may experience inconsistent response times. Using solely the adaptive parallelism capabilities in an environment that requires deterministic response times is not advised. Adaptive parallelism is controlled through the database initialization parameter `PARALLEL_ADAPTIVE_MULTI_USER`.

Note:

Because the initialization parameter `PARALLEL_ADAPTIVE_MULTI_USER` is deprecated in Oracle Database 12c Release 2 (12.2.0.1) and to be desupported in a future release, Oracle recommends using parallel statement queuing instead.

8.3 In-Memory Parallel Execution

In-memory features provide techniques for parallel execution.

This section discusses in-memory parallel execution.

- Buffer Cache Usage in Parallel Execution
- Automatic Big Table Caching

8.3.1 Buffer Cache Usage in Parallel Execution

By default parallel execution does not use the SGA (buffer cache) to cache the scanned blocks unless the object is very small or is declared as CACHE.

In-Memory Parallel Execution, enabled by setting the parameter `PARALLEL_DEGREE_POLICY` is set to `AUTO`, enables parallel statements to leverage the SGA to cache object blocks. Oracle Database decides if an object that is accessed using parallel execution would benefit from being cached in the SGA. The decision to cache an object is based on a well-defined set of heuristics including the size of the object and frequency on which it is accessed. In an Oracle Real Applications Cluster (Oracle RAC) environment, Oracle Database maps pieces of the object into each of the buffer caches on the active instances. By creating this mapping, Oracle Database automatically knows which buffer cache to access to find different parts or pieces of the object. Using this information, Oracle Database prevents multiple instances from reading the same information from disk over and over again, thus maximizing the amount of memory that can cache objects. It does this by using PX servers on the instances where the blocks are cached.

If the size of the object is larger than a specific threshold value based on the total size of the buffer cache (single instance) or the size of the buffer cache multiplied by the number of active instances in an Oracle RAC cluster, then the object is read using direct-path reads and not cached in the SGA.

8.3.2 Automatic Big Table Caching

Automatic big table caching integrates queries with the buffer cache to enhance the in-memory query capabilities of Oracle Database, in both single instance and Oracle RAC environments.

In Oracle Real Application Clusters (Oracle RAC) environments, this feature is supported only with parallel queries. In single instance environments, this feature is supported with both parallel and serial queries.

The cache section reserved for the big table cache is used for caching data for table scans. While the big table cache is primarily designed to enhance performance for data warehouse workloads, it also improves performance in Oracle Database running mixed workloads.

Automatic big table caching uses temperature and object based algorithms to track medium and big tables. Oracle does cache very small tables, but automatic big table caching does not track these tables.

To enable automatic big table caching for serial queries, you must set a value (percentage) for the `DB_BIG_TABLE_CACHE_PERCENT_TARGET` initialization parameter. Additionally, you must set the `PARALLEL_DEGREE_POLICY` initialization parameter to `AUTO` or `ADAPTIVE` to enable parallel queries to use automatic big table caching. In Oracle RAC environments, automatic big table caching is only supported in parallel queries so both settings are required.

If a large table is approximately the size of the combined size of the big table cache of all instances, then the table is partitioned and cached, or mostly cached, on all instances. An in-memory query could eliminate most disk reads for queries on the table, or the database could intelligently read from disk only for that portion of the table that does not fit in the big table cache. If the big table cache cannot cache all the tables to be scanned, only the most frequently accessed table are cached, and the rest are read through direct read automatically.

The `DB_BIG_TABLE_CACHE_PERCENT_TARGET` parameter determines the percentage of the buffer cache size used for scans. If `DB_BIG_TABLE_CACHE_PERCENT_TARGET` is set to `80` (%), then `80`

(%) of the buffer cache is used for scans and the remaining `20` (%) is used for OLTP workloads.

The `DB_BIG_TABLE_CACHE_PERCENT_TARGET` parameter is only enabled in an Oracle RAC environment if `PARALLEL_DEGREE_POLICY` is set to `AUTO` or `ADAPTIVE`. The default for `DB_BIG_TABLE_CACHE_PERCENT_TARGET` is `0` (disabled) and the upper limit is `90` (%) reserving at least 10% buffer cache for usage besides table scans. When the value is `0`, in-memory queries run with the existing least recently used (LRU) mechanism. You can adjust the `DB_BIG_TABLE_CACHE_PERCENT_TARGET` parameter dynamically.

Use the following guidelines when setting the `DB_BIG_TABLE_CACHE_PERCENT_TARGET` parameter:

- If you do not enable automatic degree of parallelism (DOP) in an Oracle RAC environment, then you should not set this parameter because the big table cache section is not used in that situation.
- When setting this parameter, you should consider the workload mix: how much of the workload is for OLTP; insert, update, and random access; and how much of the workload involves table scans. Because data warehouse workloads often perform large table scans, you may consider giving big table cache section a higher percentage of buffer cache space for data warehouses.
- This parameter can be dynamically changed if the workload changes. The change could take some time depending on the current workload to reach the target, because buffer cache memory might be actively used at the time.

When `PARALLEL_DEGREE_POLICY` is set to `AUTO` or `ADAPTIVE`, additional object-level statistics for a data warehouse load and scan buffers are added to represent the number of parallel queries (PQ) scans on the object on the particular (helper) instance.

The `V$BT_SCAN_CACHE` and `V$BT_SCAN_OBJ_TEMPS` views provide information about the big table cache.

See Also:

- *Oracle Database Administrator's Guide* for information about automatic big table caching
- *Oracle Database Concepts* for information about automatic big table caching
- *Oracle Database Reference* for information about the `DB_BIG_TABLE_CACHE_PERCENT_TARGET` initialization parameter
- *Oracle Database Reference* for information about the `V$BT_SCAN*` views

8.4 Parallel Statement Queuing

In some situations, parallel statements are queued while waiting for resources.

When the parameter `PARALLEL_DEGREE_POLICY` is set to `AUTO`, Oracle Database queues SQL statements that require parallel execution if the necessary number of parallel execution server processes are not available. After the necessary resources become available, the SQL statement is dequeued and allowed to execute. The default dequeue order is a simple first in, first out queue based on the time a statement was issued.

The following is a summary of parallel statement processing.

1. A SQL statements is issued.
2. The statement is parsed and the DOP is automatically determined.
3. Available parallel resources are checked.
   a. If there are sufficient parallel execution servers available and there are no statements ahead in the queue waiting for the resources, the SQL statement is executed.
   b. If there are not sufficient parallel execution servers available, the SQL statement is queued based on specified conditions and dequeued from the front of the queue when specified conditions are met.

Parallel statements are queued if running the statements would increase the number of active parallel servers above the value of the `PARALLEL_SERVERS_TARGET` initialization parameter. For example, if `PARALLEL_SERVERS_TARGET` is set to `64`, the number of current active servers is 60, and a new parallel statement needs 16 parallel servers, it would be queued because 16 added to 60 is greater than 64, the value of `PARALLEL_SERVERS_TARGET`.

This value is not the maximum number of parallel server processes allowed on the system, but the number available to run parallel statements before parallel statement queuing is used. It is set lower than the maximum number of parallel server processes allowed on the system (`PARALLEL_MAX_SERVERS`) to ensure each parallel statement gets all of the parallel server resources required and to prevent overloading the system with parallel server processes. Note all serial (nonparallel) statements execute immediately even if parallel statement queuing has been activated.

If a statement has been queued, it is identified by the `resmgr:pq queued` wait event.

This section discusses the following topics:

- About Managing Parallel Statement Queuing with Oracle Database Resource Manager
- Grouping Parallel Statements with BEGIN_SQL_BLOCK END_SQL_BLOCK
- About Managing Parallel Statement Queuing with Hints

> **See Also:**
>
> - V$RSRC_SESSION_INFO and V$RSRCMGRMETRIC for information about views for monitoring and analyzing parallel statement queuing
> - *Oracle Database Reference* for more information about the `PARALLEL_SERVERS_TARGET` initialization parameter

8.4.1 About Managing Parallel Statement Queuing with Oracle Database Resource Manager

By default, the parallel statement queue operates as a first-in, first-out queue, but you can modify the default behavior with a resource plan.

> **Note:**
>
> A multitenant container database is the only supported architecture in Oracle Database 21c. While the documentation is being revised, legacy terminology may persist. In most cases, "database" and "non-CDB" refer to a CDB or PDB, depending on context. In some contexts, such as upgrades, "non-CDB" refers to a non-CDB from a previous release.

By configuring and setting a resource plan, you can control the order in which parallel statements are dequeued and the number of parallel execution servers used by each workload or consumer group.

Oracle Database Resource Manager operates based on the concept of consumer groups and resource plans. Consumer groups identify groups of users with identical resource privileges and requirements. A resource plan consists of a collection of directives for each consumer group which specify controls and allocations for various database resources, such as parallel servers. For multitenant container databases (CDBs) and pluggable databases (PDBs), the order of the parallel statement queue is managed by the directive called `shares`.

A resource plan is enabled by setting the `RESOURCE_MANAGER_PLAN` parameter to the name of the resource plan.

You can use the directives described in the following sections to manage the processing of parallel statements for consumer groups when the parallel degree policy is set to `AUTO`.

- About Managing the Order of the Parallel Statement Queue
- About Limiting the Parallel Server Resources for a Consumer Group
- Specifying a Parallel Statement Queue Timeout for Each Consumer Group
- Specifying a Degree of Parallelism Limit for Consumer Groups
- Critical Parallel Statement Prioritization
- A Sample Scenario for Managing Statements in the Parallel Queue

In all cases, the parallel statement queue of a given consumer group is managed as a single queue on an Oracle RAC database. Limits for each consumer group apply to all sessions across the Oracle RAC database that belong to that consumer group. The queuing of parallel statements occurs based on the sum of the values of the `PARALLEL_SERVERS_TARGET` initialization parameter across all database instances.

You can also manage parallel statement queuing for multitenant container databases (CDBs) and pluggable databases (PDBs).

See Also:

- *Oracle Database Administrator's Guide* for information about managing Oracle Database resources with Oracle Database Resource Manager
- *Oracle Multitenant Administrator's Guide* for information about parallel execution (PX) servers and utilization limits for for multitenant container databases (CDBs) and pluggable databases (PDBs)
- *Oracle Database Reference* for information about `V$RSRC*` views, the `DBA_HIST_RSRC_CONSUMER_GROUP` view, and parallel query wait events
- *Oracle Database PL/SQL Packages and Types Reference* for information about the `DBMS_RESOURCE_MANAGER` package

8.4.1.1 About Managing the Order of the Parallel Statement Queue

You can use Oracle Database Resource Manager to manage the priority for dequeuing parallel statements from the parallel statement queue across multiple consumer groups.

The parallel statements for a particular consumer group are dequeued in FIFO order by default. With the directives `shares`, you can determine the order in which the parallel statements of a consumer group are dequeued. You configure these directives with the `CREATE_PLAN_DIRECTIVE` or `UPDATE_PLAN_DIRECTIVE` procedure of the `DBMS_RESOURCE_MANAGER` PL/SQL package. You can also set `shares` in a CDB resource plan to manage the order of parallel statements among PDBs.

For example, you can create the `PQ_HIGH`, `PQ_MEDIUM`, and `PQ_LOW` consumer groups and map parallel statement sessions to these consumer groups based on priority. You then create a resource plan that sets `shares=14` for `PQ_HIGH`, `shares=5` for `PQ_MEDIUM`, and `shares=1` for `PQ_LOW`. This indicates that `PQ_HIGH` statements are dequeued with a probability of `70%` (14/(14+5+1)=.70) of the time, `PQ_MEDIUM` dequeued with a probability of `25%` (5/(14+5+1)=.25) of the time, and `PQ_LOW` dequeued with a probability of `5%` (1/(14+5+1)=.05) of the time.

If a parallel statement has been queued and you determine that the parallel statement must be run immediately, then you can run the `DBMS_RESOURCE_MANAGER.DEQUEUE_PARALLEL_STATEMENT` PL/SQL procedure to dequeue the parallel statement.

See Also:

- *Oracle Database PL/SQL Packages and Types Reference* for information about procedures in the `DBMS_RESOURCE_MANAGER` package
- *Oracle Database Administrator's Guide* for information about creating resource plan directives

8.4.1.2 About Limiting the Parallel Server Resources for a Consumer Group

You can use Oracle Database Resource Manager to limit the number of parallel servers that parallel statements from lower priority consumer groups can use for parallel statement processing.

Using Oracle Database Resource Manager you can map parallel statement sessions to different consumer groups that each have specific limits on the number of the parallel servers that can be used. Every consumer group has its own individual parallel statement queue. When these limits for consumer groups are specified, parallel statements from a consumer group are queued when its limit would be exceeded.

This limitation becomes useful when a database has high priority and low priority consumer groups. Without limits, a user may issue a large number of parallel statements from a low-priority consumer group that uses all parallel servers. When a parallel statement from a high priority consumer group is issued, the resource allocation directives can ensure that the high priority parallel statement is dequeued first. By limiting the number of parallel servers a low-priority consumer group can use, you can ensure that there are always some parallel servers available for a high priority consumer group.

To limit the parallel servers used by a consumer group, use the `parallel_server_limit` parameter with the `CREATE_PLAN_DIRECTIVE` procedure or the `new_parallel_server_limit` parameter with the `UPDATE_PLAN_DIRECTIVE` procedure in the `DBMS_RESOURCE_MANAGER` package. The `parallel_server_limit` parameter specifies the maximum percentage of the Oracle RAC-wide parallel server pool that is specified by `PARALLEL_SERVERS_TARGET` that a consumer group can use.

For multitenant container database (CDB) resource plans, the parallel server limit applies to pluggable databases (PDBs). For PDB resource plans or non-CDB resource plans, this limit applies to consumer groups.

For example, on an Oracle RAC database in nonmultitenant configuration, the initialization parameter `PARALLEL_SERVERS_TARGET` is set to `32` on two nodes so there are a total of 32 x 2 = 64 parallel servers that can be used before queuing begins. You can set up the consumer group `PQ_LOW` to use 50% of the available parallel servers (`parallel_server_limit` = 50) and low priority statements can then be mapped to the `PQ_LOW` consumer group. This scenario limits any parallel statements from the `PQ_LOW` consumer group to 64 x 50% = 32 parallel servers, even though there are more inactive or unused parallel servers. In this scenario, after the statements from the `PQ_LOW` consumer group have used 32 parallel servers, statements from that consumer group are queued.

It is possible in one database to have some sessions with the parallelism degree policy set to `MANUAL` and some sessions set to `AUTO`. In this scenario, only the sessions with parallelism degree policy set to `AUTO` can be queued. However, the parallel servers used in sessions where the parallelism degree policy is set to `MANUAL` are included in the total of all parallel servers used by a consumer group.

> **See Also:**
>
> PARALLEL_SERVERS_TARGET for information about limiting parallel resources for users

8.4.1.3 Specifying a Parallel Statement Queue Timeout for Each Consumer Group

You can use Oracle Database Resource Manager to set specific maximum queue timeouts for consumer groups so that parallel statements do not stay in the queue for long periods of time.

To manage the queue timeout, the `parallel_queue_timeout` parameter is used with the `CREATE_PLAN_DIRECTIVE` procedure or the `new_parallel_queue_timeout` parameter is used with the `UPDATE_PLAN_DIRECTIVE` procedure in the `DBMS_RESOURCE_MANAGER` package. The

parallel_queue_timeout and new_parallel_queue_timeout parameters specify the time in seconds that a statement can remain in a consumer group parallel statement queue. After the timeout period expires, the statement is either terminated with error ORA-7454 or removed from the parallel statement queue and enabled to run based on the value for the PQ_TIMEOUT_ACTION directive in the resource manager plan.

You can specify queue timeout actions for parallel statements using the PQ_TIMEOUT_ACTION resource manager directive. Setting this directive to CANCEL terminates the statement with the error ORA-7454. Setting this directive to RUN enables the statement to run.

8.4.1.4 Specifying a Degree of Parallelism Limit for Consumer Groups

You can use Oracle Database Resource Manager to the limit the degree of parallelism for specific consumer groups.

Using Oracle Database Resource Manager you can map parallel statement sessions to different consumer groups that each have specific limits for the degree of parallelism in a resource plan.

To manage the limit of parallelism in consumer groups, use the parallel_degree_limit_p1 parameter with the CREATE_PLAN_DIRECTIVE procedure in the DBMS_RESOURCE_MANAGER package or the new_parallel_degree_limit_p1 parameter with the UPDATE_PLAN_DIRECTIVE procedure in the DBMS_RESOURCE_MANAGER package. The parallel_degree_limit_p1 and new_parallel_degree_limit_p1 parameters specify a limit on the degree of parallelism for any operation.

For example, you can create the PQ_HIGH, PQ_MEDIUM, and PQ_LOW consumer groups and map parallel statement sessions to these consumer groups based on priority. You then create a resource plan that specifies degree of parallelism limits so that the PQ_HIGH limit is set to 16, the PQ_MEDIUM limit is set to 8, and the PQ_LOW limit is set to 2.

The degree of parallelism limit is enforced, even if PARALLEL_DEGREE_POLICY is not set to AUTO.

8.4.1.5 Critical Parallel Statement Prioritization

The setting of the PARALLEL_STMT_CRITICAL parameter affects the critical designation of parallel statements in the plan directive with respect to the parallel statement queue.

- If the PARALLEL_STMT_CRITICAL parameter is set to QUEUE, then parallel statements that have PARALLEL_DEGREE_POLICY set to MANUAL are queued.
- If the PARALLEL_STMT_CRITICAL parameter is set to BYPASS_QUEUE, then parallel statements bypass the parallel statement queue and execute immediately.
- If PARALLEL_STMT_CRITICAL is set to FALSE, then that specifies the default behavior and no statement is designated as critical.

Because critical parallel statements bypass the parallel statement queue, the system may encounter more active parallel servers than specified by the PARALLEL_SERVERS_TARGET parameter. Critical parallel statements are allowed to run after the number of parallel servers reaches PARALLEL_MAX_SERVERS, so some critical parallel statements may be downgraded.

The PARALLEL_STMT_CRITICAL column in the DBA_RSRC_PLAN_DIRECTIVES view indicates whether parallel statements are from a consumer group that has been marked critical.

8.4.1.6 A Sample Scenario for Managing Statements in the Parallel Queue

This scenario discusses how to manage statements in the parallel queue with consumer groups set up with Oracle Database Resource Manager.

For this scenario, consider a data warehouse workload that consists of three types of SQL statements:

- Short-running SQL statements

  Short-running identifies statements running less than one minute. You expect these statements to have very good response times.

- Medium-running SQL statements

  Medium-running identifies statements running more than one minute, but less than 15 minutes. You expect these statements to have reasonably good response times.

- Long-running SQL statements

  Long-running identifies statements that are ad-hoc or complex queries running more than 15 minutes. You expect these statements to take a long time.

For this data warehouse workload, you want better response times for the short-running statements. To achieve this goal, you must ensure that:

- Long-running statements do not use all of the parallel server resources, forcing shorter statements to wait in the parallel statement queue.
- When both short-running and long-running statements are queued, short-running statements should be dequeued ahead of long-running statements.
- The DOP for short-running queries is limited because the speedup from a very high DOP is not significant enough to justify the use of a large number of parallel servers.

Example 8-3 shows how to set up consumer groups using Oracle Database Resource Manager to set priorities for statements in the parallel statement queue. Note the following for this example:

- By default, users are assigned to the `OTHER_GROUPS` consumer group. If the estimated execution time of a SQL statement is longer than 1 minute (60 seconds), then the user switches to `MEDIUM_SQL_GROUP`. Because `switch_for_call` is set to `TRUE`, the user returns to `OTHER_GROUPS` when the statement has completed. If the user is in `MEDIUM_SQL_GROUP` and the estimated execution time of the statement is longer than 15 minutes (900 seconds), the user switches to `LONG_SQL_GROUP`. Similarly, because `switch_for_call` is set to `TRUE`, the user returns to `OTHER_GROUPS` when the query has completed. The directives used to accomplish the switch process are `switch_time`, `switch_estimate`, `switch_for_call`, and `switch_group`.
- After the number of active parallel servers reaches the value of the `PARALLEL_SERVERS_TARGET` initialization parameter, subsequent parallel statements are queued. The `shares` directives control the order in which parallel statements are dequeued when parallel servers become available. Because `shares` is set to `100%` for `SYS_GROUP` in this example, parallel statements from `SYS_GROUP` are always dequeued first. If no parallel statements from `SYS_GROUP` are queued, then parallel statements from `OTHER_GROUPS` are dequeued with probability 70%, from `MEDIUM_SQL_GROUP` with probability 20%, and `LONG_SQL_GROUP` with probability 10%.
- Parallel statements issued from `OTHER_GROUPS` are limited to a DOP of 4 with the setting of the `parallel_degree_limit_p1` directive.

- To prevent parallel statements of the `LONG_SQL_GROUP` group from using all of the parallel servers, which could potentially cause parallel statements from `OTHER_GROUPS` or `MEDIUM_SQL_GROUP` to wait for long periods of time, its `parallel_server_limit` directive is set to `50%`. This setting means that after `LONG_SQL_GROUP` has used up 50% of the parallel servers set with the `PARALLEL_SERVERS_TARGET` initialization parameter, its parallel statements are forced to wait in the queue.
- Because parallel statements of the `LONG_SQL_GROUP` group may be queued for a significant amount of time, a timeout is configured for 14400 seconds (4 hours). When a parallel statement from `LONG_SQL_GROUP` has waited in the queue for 4 hours, the statement is terminated with the error ORA-7454.

Example 8-3 Using consumer groups to set priorities in the parallel statement queue

```
BEGIN
  DBMS_RESOURCE_MANAGER.CREATE_PENDING_AREA();

  /* Create consumer groups.
   * By default, users start in OTHER_GROUPS, which is automatically
   * created for every database.
   */
  DBMS_RESOURCE_MANAGER.CREATE_CONSUMER_GROUP(
    'MEDIUM_SQL_GROUP',
    'Medium-running SQL statements, between 1 and 15 minutes.  Medium priority.');

  DBMS_RESOURCE_MANAGER.CREATE_CONSUMER_GROUP(
    'LONG_SQL_GROUP',
    'Long-running SQL statements of over 15 minutes.  Low priority.');

  /* Create a plan to manage these consumer groups */
  DBMS_RESOURCE_MANAGER.CREATE_PLAN(
    'REPORTS_PLAN',
    'Plan for daytime that prioritizes short-running queries');

  DBMS_RESOURCE_MANAGER.CREATE_PLAN_DIRECTIVE(
    'REPORTS_PLAN', 'SYS_GROUP', 'Directive for sys activity',
    shares => 100);

  DBMS_RESOURCE_MANAGER.CREATE_PLAN_DIRECTIVE(
    'REPORTS_PLAN', 'OTHER_GROUPS', 'Directive for short-running queries',
    shares => 70,
    parallel_degree_limit_p1 => 4,
    switch_time => 60, switch_estimate => TRUE, switch_for_call => TRUE,
    switch_group => 'MEDIUM_SQL_GROUP');

  DBMS_RESOURCE_MANAGER.CREATE_PLAN_DIRECTIVE(
    'REPORTS_PLAN', 'MEDIUM_SQL_GROUP', 'Directive for medium-running queries',
    shares => 20,
    parallel_server_limit => 80,
    switch_time => 900, switch_estimate => TRUE, switch_for_call => TRUE,
    switch_group => 'LONG_SQL_GROUP');

  DBMS_RESOURCE_MANAGER.CREATE_PLAN_DIRECTIVE(
    'REPORTS_PLAN', 'LONG_SQL_GROUP', 'Directive for medium-running queries',
    shares => 10,
    parallel_server_limit => 50,
    parallel_queue_timeout => 14400);

  DBMS_RESOURCE_MANAGER.SUBMIT_PENDING_AREA();
END;
/
```

```
/* Allow all users to run in these consumer groups */
EXEC DBMS_RESOURCE_MANAGER_PRIVS.GRANT_SWITCH_CONSUMER_GROUP(
  'public', 'MEDIUM_SQL_GROUP', FALSE);

EXEC DBMS_RESOURCE_MANAGER_PRIVS.GRANT_SWITCH_CONSUMER_GROUP(
  'public', 'LONG_SQL_GROUP', FALSE);
```

8.4.2 Grouping Parallel Statements with BEGIN_SQL_BLOCK END_SQL_BLOCK

Often it is important for a report or batch job that consists of multiple parallel statements to complete as quickly as possible.

For example, when many reports are launched simultaneously, you may want all of the reports to complete as quickly as possible. However, you also want some specific reports to complete first, rather than all reports finishing at the same time.

If a report contains multiple parallel statements and `PARALLEL_DEGREE_POLICY` is set to `AUTO`, then each parallel statement may be forced to wait in the queue on a busy database. For example, the following steps describe a scenario in SQL statement processing:

```
serial statement
parallel query - dop 8
  -> wait in queue
serial statement
parallel query - dop 32
  -> wait in queue
parallel query - dop 4
  -> wait in queue
```

For a report to be completed quickly, the parallel statements must be grouped to produce the following behavior:

```
start SQL block
serial statement
parallel query - dop 8
  -> first parallel query: ok to wait in queue
serial statement
parallel query - dop 32
  -> avoid or minimize wait
parallel query - dop 4
  -> avoid or minimize wait
end SQL block
```

To group the parallel statements, you can use the `BEGIN_SQL_BLOCK` and `END_SQL_BLOCK` procedures in the `DBMS_RESOURCE_MANAGER` PL/SQL package. For each consumer group, the parallel statement queue is ordered by the time associated with each of the consumer group's parallel statements. Typically, the time associated with a parallel statement is the time that the statement was enqueued, which means that the queue appears to be FIFO. When parallel statements are grouped in a SQL block with the `BEGIN_SQL_BLOCK` and `END_SQL_BLOCK` procedures, the first queued parallel statement also uses the time that it was enqueued. However, the second and all subsequent parallel statements receive special treatment and are enqueued using the enqueue time of the first queued parallel statement within the SQL block. With this functionality, the statements frequently move to the front of the parallel statement queue. This preferential treatment ensures that their wait time is minimized.

> **See Also:**
>
> *Oracle Database PL/SQL Packages and Types Reference* for information about the `DBMS_RESOURCE_MANAGER` package

8.4.3 About Managing Parallel Statement Queuing with Hints

You can use the `NO_STATEMENT_QUEUING` and `STATEMENT_QUEUING` hints in SQL statements to influence whether or not a statement is queued with parallel statement queuing.

- `NO_STATEMENT_QUEUING`

  When `PARALLEL_DEGREE_POLICY` is set to `AUTO`, this hint enables a statement to bypass the parallel statement queue. However, a statement that bypasses the statement queue can potentially cause the system to exceed the maximum number of parallel execution servers defined by the value of the `PARALLEL_SERVERS_TARGET` initialization parameter, which determines the limit at which parallel statement queuing is initiated.

  There is no guarantee that the statement that bypasses the parallel statement queue receives the number of parallel execution servers requested because only the number of parallel execution servers available on the system, up to the value of the `PARALLEL_MAX_SERVERS` initialization parameter, can be allocated.

  For example:

```
SELECT /*+ NO_STATEMENT_QUEUING */ last_name, department_name
  FROM employees e, departments d
  WHERE e.department_id = d.department_id;
```

- `STATEMENT_QUEUING`

  When `PARALLEL_DEGREE_POLICY` is not set to `AUTO`, this hint enables a statement to be considered for parallel statement queuing, but to run only when enough parallel processes are available to run at the requested DOP. The number of available parallel execution servers, before queuing is enabled, is equal to the difference between the number of parallel execution servers in use and the maximum number allowed in the system, which is defined by the `PARALLEL_SERVERS_TARGET` initialization parameter.

  For example:

```
SELECT /*+ STATEMENT_QUEUING */ last_name, department_name
  FROM employees e, departments d
  WHERE e.department_id = d.department_id;
```

8.5 Types of Parallelism

There are multiple types of parallelism.

This section discusses the types of parallelism in the following topics:

- About Parallel Queries
- About Parallel DDL Statements
- About Parallel DML Operations
- About Parallel Execution of Functions
- About Other Types of Parallelism

- Degree of Parallelism Rules for SQL Statements

8.5.1 About Parallel Queries

You can use parallel queries and parallel subqueries in `SELECT` statements and execute in parallel the query portions of DDL statements and DML statements (`INSERT`, `UPDATE`, and `DELETE`).

You can also query external tables in parallel.

The parallelization decision for SQL queries has two components: the decision to parallelize and the degree of parallelism (DOP). These components are determined differently for queries, DDL operations, and DML operations. To determine the DOP, Oracle Database looks at the reference objects:

- Parallel query looks at each table and index, in the portion of the query to be executed in parallel, to determine which is the reference table. The basic rule is to pick the table or index with the largest DOP.
- For parallel DML (`INSERT`, `UPDATE`, `MERGE`, and `DELETE`), the reference object that determines the DOP is the table being modified by an insert, update, or delete operation. Parallel DML also adds some limits to the DOP to prevent deadlock. If the parallel DML statement includes a subquery, the subquery's DOP is equivalent to that for the DML operation.
- For parallel DDL, the reference object that determines the DOP is the table, index, or partition being created, rebuilt, split, or moved. If the parallel DDL statement includes a subquery, the subquery's DOP is equivalent to the DDL operation.

This section contains the following topics:

- Parallel Queries on Index-Organized Tables
- Parallel Queries on Object Types
- Rules for Parallelizing Queries

> **See Also:**
>
> - Parallel Execution of SQL Statements for an explanation of how the processes perform parallel queries
> - Distributed Transaction Restrictions for examples of queries that reference a remote object
> - Rules for Parallelizing Queries for information about the conditions for executing a query in parallel and the factors that determine the DOP

8.5.1.1 Parallel Queries on Index-Organized Tables

There are several parallel scan methods that are supported on index-organized tables.

These parallel scan methods include:

- Parallel fast full scan of a nonpartitioned index-organized table
- Parallel fast full scan of a partitioned index-organized table
- Parallel index range scan of a partitioned index-organized table

You can use these scan methods for index-organized tables with overflow areas and for index-organized tables that contain LOBs.

Nonpartitioned Index-Organized Tables

Parallel query on a nonpartitioned index-organized table uses parallel fast full scan. Work is allocated by dividing the index segment into a sufficiently large number of block ranges and then assigning the block ranges to parallel execution servers in a demand-driven manner. The overflow blocks corresponding to any row are accessed in a demand-driven manner only by the process, which owns that row.

Partitioned Index-Organized Tables

Both index range scan and fast full scan can be performed in parallel. For parallel fast full scan, parallelization is the same as for nonpartitioned index-organized tables. Depending on the DOP, each parallel execution server gets one or more partitions, each of which contains the primary key index segment and the associated overflow segment, if any.

8.5.1.2 Parallel Queries on Object Types

Parallel queries can be performed on object type tables and tables containing object type columns.

Parallel query for object types supports all of the features that are available for sequential queries on object types, including:

- Methods on object types
- Attribute access of object types
- Constructors to create object type instances
- Object views
- PL/SQL and Oracle Call Interface (OCI) queries for object types

There are no limitations on the size of the object types for parallel queries.

The following restrictions apply to using parallel query for object types:

- A `MAP` function is needed to execute queries in parallel for queries involving joins and sorts (through `ORDER BY`, `GROUP BY`, or set operations). Without a `MAP` function, the query is automatically executed serially.
- Parallel DML and parallel DDL are not supported with object types, and such statements are always performed serially.

In all cases where the query cannot execute in parallel because of any of these restrictions, the whole query executes serially without giving an error message.

8.5.1.3 Rules for Parallelizing Queries

A SQL query can only be executed in parallel under certain conditions.

A `SELECT` statement can be executed in parallel only if one of the following conditions is satisfied:

- The query includes a statement level or object level parallel hint specification (`PARALLEL` or `PARALLEL_INDEX`).
- The schema objects referred to in the query have a `PARALLEL` declaration associated with them.

- Automatic Degree of Parallelism (Auto DOP) has been enabled.
- Parallel query is forced using the `ALTER SESSION FORCE PARALLEL QUERY` statement.

In addition, the execution plan should have at least one of the following:

- A full table scan
- An index range scan spanning multiple partitions
- An index fast full scan
- A parallel table function

> **See Also:**
>
> - Automatic Degree of Parallelismfor information about Auto DOP
> - Degree of Parallelism Rules for SQL Statements for information about the rules for determining the degree of parallelism (DOP)
> - *Oracle Database SQL Language Reference* for more information about the `ALTER SESSION` SQL statement

8.5.2 About Parallel DDL Statements

Parallelism for DDL statements is introduced in this topic.

This section about parallelism for DDL statements contains the following topics:

- DDL Statements That Can Be Parallelized
- About Using CREATE TABLE AS SELECT in Parallel
- Recoverability and Parallel DDL
- Space Management for Parallel DDL
- Storage Space When Using Dictionary-Managed Tablespaces
- Free Space and Parallel DDL
- Rules for DDL Statements
- Rules for CREATE TABLE AS SELECT

8.5.2.1 DDL Statements That Can Be Parallelized

You can execute DDL statements in parallel for tables and indexes that are nonpartitioned or partitioned.

The parallel DDL statements for nonpartitioned tables and indexes are:

- `CREATE INDEX`
- `CREATE TABLE AS SELECT`
- `ALTER TABLE MOVE`
- `ALTER INDEX REBUILD`
- `ALTER TABLE MODIFY`

The parallel DDL statements for partitioned tables and indexes are:

- `CREATE INDEX`
- `CREATE TABLE AS SELECT`
- `ALTER TABLE {MOVE|SPLIT|COALESCE} PARTITION`
- `ALTER INDEX {REBUILD|SPLIT} PARTITION`
  - This statement can be executed in parallel only if the (global) index partition being split is usable.

All of these DDL operations can be performed in `NOLOGGING` mode for either parallel or serial execution.

`The CREATE TABLE` statement for an index-organized table can be executed in parallel either with or without an `AS SELECT` clause.

Parallel DDL cannot occur on tables with object columns. Parallel DDL cannot occur on nonpartitioned tables with `LOB` columns.

8.5.2.2 About Using CREATE TABLE AS SELECT in Parallel

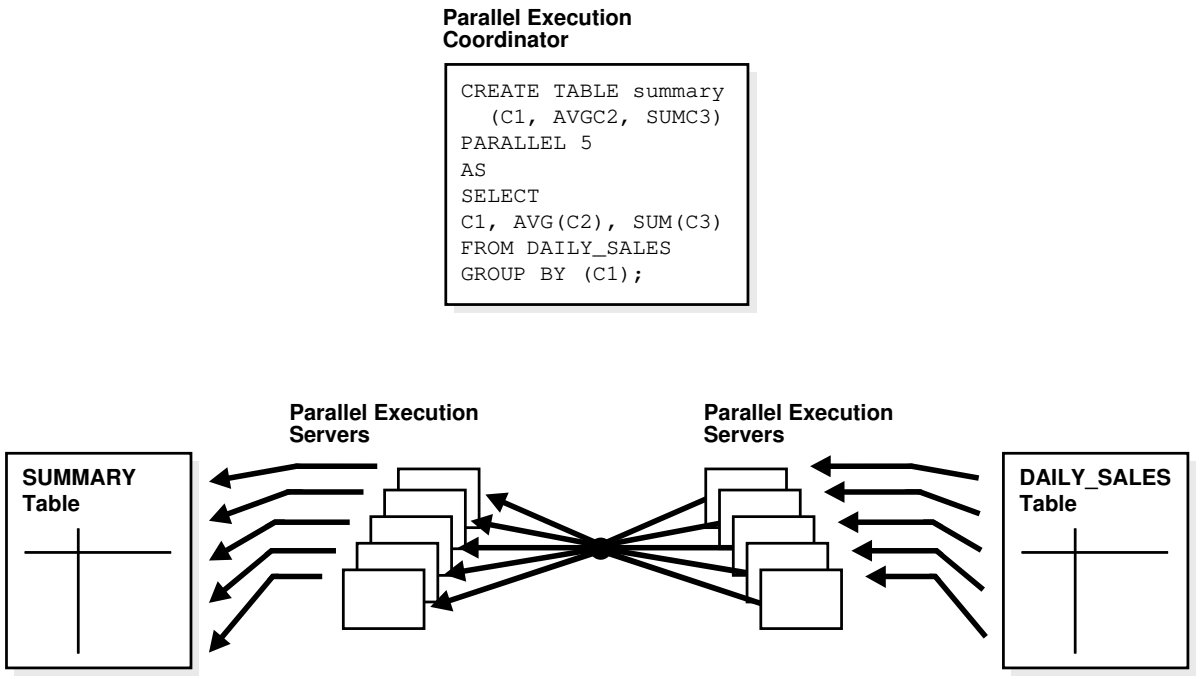
Parallel execution enables you execute the query in parallel and create operations of creating a table as a subquery from another table or set of tables.

This parallel functionality can be extremely useful in the creation of summary or rollup tables.

Note that clustered tables cannot be created and populated in parallel.

Figure 8-4 illustrates creating a summary table from a subquery in parallel.

Figure 8-4 Creating a Summary Table in Parallel



8.5.2.3 Recoverability and Parallel DDL

Parallel DDL is often used to create summary tables or do massive data loads that are standalone transactions, which do not always need to be recoverable.

By switching off Oracle Database logging, no undo or redo log is generated, so the parallel DML operation is likely to perform better, but becomes an *all or nothing* operation. In other words, if the operation fails, for whatever reason, you must redo the operation, it is not possible to restart it.

If you disable logging during parallel table creation (or any other parallel DDL operation), you should back up the tablespace containing the table after the table is created to avoid loss of the table due to media failure.

Use the `NOLOGGING` clause of the `CREATE TABLE`, `CREATE INDEX`, `ALTER TABLE`, and `ALTER INDEX` statements to disable undo and redo log generation.

8.5.2.4 Space Management for Parallel DDL

Creating a table or index in parallel has space management implications.

These space management implications affect both the storage space required during a parallel operation and the free space available after a table or index has been created.

8.5.2.5 Storage Space When Using Dictionary-Managed Tablespaces

When creating a table or index in parallel, each parallel execution server uses the values in the `STORAGE` clause of the `CREATE` statement to create temporary segments to store the rows.

A table created with a `NEXT` setting of 4 MB and a `PARALLEL DEGREE` of 16 consumes at least 64 megabytes (MB) of storage during table creation because each parallel server process starts with an extent of 4 MB. When the parallel execution coordinator combines the segments, some segments may be trimmed, and the resulting table may be smaller than the requested 64 MB.

8.5.2.6 Free Space and Parallel DDL

When you create indexes and tables in parallel, each parallel execution server allocates a new extent and fills the extent with the table or index data.

For example, if you create an index with a DOP of 4, then the index has at least four extents initially. This allocation of extents is the same for rebuilding indexes in parallel and for moving, splitting, or rebuilding partitions in parallel.

Serial operations require the schema object to have at least one extent. Parallel creations require that tables or indexes have at least as many extents as there are parallel execution servers creating the schema object.

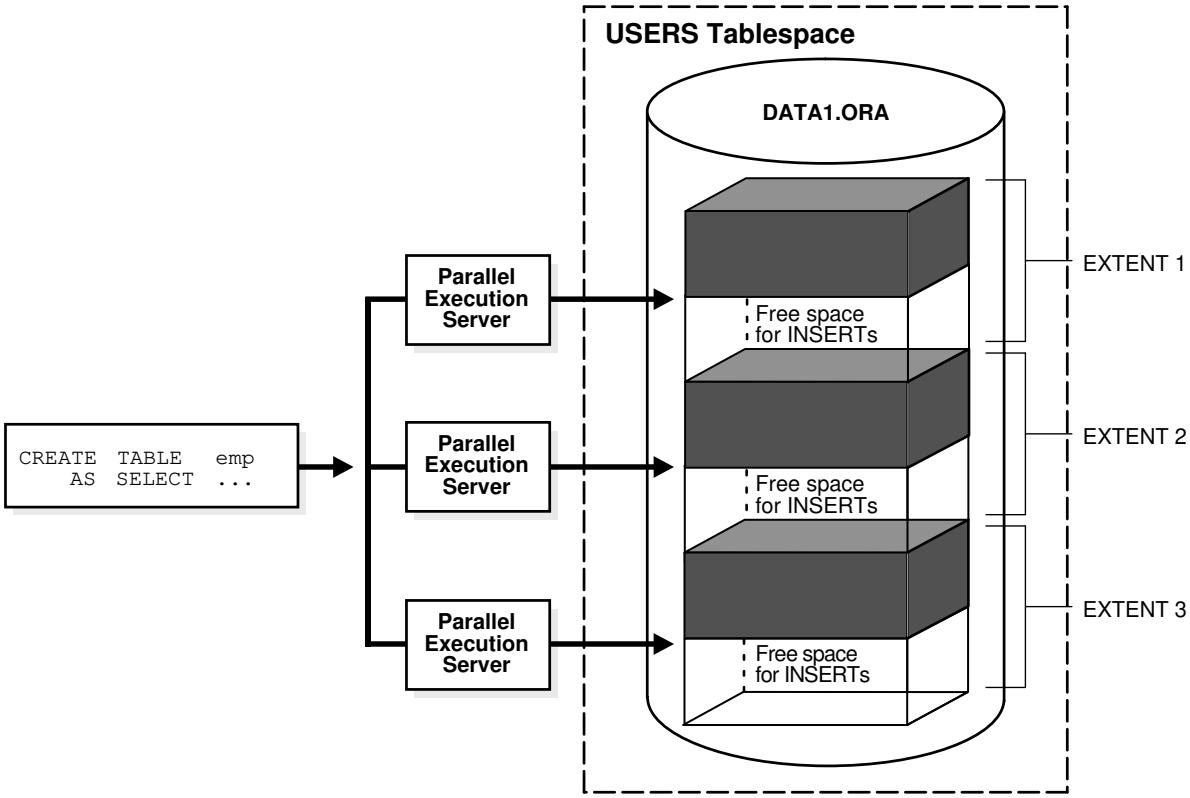
When you create a table or index in parallel, it is possible to create areas of free space. This occurs when the temporary segments used by the parallel execution servers are larger than what is needed to store the rows.

- If the unused space in each temporary segment is larger than the value of the `MINIMUM EXTENT` parameter set at the tablespace level, then Oracle Database trims the unused space when merging rows from all of the temporary segments into the table or index. The unused space is returned to the system free space and can be allocated for new extents, but it cannot be coalesced into a larger segment because it is not contiguous space (external fragmentation).

- If the unused space in each temporary segment is smaller than the value of the `MINIMUM EXTENT` parameter, then unused space cannot be trimmed when the rows in the temporary segments are merged. This unused space is not returned to the system free space; it becomes part of the table or index (internal fragmentation) and is available only for subsequent insertions or for updates that require additional space.

For example, if you specify a DOP of 3 for a `CREATE TABLE AS SELECT` statement, but there is only one data file in the tablespace, then internal fragmentation may occur, as shown in Figure 8-5. The areas of free space within the internal table extents of a data file cannot be coalesced with other free space and cannot be allocated as extents.

Figure 8-5 Unusable Free Space (Internal Fragmentation)



> **See Also:**
>
> *Oracle Database SQL Tuning Guide* for more information about creating tables and indexes in parallel

8.5.2.7 Rules for DDL Statements

DDL operations can be executed in parallel under certain conditions.

DDL operations can be executed in parallel only if at least one of the following conditions is satisfied:

- A `PARALLEL` clause (declaration) is specified in the syntax. For `CREATE TABLE`, `CREATE INDEX` , `ALTER INDEX REBUILD`, and `ALTER INDEX REBUILD PARTITION`, the parallel declaration is stored in the data dictionary.
- Automatic Degree of Parallelism (Auto DOP) has been enabled.
- Parallel DDL is forced using the `ALTER SESSION FORCE PARALLEL DDL` statement.

See Also:

- Automatic Degree of Parallelism for information about Auto DOP
- Degree of Parallelism Rules for SQL Statements for information about the rules for determining the degree of parallelism (DOP)
- *Oracle Database SQL Language Reference* for more information about the `ALTER SESSION` SQL statement

8.5.2.8 Rules for CREATE TABLE AS SELECT

The `CREATE` operation of the `CREATE TABLE AS SELECT` statement is parallelized based on the rules for parallelizing DDL statements.

In addition, a statement level `PARALLEL` hint specified in the `SELECT` part of the statement can also parallelize the DDL operation. For information about rules for parallelizing DDL statements, refer to Rules for DDL Statements.

When the `CREATE` operation of `CREATE TABLE AS SELECT` is parallelized, Oracle Database also parallelizes the scan operation if possible.

Even if the DDL part is not parallelized, the `SELECT` part can be parallelized based on the rules for parallelizing queries.

Automatic Degree of Parallelism (Auto DOP) parallelizes both the DDL and the query parts of the statement.

For information about the rules for determining the degree of parallelism (DOP), refer to Degree of Parallelism Rules for SQL Statements.

8.5.3 About Parallel DML Operations

Parallel DML operations are introduced in the topic.

Parallel DML (`PARALLEL INSERT`, `UPDATE`, `DELETE`, and `MERGE`) uses parallel execution mechanisms to speed up or scale up large DML operations against large database tables and indexes.

> **Note:**
>
> Multiple DML/parallel DML operations can modify the same object in the same session. There is no longer a requirement to commit each operation separately. This reduces the commit overhead for parallel DML.
>
> Although DML generally includes queries, in this chapter the term DML refers only to `INSERT`, `UPDATE`, `MERGE`, and `DELETE` operations.

This section discusses the following parallel DML topics:

- When to Use Parallel DML
- Enable Parallel DML Mode
- Rules for UPDATE, MERGE, and DELETE
- Rules for INSERT SELECT
- Transaction Restrictions for Parallel DML
- Rollback Segments
- Recovery for Parallel DML
- Space Considerations for Parallel DML
- Restrictions on Parallel DML
- Data Integrity Restrictions
- Trigger Restrictions
- Distributed Transaction Restrictions
- Examples of Distributed Transaction Parallelization
- Concurrent Execution of Union All

8.5.3.1 When to Use Parallel DML

Parallel DML is useful in a decision support system (DSS) environment where the performance and scalability of accessing large objects are important. Parallel DML complements parallel query in providing you with both querying and updating capabilities for your DSS databases.

The overhead of setting up parallelism makes parallel DML operations not feasible for short OLTP transactions. However, parallel DML operations can speed up batch jobs running in an OLTP database.

Several scenarios where parallel DML is used include:

- Refreshing Tables in a Data Warehouse System
- Creating Intermediate Summary Tables
- Using Scoring Tables
- Updating Historical Tables
- Running Batch Jobs

8.5.3.1.1 Refreshing Tables in a Data Warehouse System

In a data warehouse system, large tables must be refreshed (updated) periodically with new or modified data from the production system.

You can do this efficiently by using the `MERGE` statement.

8.5.3.1.2 Creating Intermediate Summary Tables

In a DSS environment, many applications require complex computations that involve constructing and manipulating many large intermediate summary tables.

These summary tables are often temporary and frequently do not need to be logged. Parallel DML can speed up the operations against these large intermediate tables. One benefit is that you can put incremental results in the intermediate tables and perform parallel updates.

In addition, the summary tables may contain cumulative or comparative information which has to persist beyond application sessions; thus, temporary tables are not feasible. Parallel DML operations can speed up the changes to these large summary tables.

8.5.3.1.3 Using Scoring Tables

Many DSS applications score customers periodically based on a set of criteria.

The scores are usually stored in large DSS tables. The score information is then used in making a decision, for example, inclusion in a mailing list.

This scoring activity queries and updates a large number of rows in the table. Parallel DML can speed up the operations against these large tables.

8.5.3.1.4 Updating Historical Tables

Historical tables describe the business transactions of an enterprise over a recent time interval.

Periodically, the DBA deletes the set of oldest rows and inserts a set of new rows into the table. Parallel `INSERT SELECT` and parallel `DELETE` operations can speed up this rollover task.

Dropping a partition can also be used to delete old rows. However, the table has to be partitioned by date and with the appropriate time interval.

8.5.3.1.5 Running Batch Jobs

Batch jobs executed in an OLTP database during off hours have a fixed time during which the jobs must complete. A good way to ensure timely job completion is to execute their operations in parallel.

As the workload increases, more computer resources can be added; the scaleup property of parallel operations ensures that the time constraint can be met.

8.5.3.2 Enable Parallel DML Mode

A DML statement can be parallelized only if you have explicitly enabled parallel DML in the session or in the SQL statement.

To enable this mode in a session, run the following SQL statement:

```
ALTER SESSION ENABLE PARALLEL DML;
```

To enable parallel DML mode in a specific SQL statement, include the `ENABLE_PARALLEL_DML` SQL hint. For example:

```
INSERT /*+ ENABLE_PARALLEL_DML */ …
```

This mode is required because parallel DML and serial DML have different locking, transaction, and disk space requirements and parallel DML is disabled for a session by default.

When parallel DML is disabled, no DML is executed in parallel even if the `PARALLEL` hint is used.

When parallel DML is enabled in a session, all DML statements in this session are considered for parallel execution. When parallel DML is enabled in a SQL statement with the `ENABLE_PARALLEL_DML` hint, only that specific statement is considered for parallel execution. However, even if parallel DML is enabled, the DML operation may still execute serially if there are no parallel hints or no tables with a parallel attribute or if restrictions on parallel operations are violated.

The session's `PARALLEL DML` mode does not influence the parallelism of `SELECT` statements, DDL statements, and the query portions of DML statements. If this mode is not set, the DML operation is not parallelized, but scans or join operations within the DML statement may still be parallelized.

When the parallel DML mode has been enabled for a session, you can disable the mode for a specific SQL statement with the `DISABLE_PARALLEL_DML` SQL hint.

> **Note:**
>
> As of Oracle Database 23ai, `ALTER SESSION ENABLE PARALLEL DML` not only enables parallel DML within a session, but also allows multiple DML/PDML/QUERY operations on the same object without requiring a separate transaction commit for each operation. This can reduce the overhead of commits in parallel DML.

For more information, refer to Space Considerations for Parallel DML and Restrictions on Parallel DML.

8.5.3.3 Rules for UPDATE, MERGE, and DELETE

An update, merge, or delete operation is parallelized only under certain conditions.

An `UPDATE`, `MERGE`, and `DELETE` operation is parallelized only if at least one of the following conditions is satisfied:

- The table being updated, merged, or deleted has a `PARALLEL` declaration set by a previous `CREATE TABLE` or `ALTER TABLE` statement.
- A statement level or object level `PARALLEL` hint is specified in the DML statement.
- Automatic Degree of Parallelism (Auto DOP) has been enabled.
- Parallel DML is forced using the `ALTER SESSION FORCE PARALLEL DML` statement.

If the statement contains subqueries or updatable views, then they may also be executed in parallel based on the rules for parallelizing queries. The decision to parallelize the `UPDATE`, `MERGE`, and `DELETE` portion is independent of the query portion, and vice versa. Statement level `PARALLEL` hints or Auto DOP parallelize both the DML and the query portions.

> **See Also:**
>
> - Automatic Degree of Parallelism for information about Auto DOP
> - Limitation on the Degree of Parallelism for possible limitations on update, merge, or delete operations
> - Degree of Parallelism Rules for SQL Statements for information about the rules for determining the degree of parallelism (DOP)
> - *Oracle Database SQL Language Reference* for more information about the `ALTER SESSION` SQL statement

8.5.3.4 Rules for INSERT SELECT

An insert operation is executed in parallel only under certain conditions.

An `INSERT` operation is executed in parallel only if at least one of the following conditions is satisfied:

- The table being inserted into (the reference object) has a `PARALLEL` declaration set by a previous `CREATE TABLE` or `ALTER TABLE` statement.
- A statement level or object level `PARALLEL` hint is specified after the `INSERT` in the DML statement.
- Automatic Degree of Parallelism (Auto DOP) has been enabled.
- Parallel DML is forced using the `ALTER SESSION FORCE PARALLEL DML` statement.

The decision to parallelize the `INSERT` operation is independent of the `SELECT` operation, and vice versa. The `SELECT` operation can be parallelized based on the rules for parallelizing queries. Statement level `PARALLEL` hints or Auto DOP parallelize both the `INSERT` and the `SELECT` operations.

> **See Also:**
>
> - Automatic Degree of Parallelism for information about Auto DOP
> - Degree of Parallelism Rules for SQL Statements for information about the rules for determining the degree of parallelism (DOP)
> - *Oracle Database SQL Language Reference* for more information about the `ALTER SESSION` SQL statement

8.5.3.5 Transaction Restrictions for Parallel DML

To execute a DML operation in parallel, the parallel execution coordinator acquires parallel execution servers, and each parallel execution server executes a portion of the work under its own parallel process transaction.

Note the following conditions:

- Each parallel execution server creates a different parallel process transaction.

- If you use rollback segments instead of Automatic Undo Management, you may want to reduce contention on the rollback segments by limiting the number of parallel process transactions residing in the same rollback segment. Refer to *Oracle Database SQL Language Reference* for more information.

The coordinator also has its own coordinator transaction, which can have its own rollback segment. To ensure user-level transactional atomicity, the coordinator uses a two-phase commit protocol to commit the changes performed by the parallel process transactions.

If a PL/SQL procedure or block is executed in a parallel DML-enabled session, then this rule applies to statements in the procedure or block.

8.5.3.6 Rollback Segments

If you use rollback segments instead of Automatic Undo Management, there are some restrictions when using parallel DML.

> **See Also:**
>
> *Oracle Database SQL Language Reference* for information about restrictions for parallel DML and rollback segments

8.5.3.7 Recovery for Parallel DML

The time required to roll back a parallel DML operation is roughly equal to the time it takes to perform the forward operation.

Oracle Database supports parallel rollback after transaction and process failures, and after instance and system failures. Oracle Database can parallelize both the rolling forward stage and the rolling back stage of transaction recovery.

> **See Also:**
>
> *Oracle Database Backup and Recovery User's Guide* for details about parallel rollback

8.5.3.7.1 Transaction Recovery for User-Issued Rollback

A user-issued rollback in a transaction failure due to statement error is performed in parallel by the parallel execution coordinator and the parallel execution servers.

The rollback takes approximately the same amount of time as the forward transaction.

8.5.3.7.2 Process Recovery

Recovery from the failure of a parallel execution coordinator or parallel execution server is performed by the PMON process.

If a parallel execution server or a parallel execution coordinator fails, then PMON rolls back the work from that process and all other processes in the transaction roll back their changes.

8.5.3.7.3 System Recovery

Recovery from a system failure requires a new startup.

Recovery is performed by the SMON process and any recovery server processes spawned by SMON. Parallel DML statements may be recovered using parallel rollback. If the initialization parameter `COMPATIBLE` is set to 8.1.3 or greater, Fast-Start On-Demand Rollback enables terminated transactions to be recovered, on demand, one block at a time.

8.5.3.8 Space Considerations for Parallel DML

Parallel `UPDATE` uses the existing free space in the object, while direct-path `INSERT` gets new extents for the data.

Space usage characteristics may be different in parallel than serial execution because multiple concurrent child transactions modify the object.

8.5.3.9 Restrictions on Parallel DML

There are several restrictions that apply to parallel DML.

The following restrictions apply to parallel DML (including direct-path `INSERT`):

- Intra-partition parallelism for `UPDATE`, `MERGE`, and `DELETE` operations require that the `COMPATIBLE` initialization parameter be set to 23.0 or greater.
- Parallel DML operations cannot be done on tables with triggers.
- Replication functionality is not supported for parallel DML.
- Parallel DML can be done on tables with object columns provided the object columns are not accessed.
- Parallel DML can be done on tables with `LOB` columns provided the table is partitioned. However, intra-partition parallelism is not supported.

  For non-partitioned tables with LOB columns, parallel `INSERT` operations are supported provided that the LOB columns are declared as SecureFiles LOBs. Parallel `UPDATE`, `DELETE`, and `MERGE` operations on such tables are not supported.
- A DML operation cannot be executed in parallel if it is in a distributed transaction or if the DML or the query operation is on a remote object.
- Clustered tables are not supported.
- Parallel `UPDATE`, `DELETE`, and `MERGE` operations are not supported for private temporary tables.
- Parallel DML is not supported on a table with bitmap indexes if the table is not partitioned.

> **Note:**
>
> As of Oracle Database 23ai, some previous restrictions on DML and PDML after direct-path inserts have been lifted. You can now do the following immediately after a direct-path insert:
>
> - Query a table multiple times after one or more PDMLs on the same table within a session.
> - Perform conventional DML (serial and parallel) on the same table within the same session.
> - Multiple direct loads (serial and parallel) on the same table in the same session.
>
> Note that the restrictions above are lifted if the following conditions are satisfied:
>
> - The table is a heap table.
> - The table is in a tablespace that was created with `SEGMENT SPACE MANAGEMENT AUTO` and `AUTOALLOCATE`.
> - The initialization parameter `COMPATIBLE` is set to `23.0` or greater.

8.5.3.9.1 Partitioning Key Restriction

You can only update the partitioning key of a partitioned table to a new value if the update does not cause the row to move to a new partition.

The update is possible if the table is defined with the row movement clause enabled.

8.5.3.9.2 Function Restrictions

The function restrictions for parallel DML are the same as those for parallel DDL and parallel query.

See About Parallel Execution of Functions for more information.

8.5.3.10 Data Integrity Restrictions

The interactions of integrity constraints and parallel DML statements are introduced in the topic.

This section contains following topics:

- NOT NULL and CHECK
- UNIQUE and PRIMARY KEY
- FOREIGN KEY (Referential Integrity)
- Delete Cascade
- Self-Referential Integrity
- Deferrable Integrity Constraints

8.5.3.10.1 NOT NULL and CHECK

The integrity constraints for `NOT NULL` and `CHECK` are discussed in this topic.

NOT NULL and CHECK integrity constraints are allowed. They are not a problem for parallel DML because they are enforced on the column and row level, respectively.

8.5.3.10.2 UNIQUE and PRIMARY KEY

The integrity constraints for UNIQUE and PRIMARY KEY are discussed in this topic.

UNIQUE and PRIMARY KEY integrity constraints are allowed.

8.5.3.10.3 FOREIGN KEY (Referential Integrity)

Restrictions for referential integrity occur whenever a DML operation on one table could cause a recursive DML operation on another table.

These restrictions also apply when, to perform an integrity check, it is necessary to see simultaneously all changes made to the object being modified.

Table 8-1 lists all of the operations that are possible on tables that are involved in referential integrity constraints.

Table 8-1 Referential Integrity Restrictions

DML Statement	Issued on Parent	Issued on Child	Self-Referential
INSERT	(Not applicable)	Not parallelized	Not parallelized
MERGE	(Not applicable)	Not parallelized	Not parallelized
UPDATE No Action	Supported	Supported	Not parallelized
DELETE No Action	Supported	Supported	Not parallelized
DELETE Cascade	Not parallelized	(Not applicable)	Not parallelized

8.5.3.10.4 Delete Cascade

The delete cascade data integrity restrictions are discussed in this topic.

Deletion on tables having a foreign key with delete cascade is not parallelized because parallel execution servers attempt to delete rows from multiple partitions (parent and child tables).

8.5.3.10.5 Self-Referential Integrity

DML on tables with self-referential integrity constraints is not parallelized if the referenced keys (primary keys) are involved.

For DML on all other columns, parallelism is possible.

8.5.3.10.6 Deferrable Integrity Constraints

The deferrable integrity constraints are discussed in this topic.

If any deferrable constraints apply to the table being operated on, the DML operation is not executed in parallel.

8.5.3.11 Trigger Restrictions

A DML operation is not executed in parallel if the affected tables contain enabled triggers that may get invoked as a result of the statement.

This implies that DML statements on tables that are being replicated are not parallelized.

Relevant triggers must be disabled to parallelize DML on the table. If you enable or disable triggers, then the dependent shared cursors are invalidated.

8.5.3.12 Distributed Transaction Restrictions

The distributed transaction restrictions are discussed in this topic.

A DML operation cannot be executed in parallel if it is in a distributed transaction or if the DML or the query operation is on a remote object.

8.5.3.13 Examples of Distributed Transaction Parallelization

Several examples of distributed transaction processing are shown in this topic.

In the first example, the DML statement queries a remote object. The DML operation is executed serially without notification because it references a remote object.

```
INSERT /*+ APPEND PARALLEL (t3,2) */ INTO t3 SELECT * FROM t4@dblink;
```

In the next example, the DML operation is applied to a remote object. The `DELETE` operation is not parallelized because it references a remote object.

```
DELETE /*+ PARALLEL (t1, 2) */ FROM t1@dblink;
```

In the last example, the DML operation is in a distributed transaction. The `DELETE` operation is not executed in parallel because it occurs in a distributed transaction (which is started by the `SELECT` statement).

```
SELECT * FROM t1@dblink;
 DELETE /*+ PARALLEL (t2,2) */ FROM t2;
 COMMIT;
```

8.5.3.14 Concurrent Execution of Union All

Set operators like `UNION` or `UNION ALL` consist of multiple queries (branches) combined to a single SQL statement.

Traditionally, set operators are processed in a sequential manner. Individual branches can be processed in serial or parallel, but only one branch at a time, one branch after another. While this approach satisfies many use cases, there are situations where the processing of multiple branches of a `UNION` or `UNION ALL` statement should occur concurrently. The most typical situation is when several or all branches are remote SQL statements. In this situation, concurrent processing on all participating remote systems is desired to speed up the overall processing time without increasing the workload of any participating system.

The default behavior of concurrent execution for `UNION` or `UNION ALL` statements is controlled by the setting of the `OPTIMIZER_FEATURES_ENABLE` initialization parameter. When set to `12.1.0.1` or higher, concurrent execution is enabled by default. Any statement where at least one branch of the statement is local and is considered being processed in parallel, the entire `UNION` or `UNION ALL` statement is also processed concurrently. The system calculates the DOP for every individual local branch of the statement and chooses the highest DOP for the execution of the entire `UNION` or `UNION ALL` statement. The system then works concurrently on as many branches as possible, using the chosen DOP both for parallelization of the branches that are processed in parallel, and as concurrent workers on serial and remote statements.

When the OPTIMIZER_FEATURES_ENABLE initialization parameter is set to a value less than 12.1.0.1, concurrent execution of UNION or UNION ALL statements must be enabled explicitly by using the PQ_CONCURRENT_UNION hint.

However, unlike the sequential processing of one branch after another, the concurrent processing does not guarantee an ordered return of the results of the individual branches. If an ordered return of one branch after another is required, then you either must disable concurrent processing using the NO_PQ_CONCURRENT_UNION hint or you must augment the SQL statement to uniquely identify individual branches of the statement and to sort on this specified identifier.

UNION or UNION ALL statements that only consist of serial or remote branches are not processed concurrently unless specifically using the PQ_CONCURRENT_UNION hint. The DOP of this SQL statement is at most the number of serial and remote inputs.

Whether or not concurrent processing of a UNION or UNION ALL statement occurs can be easily identified with the execution plan of the SQL statements. When executed in parallel, the execution of serial and remote branches is managed with a row source identifiable as PX SELECTOR. Statements that are not processed concurrently show the query coordinator (QC) as coordinator of serial and remote branches.

In Example 8-4, the SQL statement consists of local and remote branches. The SQL statement loads information about gold and platinum customers from the local database, and the information about customers from three major cities from remote databases. Because the local select statements occur in parallel, this processing is automatically performed in parallel. Each serial branch is executed by only one parallel execution server process. Because each parallel execution server can execute one serial branch, they are executed concurrently.

Example 8-4 Explain Plan for UNION ALL

```
SQL> EXPLAIN PLAN FOR INSERT INTO all_customer
  SELECT * FROM GOLD_customer UNION ALL
  SELECT * FROM PLATINUM_customer UNION ALL
  SELECT * FROM SF_customer@san_francisco UNION ALL
  SELECT * FROM LA_customer@los_angeles UNION ALL
  SELECT * FROM LV_customer@las_vegas;
-------------------------------------------------------------------------------
| Id | Operation              | Name          | TQ/Ins |IN-OUT | PQ Distrib|
-------------------------------------------------------------------------------
| 0 | INSERT STATEMENT        |               |        |       |           |
| 1 |  LOAD TABLE CONVENTIONAL | ALL_CUSTOMER |        |       |           |
| 2 |   PX COORDINATOR        |               |        |       |           |
| 3 |    PX SEND QC (RANDOM)  | :TQ10003      |        | P->S  | QC (RAND) |
| 4 |     UNION-ALL           |               |        | PCWP  |           |
| 5 |      PX BLOCK ITERATOR  |               |        | PCWC  |           |
| 6 |       TABLE ACCESS FULL | GOLD_CUSTOMER|        | PCWP  |           |
| 7 |      PX BLOCK ITERATOR  |               |        | PCWC  |           |
| 8 |       TABLE ACCESS FULL | PLATINUM_CUST|        | PCWP  |           |
| 9 |      PX SELECTOR        |               |        | PCWP  |           |
|10 |       REMOTE            | SF_CUSTOMER  |        | PCWP  |           |
|11 |      PX SELECTOR        |               |        | PCWP  |           |
|12 |       REMOTE            | LA_CUSTOMER  |        | PCWP  |           |
|13 |      PX SELECTOR        |               |        | PCWP  |           |
|14 |       REMOTE            | LV_CUSTOMER  |        | PCWP  |           |
-------------------------------------------------------------------------------
```

8.5.4 About Parallel Execution of Functions

SQL statements can contain user-defined functions written in PL/SQL, in Java, or as external procedures in C that can appear as part of the SELECT list, SET clause, or WHERE clause.

When the SQL statement is parallelized, these functions are executed on a per-row basis by the parallel execution server process. Any PL/SQL package variables or Java static attributes used by the function are entirely private to each individual parallel execution process and are newly initialized when each row is processed, rather than being copied from the original session. Because of this process, not all functions generate correct results if executed in parallel.

User-written table functions can appear in the statement's `FROM` list. These functions act like source tables in that they produce row output. Table functions are initialized once during the statement at the start of each parallel execution process. All variables are entirely private to the parallel execution process.

This section contains the following topics:

- Functions in Parallel Queries
- Functions in Parallel DML and DDL Statements

8.5.4.1 Functions in Parallel Queries

User functions can be executed in parallel in a SQL query statement, or a subquery in a DML or DDL statement.

In a `SELECT` statement or a subquery in a DML or DDL statement, a user-written function may be executed in parallel in any of the following cases:

- If it has been declared with the `PARALLEL_ENABLE` keyword
- If it is declared in a package or type and has a `PRAGMA RESTRICT_REFERENCES` clause that indicates all of `WNDS`, `RNPS`, and `WNPS`
- If it is declared with `CREATE FUNCTION` and the system can analyze the body of the PL/SQL code and determine that the code neither writes to the database nor reads or modifies package variables

Other parts of a query or subquery can sometimes execute in parallel even if a given function execution must remain serial.

> **See Also:**
>
> - *Oracle Database Development Guide* for information about the `PRAGMA RESTRICT_REFERENCES` clause
> - *Oracle Database SQL Language Reference* for information about the `CREATE FUNCTION` statement

8.5.4.2 Functions in Parallel DML and DDL Statements

A user function can be executed in a parallel DML or DDL statement under certain conditions.

In a parallel DML or DDL statement, as in a parallel query, a user-written function may be executed in parallel in any of the following cases:

- If it has been declared with the `PARALLEL_ENABLE` keyword
- If it is declared in a package or type and has a `PRAGMA RESTRICT_REFERENCES` clause that indicates all of `RNDS`, `WNDS`, `RNPS`, and `WNPS`

- If it is declared with the `CREATE FUNCTION` statement and the system can analyze the body of the PL/SQL code and determine that the code neither reads nor writes to the database or reads or modifies package variables

For a parallel DML statement, any function call that cannot be executed in parallel causes the entire DML statement to be executed serially. For an `INSERT SELECT` or `CREATE TABLE AS SELECT` statement, function calls in the query portion are parallelized according to the parallel query rules described in this section. The query may be parallelized even if the remainder of the statement must execute serially, or vice versa.

8.5.5 About Other Types of Parallelism

An Oracle Database can use parallelism in multiple types of operations.

In addition to parallel SQL execution, Oracle Database can use parallelism for the following types of operations:

- Parallel recovery
- Parallel propagation (replication)
- Parallel load (external tables and the SQL*Loader utility)

Like parallel SQL, parallel recovery, propagation, and external table loads are performed by a parallel execution coordinator and multiple parallel execution servers. Parallel load using SQL*Loader, however, uses a different mechanism.

The behavior of the parallel execution coordinator and parallel execution servers may differ, depending on what kind of operation they perform (SQL, recovery, or propagation). For example, if all parallel execution servers in the pool are occupied and the maximum number of parallel execution servers has been started:

- In parallel SQL and external table loads, the parallel execution coordinator switches to serial processing.
- In parallel propagation, the parallel execution coordinator returns an error.

For a given session, the parallel execution coordinator coordinates only one kind of operation. A parallel execution coordinator cannot coordinate, for example, parallel SQL and parallel recovery or propagation at the same time.

See Also:

- *Oracle Database Utilities* for information about parallel load and SQL*Loader
- *Oracle Database Backup and Recovery User's Guide* for information about parallel media recovery
- *Oracle Database Performance Tuning Guide* for information about parallel instance recovery

8.5.6 Degree of Parallelism Rules for SQL Statements

The parallelization decision for SQL statements has two components: the decision to parallelize and the degree of parallelism (DOP).

These components are determined differently for queries, DDL operations, and DML operations.

The decision to parallelize is discussed in the following sections:

- Rules for Parallelizing Queries
- Rules for DDL Statements
- Rules for CREATE TABLE AS SELECT
- Rules for UPDATE, MERGE, and DELETE
- Rules for INSERT SELECT

The degree of parallelism for various types of SQL statements can be determined by statement or object level `PARALLEL` hints, `PARALLEL` clauses, `ALTER SESSION FORCE PARALLEL` statements, automatic degree of parallelism (Auto DOP), or table or index `PARALLEL` declarations. When more than one of these methods are used, the Oracle Database uses precedence rules to determine which method is used to determine the DOP.

Table 8-2 shows the precedence rules for determining the degree of parallelism (DOP) for various types of SQL statements. In the table, the smaller priority number indicates that the method takes precedence over higher numbers. For example, priority (1) takes precedence over priority (2), priority (3), priority (4), and priority (5).

Table 8-2 Parallelization Priority Order

Parallel Operation	Statement Level PARALLEL Hint	Object Level PARALLEL Hint	PARALLEL Clause	ALTER SESSION	Auto DOP	Parallel Declaration
Parallel query table/index scan. For more information, refer to Rules for Parallelizing Queries.	Priority (1)	Priority (2)	N/A	Priority (3) `FORCE PARALLEL QUERY`	Priority (4)	Priority (5)
Parallel `UPDATE` ,`DELETE`, or `MERGE`. For more information, refer to Rules for UPDATE, MERGE, and DELETE.	Priority (1)	Priority (2)	N/A	Priority (3) `FORCE PARALLEL DML`	Priority (4)	Priority (5) of the target table
`INSERT` operation of parallel `INSERT... SELECT` . For more information, refer to Rules for INSERT SELECT.	Priority (1)	Priority (2)	N/A	Priority (3) `FORCE PARALLEL DML`	Priority (4)	Priority (5) of table being inserted into
`SELECT` operation of `INSERT SELECT` when `INSERT` is serial. For more information, refer to Rules for INSERT SELECT.	Priority (1)	Priority (2)	N/A	Priority (3) `FORCE PARALLEL QUERY`	Priority (4)	Priority (5) of table being selected from
`CREATE` operation of parallel `CREATE TABLE AS SELECT` (partitioned or nonpartitioned table). For more information, refer to Rules for CREATE TABLE AS SELECT.	Priority (1)	N/A	Priority (4)	Priority (2) `FORCE PARALLEL DDL`	Priority (3)	N/A

Table 8-2 (Cont.) Parallelization Priority Order

Parallel Operation	Statement Level PARALLEL Hint	Object Level PARALLEL Hint	PARALLEL Clause	ALTER SESSION	Auto DOP	Parallel Declaration
`SELECT` operation of `CREATE TABLE AS SELECT` when `CREATE` is serial. For more information, refer to Rules for CREATE TABLE AS SELECT.	Priority (1)	Priority (2)	N/A	Priority (3) `FORCE PARALLEL QUERY`	Priority (4)	Priority (5)
Other DDL operations. For more information, refer to Rules for DDL Statements.	N/A	N/A	Priority (3)	Priority (1) `FORCE PARALLEL DDL`	Priority (2)	N/A

See Also:

- *Oracle Database SQL Language Reference* for information about the `PARALLEL` hint
- *Oracle Database SQL Language Reference* for information about the `PARALLEL` clause
- *Oracle Database SQL Language Reference* for more information about the `ALTER SESSION` SQL statement

8.6 About Initializing and Tuning Parameters for Parallel Execution

You can use parameters to initialize and tune parallel execution.

Oracle Database computes defaults for the parallel execution parameters based on the value at database startup of `CPU_COUNT` and `PARALLEL_THREADS_PER_CPU`. The parameters can also be manually tuned, increasing or decreasing their values to suit specific system configurations or performance goals. For example, on systems where parallel execution is never used, `PARALLEL_MAX_SERVERS` can be set to zero.

You can also manually tune parallel execution parameters. Parallel execution is enabled by default.

Initializing and tuning parallel execution is discussed in the following topics:

- Default Parameter Settings
- Forcing Parallel Execution for a Session
- Tuning General Parameters for Parallel Execution

8.6.1 Default Parameter Settings

Oracle Database automatically sets parallel execution parameters by default.

The parallel execution parameters are shown in Table 8-3.

Table 8-3 Parameters and Their Defaults

Parameter	Default	Comments
PARALLEL_ADAPTIVE_MULTI_USER	FALSE	Causes parallel execution SQL to throttle degree of parallelism (DOP) requests to prevent system overload. PARALLEL_ADAPTIVE_MULTI_USER is deprecated in Oracle Database 12c Release 2 (12.2.0.1) to be desupported in a future release. Oracle recommends using parallel statement queuing instead.
PARALLEL_DEGREE_LIMIT	CPU	Controls the maximum DOP a statement can have when automatic DOP is in use. The maximum DOP is SUM(CPU_COUNT)*PARALLEL_THREADS_PER_CPU The value AUTO for PARALLEL_DEGREE_LIMIT has the same functionality as the value CPU.
PARALLEL_DEGREE_POLICY	MANUAL	Controls whether auto DOP, parallel statement queuing and in-memory parallel execution are used. By default, all of these features are disabled.
PARALLEL_EXECUTION_MESSAGE_SIZE	16 KB	Specifies the size of the buffers used by the parallel execution servers to communicate among themselves and with the query coordinator. These buffers are allocated out of the shared pool.
PARALLEL_FORCE_LOCAL	FALSE	Restricts parallel execution to the current Oracle RAC instance.
PARALLEL_INSTANCE_GROUP	None. By default, parallel execution is enabled across all currently active instances.	Lets you restrict parallel query operations to a limited number of instances. Used in conjunction with services and also with the deprecated parameter INSTANCE_GROUPS.
PARALLEL_MAX_SERVERS	See PARALLEL_MAX_SERVERS.	Specifies the maximum number of parallel execution processes and parallel recovery processes for an instance. As demand increases, Oracle Database increases the number of processes from the number created at instance startup up to this value. If you set this parameter too low, some queries may not have a parallel execution process available to them during query processing. If you set it too high, memory resource shortages may occur during peak periods, which can degrade performance.
PARALLEL_MIN_DEGREE	1	Controls the minimum degree of parallelism computed by automatic degree of parallelism.
PARALLEL_MIN_SERVERS	CPU_COUNT * PARALLEL_THREADS_PER_CPU * 2	Specifies the number of parallel execution processes to be started and reserved for parallel operations, when Oracle Database is started up. Increasing this setting can help balance the startup cost of a parallel statement, but requires greater memory usage as these parallel execution processes are not removed until the database is shut down.
PARALLEL_MIN_PERCENT	0	Specifies the minimum percentage of requested parallel execution processes required for parallel execution. With the default value of 0, a parallel statement executes serially if no parallel server processes are available.
PARALLEL_MIN_TIME_THRESHOLD	AUTO	Specifies the execution time, as estimated by the optimizer, above which a statement is considered for automatic parallel query and automatic derivation of DOP. Note that the interpretation of AUTO is dependent on whether or not Database In-Memory is used.

Table 8-3 (Cont.) Parameters and Their Defaults

Parameter	Default	Comments
PARALLEL_SERVERS_TARGET	See PARALLEL_SERVERS_TARGET.	Specifies the number of parallel execution server processes available to run queries before parallel statement queuing is used. Note that parallel statement queuing is only active if PARALLEL_DEGREE_POLICY is set to AUTO.
PARALLEL_THREADS_PER_CPU	1	Describes the number of parallel execution processes or threads that a CPU can handle during parallel execution.

You can set some parameters in such a way that Oracle Database is constrained. For example, if you set PROCESSES to 20, you are not be able to get 25 child processes.

> **See Also:**
>
> *Oracle Database Reference* for more information about initialization parameters

8.6.2 Forcing Parallel Execution for a Session

Youu can force parallelism for a session.

If you are sure you want to execute in parallel and want to avoid setting the DOP for a table or modifying the queries involved, you can force parallelism with the following statement:

```
ALTER SESSION FORCE PARALLEL QUERY;
```

All subsequent queries are executed in parallel provided no restrictions are violated. You can also force DML and DDL statements. This clause overrides any parallel clause specified in subsequent statements in the session, but is overridden by a parallel hint.

In typical OLTP environments, for example, the tables are not set parallel, but nightly batch scripts may want to collect data from these tables in parallel. By setting the DOP in the session, the user avoids altering each table in parallel and then altering it back to serial when finished.

8.6.3 Tuning General Parameters for Parallel Execution

The discussion about tuning general parameters for parallel execution is introduced in the topic.

This section discusses the following topics:

- Parameters Establishing Resource Limits for Parallel Operations
- Parameters Affecting Resource Consumption
- Parameters Related to I/O

8.6.3.1 Parameters Establishing Resource Limits for Parallel Operations

You can set initialization parameters to determine resource limits.

The parameters that establish resource limits are discussed in the following topics:

- PARALLEL_FORCE_LOCAL
- PARALLEL_MAX_SERVERS
- PARALLEL_MIN_PERCENT
- PARALLEL_MIN_SERVERS
- PARALLEL_MIN_TIME_THRESHOLD
- PARALLEL_SERVERS_TARGET
- SHARED_POOL_SIZE
- Additional Memory Requirements for Message Buffers
- Monitor Memory Usage After Processing Begins

> **See Also:**
>
> *Oracle Database Reference* for information about initialization parameters

8.6.3.1.1 PARALLEL_FORCE_LOCAL

The `PARALLEL_FORCE_LOCAL` parameter specifies whether a SQL statement executed in parallel is restricted to a single instance in an Oracle RAC environment.

By setting this parameter to `TRUE`, you restrict the scope of the parallel server processed to the single Oracle RAC instance where the query coordinator is running.

The recommended value for the `PARALLEL_FORCE_LOCAL` parameter is `FALSE`.

> **See Also:**
>
> *Oracle Database Reference* for information about the `PARALLEL_FORCE_LOCAL` initialization parameter

8.6.3.1.2 PARALLEL_MAX_SERVERS

The `PARALLEL_MAX_SERVERS` parameter specifies the maximum number of parallel execution processes and parallel recovery processes for an instance.

As demand increases, Oracle Database increases the number of processes from the number created at instance startup up to this value.

For example, setting the value to 64 enables you to run four parallel queries simultaneously, if each query is using two worker sets with a DOP of `8` for each set.

When Users Have Too Many Processes

When concurrent users have too many query server processes, memory contention (paging), I/O contention, or excessive context switching can occur.

This contention can reduce system throughput to a level lower than if parallel execution were not used. Increase the `PARALLEL_MAX_SERVERS` value only if the system has sufficient memory and I/O bandwidth for the resulting load.

You can use performance monitoring tools of the operating system to determine how much memory, swap space and I/O bandwidth are free. Look at the run queue lengths for both your CPUs and disks, and the service time for I/O operations on the system. Verify that the system has sufficient swap space to add more processes. Limiting the total number of query server processes might restrict the number of concurrent users who can execute parallel operations, but system throughput tends to remain stable.

When to Limit the Number of Resources for a User using a Consumer Group

When necessary, you can limit the amount of parallelism available to a given user by establishing a resource consumer group for the user.

Do this to limit the number of sessions, concurrent logons, and the number of parallel processes that any one user or group of users can have.

Each query server process working on a parallel execution statement is logged on with a session ID. Each process counts against the user's limit of concurrent sessions. For example, to limit a user to 10 parallel execution processes, set the user's limit to 11. One process is for the parallel execution coordinator and the other 10 consist of two sets of query servers. This would allow one session for the parallel execution coordinator and 10 sessions for the parallel execution processes.

See Also:

- *Oracle Database Reference* for information about the `PARALLEL_MAX_SERVERS` initialization parameter
- *Oracle Database Administrator's Guide* for more information about managing resources with user profiles
- *Oracle Real Application Clusters Administration and Deployment Guide* for more information about querying `GV$` views

8.6.3.1.3 PARALLEL_MIN_PERCENT

The `PARALLEL_MIN_PERCENT` parameter enables users to wait for an acceptable DOP, depending on the application in use.

The recommended value for the `PARALLEL_MIN_PERCENT` parameter is 0 (zero). Setting this parameter to values other than 0 (zero) causes Oracle Database to return an error when the requested DOP cannot be satisfied by the system at a given time. For example, if you set `PARALLEL_MIN_PERCENT` to 50, which translates to 50 percent, and the DOP is reduced by 50 percent or greater because of the adaptive algorithm or because of a resource limitation, then Oracle Database returns `ORA-12827`. For example:

```
SELECT /*+ FULL(e) PARALLEL(e, 8) */ d.department_id, SUM(SALARY)
  FROM employees e, departments d WHERE e.department_id = d.department_id
  GROUP BY d.department_id ORDER BY d.department_id;
```

Oracle Database responds with this message:

```
ORA-12827: insufficient number of parallel execution servers
```

> **See Also:**
>
> *Oracle Database Reference* for information about the `PARALLEL_MIN_PERCENT` initialization parameter

8.6.3.1.4 PARALLEL_MIN_SERVERS

The `PARALLEL_MIN_SERVERS` parameter specifies the number of processes to be started in a single instance that are reserved for parallel operations.

Setting `PARALLEL_MIN_SERVERS` balances the startup cost against memory usage. Processes started using `PARALLEL_MIN_SERVERS` do not exit until the database is shut down. This way, when a query is issued, the processes are likely to be available.

> **See Also:**
>
> *Oracle Database Reference* for information about the `PARALLEL_MIN_SERVERS` initialization parameter

8.6.3.1.5 PARALLEL_MIN_TIME_THRESHOLD

The `PARALLEL_MIN_TIME_THRESHOLD` parameter specifies the minimum execution time a statement should have before the statement is considered for automatic degree of parallelism.

> **See Also:**
>
> *Oracle Database Reference* for information about the `PARALLEL_MIN_TIME_THRESHOLD` initialization parameter

8.6.3.1.6 PARALLEL_SERVERS_TARGET

The `PARALLEL_DEGREE_POLICY` parameter specifies the number of parallel server processes allowed to run parallel statements before statement queuing is used.

When `PARALLEL_DEGREE_POLICY` is set to `AUTO`, statements that require parallel execution are queued if the number of parallel processes currently in use on the system equals or is greater than `PARALLEL_SERVERS_TARGET`. This is not the maximum number of parallel server processes allowed on a system (that is controlled by `PARALLEL_MAX_SERVERS`). However, `PARALLEL_SERVERS_TARGET` and parallel statement queuing is used to ensure that each statement that requires parallel execution is allocated the necessary parallel server resources and the system is not flooded with too many parallel server processes.

> **See Also:**
>
> *Oracle Database Reference* for information about the `PARALLEL_SERVERS_TARGET` initialization parameter

8.6.3.1.7 SHARED_POOL_SIZE

The `SHARED_POOL_SIZE` parameter specifies the memory size of the shared pool.

Parallel execution requires memory resources in addition to those required by serial SQL execution. Additional memory is used for communication and passing data between query server processes and the query coordinator.

Oracle Database allocates memory for query server processes from the shared pool. Tune the shared pool as follows:

- Allow for other clients of the shared pool, such as shared cursors and stored procedures.
- Remember that larger values improve performance in multiuser systems, but smaller values use less memory.
- You can then monitor the number of buffers used by parallel execution and compare the `shared pool PX msg pool` to the current high water mark reported in output from the view `V$PX_PROCESS_SYSSTAT`.

> **Note:**
>
> If you do not have enough memory available, error message 12853 occurs (insufficient memory for PX buffers: current *string*K, max needed *string*K). This is caused by having insufficient SGA memory available for PX buffers. You must reconfigure the SGA to have at least (`MAX - CURRENT`) bytes of additional memory.

By default, Oracle Database allocates parallel execution buffers from the shared pool.

If Oracle Database displays the following error on startup, you should reduce the value for `SHARED_POOL_SIZE` low enough so your database starts:

```
ORA-27102: out of memory
SVR4 Error: 12: Not enough space
```

After reducing the value of `SHARED_POOL_SIZE`, you might see the error:

```
ORA-04031: unable to allocate 16084 bytes of shared memory
   ("SHARED pool","unknown object","SHARED pool heap","PX msg pool")
```

If so, execute the following query to determine why Oracle Database could not allocate the 16,084 bytes:

```
SELECT NAME, SUM(BYTES) FROM V$SGASTAT WHERE UPPER(POOL)='SHARED POOL'
  GROUP BY ROLLUP (NAME);
```

Your output should resemble the following:

```
NAME                      SUM(BYTES)
------------------------- ----------
PX msg pool                  1474572
```

```
free memory                   562132
                             2036704
```

If you specify SHARED_POOL_SIZE and the amount of memory you specify to reserve is bigger than the pool, Oracle Database does not allocate all the memory it can get. Instead, it leaves some space. When the query runs, Oracle Database tries to get what it needs. Oracle Database uses the 560 KB and needs another 16 KB when it fails. The error does not report the cumulative amount that is needed. The best way of determining how much more memory is needed is to use the formulas in Additional Memory Requirements for Message Buffers.

To resolve the problem in the current example, increase the value for SHARED_POOL_SIZE. As shown in the sample output, the SHARED_POOL_SIZE is about 2 MB. Depending on the amount of memory available, you could increase the value of SHARED_POOL_SIZE to 4 MB and attempt to start your database. If Oracle Database continues to display an ORA-4031 message, gradually increase the value for SHARED_POOL_SIZE until startup is successful.

> **See Also:**
>
> *Oracle Database Reference* for information about the SHARED_POOL_SIZE initialization parameter

8.6.3.1.8 Additional Memory Requirements for Message Buffers

Additional memory requirements for message buffers and cursors when using parallel execution plans are discussed in this topic.

After you determine the initial setting for the shared pool, you must calculate additional memory requirements for message buffers and determine how much additional space you need for cursors.

Required Memory for Message Buffers

You must increase the value for the SHARED_POOL_SIZE parameter to accommodate message buffers. The message buffers allow query server processes to communicate with each other.

Oracle Database uses a fixed number of buffers for each virtual connection between producer query servers and consumer query servers. Connections increase as the square of the DOP increases. For this reason, the maximum amount of memory used by parallel execution is bound by the highest DOP allowed on your system. You can control this value by using either the PARALLEL_MAX_SERVERS parameter or by using policies and profiles.

To calculate the amount of memory required, use one of the following formulas:

- For SMP systems:

  ```
  mem in bytes = (3 x size x users x groups x connections)
  ```

- For Oracle Real Application Clusters and MPP systems:

  ```
  mem in bytes = ((3 x local) + (2 x remote)) x (size x users x groups)
    / instances
  ```

Each instance uses the memory computed by the formula.

The terms are:

- SIZE = PARALLEL_EXECUTION_MESSAGE_SIZE

- USERS = the number of concurrent parallel execution users that you expect to have running with the optimal DOP
- GROUPS = the number of query server process groups used for each query

  A simple SQL statement requires only one group. However, if your queries involve subqueries which are processed in parallel, then Oracle Database uses an additional group of query server processes.
- CONNECTIONS = (DOP^2 + 2 x DOP)

  If your system is a cluster or MPP, then you should account for the number of instances because this increases the DOP. In other words, using a DOP of 4 on a two-instance cluster results in a DOP of 8. A value of PARALLEL_MAX_SERVERS times the number of instances divided by four is a conservative estimate to use as a starting point.
- LOCAL = CONNECTIONS/INSTANCES
- REMOTE = CONNECTIONS - LOCAL

Add this amount to your original setting for the shared pool. However, before setting a value for either of these memory structures, you must also consider additional memory for cursors, as explained in the following section.

Additional Memory for Cursors

Parallel execution plans consume more space in the SQL area than serial execution plans. You should regularly monitor shared pool resource use to ensure that the memory used by both messages and cursors can accommodate your system's processing requirements.

8.6.3.1.9 Monitor Memory Usage After Processing Begins

Whether you are using automated or manual tuning, you should monitor usage on an on-going basis to ensure the size of memory is not too large or too small.

The formulas in this section are just starting points. To ensure the correct memory size, tune the shared pool using the following query:

```
SELECT POOL, NAME, SUM(BYTES) FROM V$SGASTAT WHERE POOL LIKE '%pool%'
  GROUP BY ROLLUP (POOL, NAME);
```

Your output should resemble the following:

```
POOL        NAME                         SUM(BYTES)
----------- -------------------------- ----------
shared pool Checkpoint queue                 38496
shared pool KGFF heap                         1964
shared pool KGK heap                          4372
shared pool KQLS heap                      1134432
shared pool LRMPD SGA Table                  23856
shared pool PLS non-lib hp                    2096
shared pool PX subheap                      186828
shared pool SYSTEM PARAMETERS                55756
shared pool State objects                  3907808
shared pool character set memory             30260
shared pool db_block_buffers                200000
shared pool db_block_hash_buckets            33132
shared pool db_files                        122984
shared pool db_handles                       52416
shared pool dictionary cache                198216
shared pool dlm shared memory              5387924
shared pool event statistics per sess       264768
shared pool fixed allocation callback         1376
```

```
shared pool free memory                  26329104
shared pool gc_*                            64000
shared pool latch nowait fails or sle       34944
shared pool library cache                 2176808
shared pool log_buffer                      24576
shared pool log_checkpoint_timeout          24700
shared pool long op statistics array        30240
shared pool message pool freequeue         116232
shared pool miscellaneous                  267624
shared pool processes                       76896
shared pool session param values            41424
shared pool sessions                       170016
shared pool sql area                      9549116
shared pool table columns                  148104
shared pool trace_buffers_per_process     1476320
shared pool transactions                    18480
shared pool trigger inform                  24684
shared pool                              52248968
                                         90641768
```

Evaluate the memory used as shown in your output, and alter the setting for `SHARED_POOL_SIZE` based on your processing needs.

To obtain more memory usage statistics, execute the following query:

```
SELECT * FROM V$PX_PROCESS_SYSSTAT WHERE STATISTIC LIKE 'Buffers%';
```

Your output should resemble the following:

```
STATISTIC                           VALUE
-------------------                 -----
Buffers Allocated                   23225
Buffers Freed                       23225
Buffers Current                         0
Buffers HWM                          3620
```

The amount of memory used appears in the `Buffers Current` and `Buffers HWM` statistics. Calculate a value in bytes by multiplying the number of buffers by the value for `PARALLEL_EXECUTION_MESSAGE_SIZE`. Compare the high water mark to the parallel execution message pool size to determine if you allocated too much memory. For example, in the first output, the value for large pool as shown in `px msg pool` is 38,092,812 or 38 MB. The `Buffers HWM` from the second output is 3,620, which when multiplied by a parallel execution message size of 4,096 is 14,827,520, or approximately 15 MB. In this case, the high water mark has reached approximately 40 percent of its capacity.

8.6.3.2 Parameters Affecting Resource Consumption

The parameters affecting resource consumption are discussed in the topic.

> **Note:**
>
> Before considering the following section, you should read the descriptions of the `MEMORY_TARGET` and `MEMORY_MAX_TARGET` initialization parameters for details. The `PGA_AGGREGATE_TARGET` initialization parameter need not be set as `MEMORY_TARGET` autotunes the SGA and PGA components.

The first group of parameters discussed in this section affects memory and resource consumption for all parallel operations, in particular, for parallel execution. These parameters are:

- PGA_AGGREGATE_TARGET
- PARALLEL_EXECUTION_MESSAGE_SIZE

A second subset of parameters are discussed in Parameters Affecting Resource Consumption for Parallel DML and Parallel DDL.

To control resource consumption, you should configure memory at two levels:

- At the database level, so the system uses an appropriate amount of memory from the operating system.
- At the operating system level for consistency.

  On some platforms, you might need to set operating system parameters that control the total amount of virtual memory available, totalled across all processes.

A large percentage of the memory used in data warehousing operations (compared to OLTP) is more dynamic. This memory comes from Process Global Area (PGA), and both the size of process memory and the number of processes can vary greatly. Use the `PGA_AGGREGATE_TARGET` initialization parameter to control both the process memory and the number of processes in such cases. Explicitly setting `PGA_AGGREGATE_TARGET` along with `MEMORY_TARGET` ensures that autotuning still occurs but `PGA_AGGREGATE_TARGET` is not tuned below the specified value.

See Also:

- *Oracle Database Performance Tuning Guide* for descriptions of the `MEMORY_TARGET` and `MEMORY_MAX_TARGET` initialization parameters
- *Oracle Database Administrator's Guide* for additional information about the use of the `MEMORY_TARGET` and `MEMORY_MAX_TARGET` initialization parameters

8.6.3.2.1 PGA_AGGREGATE_TARGET

You can enable automatic PGA memory management with the setting of initialization parameters, such as `PGA_AGGREGATE_TARGET`.

You can simplify and improve the way PGA memory is allocated by enabling automatic PGA memory management. In this mode, Oracle Database dynamically adjusts the size of the portion of the PGA memory dedicated to work areas, based on an overall PGA memory target explicitly set by the DBA. To enable automatic PGA memory management, you must set the initialization parameter `PGA_AGGREGATE_TARGET`. For new installations, `PGA_AGGREGATE_TARGET` and `SGA_TARGET` are set automatically by the database configuration assistant (DBCA), and `MEMORY_TARGET` is zero. That is, automatic memory management is disabled. Therefore, automatic tuning of the aggregate PGA is enabled by default. However, the aggregate PGA does not grow unless you enable automatic memory management by setting `MEMORY_TARGET` to a nonzero value.

> **See Also:**
>
> - *Oracle Database Reference* for more information about the `PGA_AGGREGATE_TARGET` initialization parameter
> - *Oracle Database Performance Tuning Guide* for descriptions of how to use `PGA_AGGREGATE_TARGET` in different scenarios

8.6.3.2.1.1 HASH_AREA_SIZE

This parameter has been deprecated.

`HASH_AREA_SIZE` has been deprecated and you should use `PGA_AGGREGATE_TARGET` instead. For information, refer to PGA_AGGREGATE_TARGET.

8.6.3.2.1.2 SORT_AREA_SIZE

This parameter has been deprecated.

`SORT_AREA_SIZE` has been deprecated and you should use `PGA_AGGREGATE_TARGET` instead. For information, refer to PGA_AGGREGATE_TARGET.

8.6.3.2.2 PARALLEL_EXECUTION_MESSAGE_SIZE

The `PARALLEL_EXECUTION_MESSAGE_SIZE` parameter specifies the size of the buffer used for parallel execution messages.

The default value of `PARALLEL_EXECUTION_MESSAGE_SIZE` is operating system-specific, but is typically 16 K. This value should be adequate for most applications.

> **See Also:**
>
> *Oracle Database Reference* for information about the `PARALLEL_EXECUTION_MESSAGE_TIME` initialization parameter

8.6.3.2.3 Parameters Affecting Resource Consumption for Parallel DML and Parallel DDL

The parameters affecting resource consumption for parallel DML and parallel DDL operations are introduced in this topic.

The parameters that affect parallel DML and parallel DDL resource consumption are:

- TRANSACTIONS
- FAST_START_PARALLEL_ROLLBACK
- DML_LOCKS

Parallel insert, update, and delete operations require more resources than serial DML operations. Similarly, `PARALLEL CREATE TABLE AS SELECT` and `PARALLEL CREATE INDEX` can require more resources. For this reason, you may need to increase the value of several additional initialization parameters. These parameters do *not* affect resources for queries.

> **See Also:**
>
> *Oracle Database Reference* for information about initialization parameters

8.6.3.2.3.1 TRANSACTIONS

The `TRANSACTIONS` parameter affects the number of transactions under parallel DML and DDL.

For parallel DML and DDL, each query server process starts a transaction. The parallel execution coordinator uses the two-phase commit protocol to commit transactions; therefore, the number of transactions being processed increases by the DOP. Consequently, you might need to increase the value of the `TRANSACTIONS` initialization parameter.

The `TRANSACTIONS` parameter specifies the maximum number of concurrent transactions. The default value of `TRANSACTIONS` assumes no parallelism. For example, if you have a DOP of 20, you have 20 more new server transactions (or 40, if you have two server sets) and 1 coordinator transaction. In this case, you should increase `TRANSACTIONS` by 21 (or 41) if the transactions are running in the same instance. If you do not set this parameter, Oracle Database sets it to a value equal to 1.1 x `SESSIONS`. This discussion does not apply if you are using server-managed undo.

8.6.3.2.3.2 FAST_START_PARALLEL_ROLLBACK

If a system fails when there are uncommitted parallel DML or DDL transactions, you can speed up transaction recovery during startup by using the `FAST_START_PARALLEL_ROLLBACK` parameter.

The `FAST_START_PARALLEL_ROLLBACK` parameter controls the DOP used when recovering terminated transactions. Terminated transactions are transactions that are active before a system failure. By default, the DOP is chosen to be at most two times the value of the `CPU_COUNT` parameter.

If the default DOP is insufficient, set the parameter to `HIGH`. This gives a maximum DOP of at most four times the value of the `CPU_COUNT` parameter. This feature is available by default.

8.6.3.2.3.3 DML_LOCKS

The `DML_LOCKS` parameter should be set to account for the number of locks held by a parallel DML operation.

The `DML_LOCKS` parameter specifies the maximum number of DML locks. Its value should equal the total number of locks on all tables referenced by all users. A parallel DML operation's lock requirement is very different from serial DML. Parallel DML holds many more locks, so you should increase the value of the `DML_LOCKS` parameter by equal amounts.

> **Note:**
>
> Parallel DML operations are not performed when the table lock of the target table is disabled.

Table 8-4 shows the types of locks acquired by coordinator and parallel execution server processes for different types of parallel DML statements. Using this information, you can determine the value required for these parameters.

Table 8-4 Locks Acquired by Parallel DML Statements

Type of Statement	Coordinator Process Acquires:	Each Parallel Execution Server Acquires:
Parallel `UPDATE` or `DELETE` into partitioned table; `WHERE` clause pruned to a subset of partitions or subpartitions	1 table lock SX 1 partition lock X for each pruned partition or subpartition	1 table lock SX 1 partition lock `NULL` for each pruned partition or subpartition owned by the query server process 1 partition-wait lock S for each pruned partition or subpartition owned by the query server process
Parallel row-migrating `UPDATE` into partitioned table; `WHERE` clause pruned to a subset of partitions or subpartitions	1 table lock SX 1 partition X lock for each pruned partition or subpartition 1 partition lock SX for all other partitions or subpartitions	1 table lock SX 1 partition lock `NULL` for each pruned partition or subpartition owned by the query server process 1 partition-wait lock S for each pruned partition owned by the query server process 1 partition lock SX for all other partitions or subpartitions
Parallel `UPDATE`, `MERGE`, `DELETE`, or `INSERT` into partitioned table	1 table lock SX Partition locks X for all partitions or subpartitions	1 table lock SX 1 partition lock `NULL` for each partition or subpartition 1 partition-wait lock S for each partition or subpartition
Parallel `INSERT` into partitioned table; destination table with partition or subpartition clause	1 table lock SX 1 partition lock X for each specified partition or subpartition	1 table lock SX 1 partition lock `NULL` for each specified partition or subpartition 1 partition-wait lock S for each specified partition or subpartition
Parallel `INSERT` into nonpartitioned table	1 table lock X	None

> **Note:**
>
> Table, partition, and partition-wait DML locks all appear as TM locks in the `V$LOCK` view.

Consider a table with 600 partitions running with a DOP of 100. Assume all partitions are involved in a parallel `UPDATE` or `DELETE` statement with no row-migrations.

The coordinator acquires:

- 1 table lock SX
- 600 partition locks X

Total server processes acquire:

- 100 table locks SX
- 600 partition locks NULL
- 600 partition-wait locks S

8.6.3.3 Parameters Related to I/O

The parameters that affect I/O are introduced in this topic:

The parameters that affect I/O are:

- DB_CACHE_SIZE
- DB_BLOCK_SIZE
- DB_FILE_MULTIBLOCK_READ_COUNT
- DISK_ASYNCH_IO and TAPE_ASYNCH_IO

These parameters also affect the optimizer, which ensures optimal performance for parallel execution of I/O operations.

> **See Also:**
>
> *Oracle Database Reference* for information about initialization parameters

8.6.3.3.1 DB_CACHE_SIZE

The `DB_CACHE_SIZE` parameter sets the size of the `DEFAULT` buffer pool for buffers with the primary block size.

When you perform parallel update, merge, and delete operations, the buffer cache behavior is very similar to any OLTP system running a high volume of updates.

8.6.3.3.2 DB_BLOCK_SIZE

The `DB_BLOCK_SIZE` parameter sets the size of Oracle database blocks.

The recommended value for this parameter is 8 KB or 16 KB.

Set the database block size when you create the database. If you are creating a new database, use a large block size such as 8 KB or 16 KB.

8.6.3.3.3 DB_FILE_MULTIBLOCK_READ_COUNT

The `DB_FILE_MULTIBLOCK_READ_COUNT` parameter determines how many database blocks are read with a single operating system `READ` call.

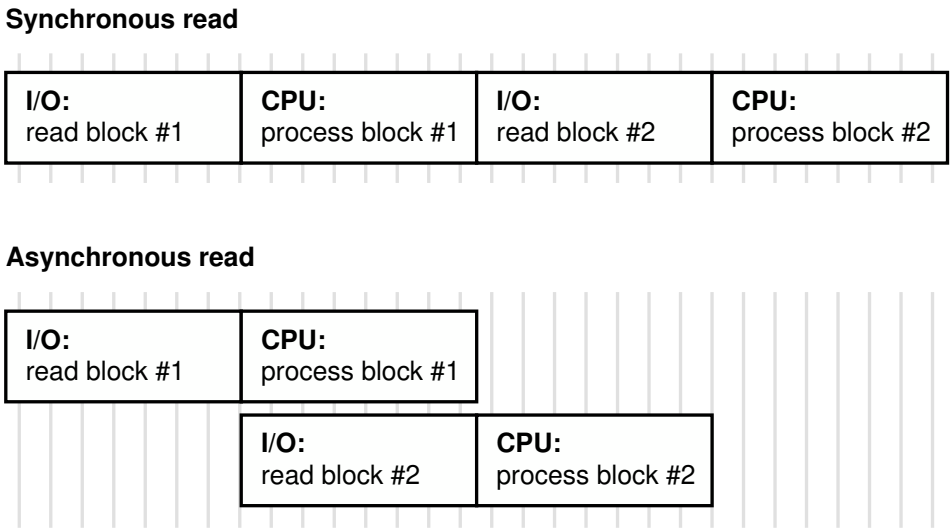
The default value of this parameter is a value that corresponds to the maximum I/O size that can be performed efficiently. The maximum I/O size value is platform-dependent and is 1 MB for most platforms. If you set `DB_FILE_MULTIBLOCK_READ_COUNT` to an excessively high value, your operating system lowers the value to the highest allowable level when you start your database.

8.6.3.3.4 DISK_ASYNCH_IO and TAPE_ASYNCH_IO

The `DISK_ASYNCH_IO` and `TAPE_ASYNCH_IO` parameters enable or disable the operating system's asynchronous I/O facility.

The recommended value for the both `DISK_ASYNCH_IO` and `TAPE_ASYNCH_IO`parameters is `TRUE`. These parameters enable query server processes to overlap I/O requests with processing when performing table scans. If the operating system supports asynchronous I/O, leave these parameters at the default value of `TRUE`. Figure 8-6 illustrates how asynchronous read works.

Figure 8-6 Asynchronous Read



Asynchronous operations are currently supported for parallel table scans, hash joins, sorts, and serial table scans. However, this feature can require operating system-specific configuration and may not be supported on all platforms.

8.7 Monitoring Parallel Execution Performance

You should perform the following types of monitoring when trying to diagnose parallel execution performance problems.

These types of monitoring include:

- Monitoring Parallel Execution Performance with Dynamic Performance Views
- Monitoring Session Statistics
- Monitoring System Statistics
- Monitoring Operating System Statistics

> **See Also:**
>
> *Oracle Database Reference* for information about dynamic views

8.7.1 Monitoring Parallel Execution Performance with Dynamic Performance Views

You can monitor parallel execution performance with dynamic performance views.

Oracle Database real-time monitoring feature enables you to monitor the performance of SQL statements while they are executing. SQL monitoring is automatically started when a SQL statement runs parallel or when it has consumed at least 5 seconds of CPU or I/O time for a single execution.

After your system has run for a few days, you should monitor parallel execution performance statistics to determine whether your parallel processing is optimal. Do this using any of the views discussed in this section.

In Oracle Real Application Clusters, global versions of the views described in this section aggregate statistics from multiple instances. The global views have names beginning with `G`, such as `GV$FILESTAT` for `V$FILESTAT`, and so on.

> **See Also:**
>
> *Oracle Database SQL Tuning Guide* for more information about monitoring performance

8.7.1.1 V$PX_BUFFER_ADVICE

You can monitor parallel execution performance with the `V$PX_BUFFER_ADVICE` dynamic performance view.

The `V$PX_BUFFER_ADVICE` view provides statistics on historical and projected maximum buffer usage by all parallel queries. You can consult this view to reconfigure SGA size in response to insufficient memory problems for parallel queries.

8.7.1.2 V$PX_SESSION

You can monitor parallel execution performance with the `V$PX_SESSION` dynamic performance view.

The `V$PX_SESSION` view shows data about query server sessions, groups, sets, and server numbers. It also displays real-time data about the processes working on behalf of parallel execution. This table includes information about the requested degree of parallelism (DOP) and the actual DOP granted to the operation.

8.7.1.3 V$PX_SESSTAT

You can monitor parallel execution performance with the `V$PX_SESSTAT` dynamic performance view.

The `V$PX_SESSTAT` view provides a join of the session information from `V$PX_SESSION` and the `V$SESSTAT` table. Thus, all session statistics available to a standard session are available for all sessions performed using parallel execution.

8.7.1.4 V$PX_PROCESS

You can monitor parallel execution performance with the `V$PX_PROCESS` dynamic performance view.

The `V$PX_PROCESS` view contains information about the parallel processes, including status, session ID, process ID, and other information.

8.7.1.5 V$PX_PROCESS_DETAIL

`V$PX_PROCESS_DETAIL` lists statistics for each of the active PX (Parallel Execution) servers on an instance.

This dynamic performance view is similar to `V$PX_PROCESS`, but provides more detailed information. It may offer additional insight into PX process management in the database, which can help you to better adjust the environment to the actual workload.

> **See Also:**
>
> V$PX_PROCESS_DETAIL

8.7.1.6 V$PX_PROCESS_SYSSTAT

You can monitor parallel execution performance with the `V$PX_PROCESS_SYSSTAT` dynamic performance view.

The `V$PX_PROCESS_SYSSTAT` view shows the status of query servers and provides buffer allocation statistics.

8.7.1.7 V$PQ_SESSTAT

You can monitor parallel execution performance with the `V$PQ_SESSTAT` dynamic performance view.

The `V$PQ_SESSTAT` view shows the status of all current server groups in the system such as data about how queries allocate processes and how the multiuser and load balancing algorithms are affecting the default and hinted values.

You might need to adjust some parameter settings to improve performance after reviewing data from these views. In this case, refer to the discussion of Tuning General Parameters for Parallel Execution. Query these views periodically to monitor the progress of long-running parallel operations.

For many dynamic performance views, you must set the parameter `TIMED_STATISTICS` to `TRUE` in order for Oracle Database to collect statistics for each view. You can use the `ALTER SYSTEM` or `ALTER SESSION` statements to turn `TIMED_STATISTICS` on and off.

8.7.1.8 V$PQ_TQSTAT

You can monitor parallel execution performance with the V$PQ_TQSTAT dynamic performance view.

As a simple example, consider a hash join between two tables, with a join on a column with only two distinct values. At best, this hash function has one hash value to parallel execution server A and the other to parallel execution server B. A DOP of two is fine, but, if it is four, then at least two parallel execution servers have no work. To discover this type of deviation, use a query similar to the following example:

```
SELECT dfo_number, tq_id, server_type, process, num_rows
FROM V$PQ_TQSTAT ORDER BY dfo_number DESC, tq_id, server_type, process;
```

The best way to resolve this problem might be to choose a different join method; a nested loop join might be the best option. Alternatively, if one join table is small relative to the other, a BROADCAST distribution method can be hinted using PQ_DISTRIBUTE hint. The optimizer considers the BROADCAST distribution method, but requires OPTIMIZER_FEATURES_ENABLE set to 9.0.2 or higher.

Now, assume that you have a join key with high cardinality, but one value contains most of the data, for example, lava lamp sales by year. The only year that had big sales was 1968, and the parallel execution server for the 1968 records is overwhelmed. You should use the same corrective actions as described in the previous paragraph.

The V$PQ_TQSTAT view provides a detailed report of message traffic at the table queue level. V$PQ_TQSTAT data is valid only when queried from a session that is executing parallel SQL statements. A table queue is the pipeline between query server groups, between the parallel execution coordinator and a query server group, or between a query server group and the coordinator. The table queues are represented explicitly in the operation column by PX SEND <partitioning type> (for example, PX SEND HASH) and PX RECEIVE.

V$PQ_TQSTAT has a row for each query server process that it reads from or writes to in each table queue. A table queue connecting 10 consumer processes to 10 producer processes has 20 rows in the view. Total the bytes column and group by TQ_ID, and the table queue identifier, to obtain the total number of bytes sent through each table queue. Compare this to the optimizer estimates; large variations might indicate a need to analyze the data using a larger sample.

Compute the variance of bytes grouped by TQ_ID. Large variances indicate workload imbalances. You should investigate large variances to determine whether the producers start out with unequal distributions of data, or whether the distribution itself is skewed. If the data itself is skewed, this might indicate a low cardinality, or low number of distinct values.

8.7.1.9 V$RSRC_CONS_GROUP_HISTORY

You can monitor parallel execution performance with the V$RSRC_CONS_GROUP_HISTORY dynamic performance view.

The V$RSRC_CONS_GROUP_HISTORY view displays a history of consumer group statistics for each entry in V$RSRC_PLAN_HISTORY that has a non-NULL plan, including information about parallel statement queuing.

8.7.1.10 V$RSRC_CONSUMER_GROUP

You can monitor parallel execution performance with the `V$RSRC_CONSUMER_GROUP` dynamic performance view.

The `V$RSRC_CONSUMER_GROUP` view displays data related to currently active resource consumer groups, including information about parallel statements.

8.7.1.11 V$RSRC_PLAN

You can monitor parallel execution performance with the `V$RSRC_PLAN` dynamic performance view.

The `V$RSRC_PLAN` view displays the names of all currently active resource plans, including the state of parallel statement queuing.

8.7.1.12 V$RSRC_PLAN_HISTORY

You can monitor parallel execution performance with the `V$RSRC_PLAN_HISTORY` dynamic performance view.

The `V$RSRC_PLAN_HISTORY` displays a history of when a resource plan was enabled, disabled, or modified on the instance. The history includes the state of parallel statement queuing

8.7.1.13 V$RSRC_SESSION_INFO

You can monitor parallel execution performance with the `V$RSRC_SESSION_INFO` dynamic performance view.

The `V$RSRC_SESSION_INFO` view displays resource manager statistics per session, including parallel statement queue statistics. Columns include `PQ_SERVERS` and `PQ_STATUS`.

The `PQ_SERVERS` column of the `V$RSRC_SESSION_INFO` view contains the number of active parallel servers if the session is active and running the parallel query. If the query is queued, the number of parallel servers that this query is trying to run with is shown.

The `PQ_STATUS` column maintains the reason that a parallel statement is queued

> **See Also:**
>
> *Oracle Database Reference* for information about the `V$RSRC_SESSION_INFO` view

8.7.1.14 V$RSRCMGRMETRIC

You can monitor parallel execution performance with the `V$RSRCMGRMETRIC` dynamic performance view.

The `V$RSRCMGRMETRIC` view displays statistics related to parallel statement queuing.

Statistics related to parallel statement queuing are added to the resource manager metrics that takes statistics for a given one-minute window and retains them for approximately one hour.

Columns include AVG_ACTIVE_PARALLEL_STMTS, AVG_QUEUED_PARALLEL_STMTS, AVG_ACTIVE_PARALLEL_SERVERS, AVG_QUEUED_PARALLEL_SERVERS, and PARALLEL_SERVERS_LIMIT.

> **See Also:**
>
> *Oracle Database Reference* for information about the V$RSRCMGRMETRIC view

8.7.2 Monitoring Session Statistics

You can monitor session statistics with the dynamic performance views to diagnose parallel execution performance.

Use GV$PX_SESSION to determine the configuration of the server group executing in parallel. In this example, session 9 is the query coordinator, while sessions 7 and 21 are in the first group, first set. Sessions 18 and 20 are in the first group, second set. The requested and granted DOP for this query is 2, as shown by the output from the following query:

```
SELECT QCSID, SID, INST_ID "Inst", SERVER_GROUP "Group", SERVER_SET "Set",
  DEGREE "Degree", REQ_DEGREE "Req Degree"
FROM GV$PX_SESSION ORDER BY QCSID, QCINST_ID, SERVER_GROUP, SERVER_SET;
```

Your output should resemble the following:

```
QCSID      SID        Inst       Group      Set        Degree     Req Degree
---------- ---------- ---------- ---------- ---------- ---------- ----------
         9          9          1
         9          7          1          1          1          2          2
         9         21          1          1          1          2          2
         9         18          1          1          2          2          2
         9         20          1          1          2          2          2
```

For a single instance, use SELECT FROM V$PX_SESSION and do not include the column name Instance ID.

The processes shown in the output from the previous example using GV$PX_SESSION collaborate to complete the same task. The next example shows the execution of a join query to determine the progress of these processes in terms of physical reads. Use this query to track any specific statistic:

```
SELECT QCSID, SID, INST_ID "Inst", SERVER_GROUP "Group", SERVER_SET "Set",
  NAME "Stat Name", VALUE
FROM GV$PX_SESSTAT A, V$STATNAME B
WHERE A.STATISTIC# = B.STATISTIC# AND NAME LIKE 'PHYSICAL READS'
  AND VALUE > 0 ORDER BY QCSID, QCINST_ID, SERVER_GROUP, SERVER_SET;
```

Your output should resemble the following:

```
QCSID  SID   Inst   Group  Set    Stat Name          VALUE
------ ----- ------ ------ ------ ------------------ ----------
     9     9      1                physical reads           3863
     9     7      1      1      1 physical reads              2
     9    21      1      1      1 physical reads              2
     9    18      1      1      2 physical reads              2
     9    20      1      1      2 physical reads              2
```

Use the previous type of query to track statistics in V$STATNAME. Repeat this query as often as required to observe the progress of the query server processes.

The next query uses V$PX_PROCESS to check the status of the query servers.

```
SELECT * FROM V$PX_PROCESS;
```

Your output should resemble the following:

```
SERV STATUS    PID    SPID      SID    SERIAL# IS_GV   CON_ID
---- --------- ------ --------- ------ ------- -----  -------
P002 IN USE        16     16955     21    7729  FALSE        0
P003 IN USE        17     16957     20    2921  FALSE        0
P004 AVAILABLE     18     16959                 FALSE        0
P005 AVAILABLE     19     16962                 FALSE        0
P000 IN USE        12      6999     18    4720  FALSE        0
P001 IN USE        13      7004      7     234  FALSE        0
```

> **See Also:**
>
> Monitoring Parallel Execution Performance with Dynamic Performance Views for descriptions of the dynamic performance views used in the examples

8.7.3 Monitoring System Statistics

You can monitor system statistics with the dynamic performance views to diagnose parallel execution performance.

The V$SYSSTAT and V$SESSTAT views contain several statistics for monitoring parallel execution. Use these statistics to track the number of parallel queries, DMLs, DDLs, data flow operators (DFOs), and operations. Each query, DML, or DDL can have multiple parallel operations and multiple DFOs.

In addition, statistics also count the number of query operations for which the DOP was reduced, or downgraded, due to either the adaptive multiuser algorithm or the depletion of available parallel execution servers.

Finally, statistics in these views also count the number of messages sent on behalf of parallel execution. The following syntax is an example of how to display these statistics:

```
SELECT NAME, VALUE FROM GV$SYSSTAT
  WHERE UPPER (NAME) LIKE '%PARALLEL OPERATIONS%'
  OR UPPER (NAME) LIKE '%PARALLELIZED%' OR UPPER (NAME) LIKE '%PX%';
```

Your output should resemble the following:

```
NAME                                                  VALUE
---------------------------------------------------- ----------
queries parallelized                                        347
DML statements parallelized                                   0
DDL statements parallelized                                   0
DFO trees parallelized                                      463
Parallel operations not downgraded                           28
Parallel operations downgraded to serial                     31
Parallel operations downgraded 75 to 99 pct                 252
Parallel operations downgraded 50 to 75 pct                 128
Parallel operations downgraded 25 to 50 pct                  43
Parallel operations downgraded 1 to 25 pct                   12
```

```
PX local messages sent                                74548
PX local messages recv'd                              74128
PX remote messages sent                                   0
PX remote messages recv'd                                 0
```

The following query shows the current wait state of each worker (child process) and query coordinator process on the system:

```
SELECT px.SID "SID", p.PID, p.SPID "SPID", px.INST_ID "Inst",
       px.SERVER_GROUP "Group", px.SERVER_SET "Set",
       px.DEGREE "Degree", px.REQ_DEGREE "Req Degree", w.event "Wait Event"
FROM GV$SESSION s, GV$PX_SESSION px, GV$PROCESS p, GV$SESSION_WAIT w
WHERE s.sid (+) = px.sid AND s.inst_id (+) = px.inst_id AND
      s.sid = w.sid (+) AND s.inst_id = w.inst_id (+) AND
      s.paddr = p.addr (+) AND s.inst_id = p.inst_id (+)
ORDER BY DECODE(px.QCINST_ID,  NULL, px.INST_ID,  px.QCINST_ID), px.QCSID,
DECODE(px.SERVER_GROUP, NULL, 0, px.SERVER_GROUP), px.SERVER_SET, px.INST_ID;
```

8.7.4 Monitoring Operating System Statistics

There is considerable overlap between information available in Oracle Database and information available though operating system utilities, such as `sar` and `vmstat` on UNIX-based systems.

Operating systems provide performance statistics on I/O, communication, CPU, memory and paging, scheduling, and synchronization primitives. The `V$SESSTAT` view provides the major categories of operating system statistics as well.

Typically, operating system information about I/O devices and semaphore operations is harder to map back to database objects and operations than is Oracle Database information. However, some operating systems have good visualization tools and efficient means of collecting the data.

Operating system information about CPU and memory usage is very important for assessing performance. Probably the most important statistic is CPU usage. The goal of low-level performance tuning is to become CPU bound on all CPUs. After this is achieved, you can work at the SQL level to find an alternate plan that might be more I/O intensive but use less CPU.

Operating system memory and paging information is valuable for fine tuning the many system parameters that control how memory is divided among memory-intensive data warehouse subsystems like parallel communication, sort, and hash join.

8.8 Tips for Tuning Parallel Execution

Various ideas for improving performance in a parallel execution environment are discussed under this section.

This section contains the following topics:

- Implementing a Parallel Execution Strategy
- Optimizing Performance by Creating and Populating Tables in Parallel
- Using EXPLAIN PLAN to Show Parallel Operations Plans
- Additional Considerations for Parallel DML
- Optimizing Performance by Creating Indexes in Parallel
- Parallel DML Tips

- Incremental Data Loading in Parallel

8.8.1 Implementing a Parallel Execution Strategy

Implementing a good parallel execution strategy is important to ensure high performance.

Recommendations for a good strategy include:

- Implement a simple setup to understand what is happening in your system.
- Use resource manager to specify the maximum degree of parallelism (DOP) for consumer groups so that each group is allotted a specific amount of processing resources without overwhelming the system. A resource management policy is needed when using parallel execution to keep the system under control, and to ensure SQL statements are able to execute in parallel.
- Base your strategy on the amount of system resources you want to make available for parallel execution. Adjust the values of the parameters `PARALLEL_MAX_SERVERS` and `PARALLEL_SERVERS_TARGET` to limit the number of parallel execution (PX) servers running in the system.
- Consider taking an ELT (Extract, Load, and Transform) strategy rather than an ETL (Extract, Transform, and Load) strategy.
- Use external tables with a parallel SQL statement, such as CTAS or IAS, for faster data loads

8.8.2 Optimizing Performance by Creating and Populating Tables in Parallel

To optimize parallel execution performance for queries that retrieve large result sets, create and populate tables in parallel.

Oracle Database cannot return results to a user process in parallel. If a query returns a large number of rows, execution of the query might indeed be faster. However, the user process can receive the rows only serially. To optimize parallel execution performance for queries that retrieve large result sets, use `PARALLEL CREATE TABLE AS SELECT` or direct-path `INSERT` to store the result set in the database. At a later time, users can view the result set serially.

Performing the `SELECT` in parallel does not influence the `CREATE` statement. If the `CREATE` statement is executed in parallel, however, the optimizer tries to make the `SELECT` run in parallel also.

When combined with the `NOLOGGING` option, the parallel version of `CREATE TABLE AS SELECT` provides a very efficient intermediate table facility, for example:

```
CREATE TABLE summary PARALLEL NOLOGGING AS SELECT dim_1, dim_2 ...,
SUM (meas_1)
FROM facts GROUP BY dim_1, dim_2;
```

These tables can also be incrementally loaded with parallel `INSERT`. You can take advantage of intermediate tables using the following techniques:

- Common subqueries can be computed once and referenced many times. This can allow some queries against star schemas (in particular, queries without selective `WHERE`-clause predicates) to be better parallelized. Star queries with selective `WHERE`-clause predicates using the star-transformation technique can be effectively parallelized automatically without any modification to the SQL.
- Decompose complex queries into simpler steps to provide application-level checkpoint or restart. For example, a complex multitable join on a one terabyte database could run for

dozens of hours. A failure during this query would mean starting over from the beginning. Using `CREATE TABLE AS SELECT` or `PARALLEL INSERT AS SELECT`, you can rewrite the query as a sequence of simpler queries that run for a few hours each. If a system failure occurs, the query can be restarted from the last completed step.

- Implement manual parallel delete operations efficiently by creating a new table that omits the unwanted rows from the original table, and then dropping the original table. Alternatively, you can use the convenient parallel delete feature, which directly deletes rows from the original table.
- Create summary tables for efficient multidimensional drill-down analysis. For example, a summary table might store the sum of revenue grouped by month, brand, region, and salesperson.
- Reorganize tables, eliminating chained rows, compressing free space, and so on, by copying the old table to a new table. This is much faster than export/import and easier than reloading.

Be sure to use the `DBMS_STATS` package to gather optimizer statistics on newly created tables. To avoid I/O bottlenecks, specify a tablespace that is striped across at least as many physical disks as CPUs. To avoid fragmentation in allocating space, the number of files in a tablespace should be a multiple of the number of CPUs.

> **See Also:**
>
> *Oracle Database Data Warehousing Guide* for information about parallel execution in data warehouses

8.8.3 Using EXPLAIN PLAN to Show Parallel Operations Plans

Use the `EXPLAIN PLAN` statement to see the execution plans for parallel queries.

The `EXPLAIN PLAN` output shows optimizer information in the `COST`, `BYTES`, and `CARDINALITY` columns. You can also use the `utlxplp.sql` script to present the `EXPLAIN PLAN` output with all relevant parallel information.

There are several ways to optimize the parallel execution of join statements. You can alter system configuration, adjust parameters as discussed earlier in this chapter, or use hints, such as the `DISTRIBUTION` hint.

The key points when using `EXPLAIN PLAN` are to:

- Verify optimizer selectivity estimates. If the optimizer thinks that only one row is produced from a query, it tends to favor using a nested loop. This could be an indication that the tables are not analyzed or that the optimizer has made an incorrect estimate about the correlation of multiple predicates on the same table. Extended statistics or a hint may be required to provide the optimizer with the correct selectivity or to force the optimizer to use another join method.
- Use hash join on low cardinality join keys. If a join key has few distinct values, then a hash join may not be optimal. If the number of distinct values is less than the degree of parallelism (DOP), then some parallel query servers may be unable to work on the particular query.
- Consider data skew. If a join key involves excessive data skew, a hash join may require some parallel query servers to work more than others. Consider using a hint to cause a `BROADCAST` distribution method if the optimizer did not choose it. The optimizer considers

the BROADCAST distribution method only if the OPTIMIZER_FEATURES_ENABLE is set to 9.0.2 or higher. See V$PQ_TQSTAT for more information.

8.8.3.1 Example: Using EXPLAIN PLAN to Show Parallel Operations

You can use EXPLAIN PLAN to show parallel operations.

The following example illustrates how the optimizer intends to execute a parallel query:

```
explain plan for
  SELECT /*+ PARALLEL */ cust_first_name, cust_last_name
   FROM customers c, sales s WHERE c.cust_id = s.cust_id;

-----------------------------------------------------------
| Id  | Operation                         |  Name          |
-----------------------------------------------------------
|   0 | SELECT STATEMENT                  |                |
|   1 |  PX COORDINATOR                   |                |
|   2 |   PX SEND QC (RANDOM)             | :TQ10000       |
|   3 |    NESTED LOOPS                   |                |
|   4 |     PX BLOCK ITERATOR             |                |
|   5 |      TABLE ACCESS FULL            | CUSTOMERS      |
|   6 |     PARTITION RANGE ALL           |                |
|   7 |      BITMAP CONVERSION TO ROWIDS|                |
|   8 |       BITMAP INDEX SINGLE VALUE | SALES_CUST_BIX |
-----------------------------------------------------------

Note
-----
   - automatic DOP: Computed Degree of Parallelism is 2
```

8.8.4 Additional Considerations for Parallel DML

Additional considerations when using parallel DML operations are introduced in this topic.

When you want to refresh your data warehouse database using parallel insert, update, or delete operations on a data warehouse, there are additional issues to consider when designing the physical database. These considerations do not affect parallel execution operations. These issues are:

- Parallel DML and Direct-Path Restrictions
- Limitation on the Degree of Parallelism
- When to Increase INITRANS
- Limitation on Available Number of Transaction Free Lists for Segments
- Multiple Archivers for Large Numbers of Redo Logs
- Database Writer Process (DBWn) Workload
- [NO]LOGGING Clause

8.8.4.1 Parallel DML and Direct-Path Restrictions

The restrictions for parallel DML and direct-path operations are identified in this topic.

If a parallel restriction is violated, then the operation is simply performed serially. If a direct-path INSERT restriction is violated, then the APPEND hint is ignored and a conventional insert operation is performed. No error message is returned.

8.8.4.2 Limitation on the Degree of Parallelism

There are certain limitations on the degree of parallelism based on the software level of Oracle Database in use.

For tables that do not have the parallel DML `itl` invariant property (tables created before Oracle9*i* Release 2 (9.2) or tables that were created with the `COMPATIBLE` initialization parameter set to less than `9.2`), the degree of parallelism (DOP) equals the number of partitions or subpartitions. That means that, if the table is not partitioned, the query runs serially. To determine which tables do not have this property, issue the following statement:

```
SELECT u.name, o.name FROM obj$ o, tab$ t, user$ u
 WHERE o.obj# = t.obj# AND o.owner# = u.user#
 AND bitand(t.property,536870912) != 536870912;
```

> **See Also:**
>
> *Oracle Database Concepts* for information about the interested transaction list (ITL), also called the transaction table

8.8.4.3 When to Increase INITRANS

You should increase the value of `INITRANS` under certain situations.

If you have global indexes, a global index segment and global index blocks are shared by server processes of the same parallel DML statement. Even if the operations are not performed against the same row, the server processes can share the same index blocks. Each server transaction needs one transaction entry in the index block header before it can make changes to a block.

In this situation, when using the `CREATE INDEX` or `ALTER INDEX` statements, you should set `INITRANS`, the initial number of transactions allocated within each data block, to a large value, such as the maximum DOP against this index.

8.8.4.4 Limitation on Available Number of Transaction Free Lists for Segments

There is a limitation on the available number of transaction free lists for segments in dictionary-managed tablespaces.

After a segment has been created, the number of process and transaction free lists is fixed and cannot be altered. If you specify a large number of process free lists in the segment header, you might find that this limits the number of transaction free lists that are available. You can abate this limitation the next time you re-create the segment header by decreasing the number of process free lists; this leaves more room for transaction free lists in the segment header.

For `UPDATE` and `DELETE` operations, each server process can require its own transaction free list. The parallel DML DOP is thus effectively limited by the smallest number of transaction free lists available on the table and on any of the global indexes the DML statement must maintain. For example, if the table has 25 transaction free lists and the table has two global indexes, one with 50 transaction free lists and one with 30 transaction free lists, the DOP is limited to 25. If the table had 40 transaction free lists, the DOP would have been limited to 30.

The `FREELISTS` parameter of the `STORAGE` clause is used to set the number of process free lists. By default, no process free lists are created.

The default number of transaction free lists depends on the block size. For example, if the number of process free lists is not set explicitly, a 4 KB block has about 80 transaction free lists by default. The minimum number of transaction free lists is 25.

8.8.4.5 Multiple Archivers for Large Numbers of Redo Logs

Multiple archiver processes are needed for archiving large numbers of redo logs.

Parallel DDL and parallel DML operations can generate a large number of redo logs. A single `ARCH` process to archive these redo logs might not be able to keep up. To avoid this problem, you can spawn multiple archiver processes manually or by using a job queue.

8.8.4.6 Database Writer Process (DBWn) Workload

There are situations when you should increase the number of database writer processes.

Parallel DML operations use a large number of data, index, and undo blocks in the buffer cache during a short interval. For example, suppose you see a high number of `free_buffer_waits` after querying the `V$SYSTEM_EVENT` view, as in the following syntax:

```
SELECT TOTAL_WAITS FROM V$SYSTEM_EVENT WHERE EVENT = 'FREE BUFFER WAITS';
```

In this case, you should consider increasing the DBW*n* processes. If there are no waits for free buffers, the query does not return any rows.

8.8.4.7 [NO]LOGGING Clause

Understand the considerations when setting the `[NO]LOGGING` clause.

The `[NO]LOGGING` clause applies to tables, partitions, tablespaces, and indexes. Virtually no log is generated for certain operations (such as direct-path `INSERT`) if the `NOLOGGING` clause is used. The `NOLOGGING` attribute is not specified at the `INSERT` statement level but is instead specified when using the `ALTER` or `CREATE` statement for a table, partition, index, or tablespace.

When a table or index has `NOLOGGING` set, neither parallel nor serial direct-path `INSERT` operations generate redo logs. Processes running with the `NOLOGGING` option set run faster because no redo is generated. However, after a `NOLOGGING` operation against a table, partition, or index, if a media failure occurs before a backup is performed, then all tables, partitions, and indexes that have been modified might be corrupted.

Direct-path `INSERT` operations (except for dictionary updates) never generate redo logs if the `NOLOGGING` clause is used. The `NOLOGGING` attribute does not affect undo, only redo. To be precise, `NOLOGGING` allows the direct-path `INSERT` operation to generate a negligible amount of redo (range-invalidation redo, as opposed to full image redo).

For backward compatibility, `[UN]RECOVERABLE` is still supported as an alternate keyword with the `CREATE TABLE` statement. This alternate keyword might not be supported, however, in future releases.

At the tablespace level, the logging clause specifies the default logging attribute for all tables, indexes, and partitions created in the tablespace. When an existing tablespace logging attribute is changed by the `ALTER TABLESPACE` statement, then all tables, indexes, and partitions created after the `ALTER` statement have the new logging attribute; existing ones do not change their logging attributes. The tablespace-level logging attribute can be overridden by the specifications at the table, index, or partition level.

The default logging attribute is LOGGING. However, if you have put the database in NOARCHIVELOG mode, by issuing ALTER DATABASE NOARCHIVELOG, then all operations that can be done without logging do not generate logs, regardless of the specified logging attribute.

8.8.5 Optimizing Performance by Creating Indexes in Parallel

You can optimize performance by creating indexes in parallel.

Multiple processes can work simultaneously to create an index. By dividing the work necessary to create an index among multiple server processes, Oracle Database can create the index more quickly than if a single server process created the index serially.

Parallel index creation works in much the same way as a table scan with an ORDER BY clause. The table is randomly sampled and a set of index keys is found that equally divides the index into the same number of pieces as the DOP. A first set of query processes scans the table, extracts key-rowid pairs, and sends each pair to a process in a second set of query processes based on a key. Each process in the second set sorts the keys and builds an index in the usual fashion. After all index pieces are built, the parallel execution coordinator simply concatenates the pieces (which are ordered) to form the final index.

Parallel local index creation uses a single server set. Each server process in the set is assigned a table partition to scan and for which to build an index partition. Because half as many server processes are used for a given DOP, parallel local index creation can be run with a higher DOP. However, the DOP is restricted to be less than or equal to the number of index partitions you want to create. To avoid this limitation, you can use the DBMS_PCLXUTIL package.

You can optionally specify that no redo and undo logging should occur during index creation. This can significantly improve performance but temporarily renders the index unrecoverable. Recoverability is restored after the new index is backed up. If your application can tolerate a window where recovery of the index requires it to be re-created, then you should consider using the NOLOGGING clause.

The PARALLEL clause in the CREATE INDEX statement is the only way in which you can specify the DOP for creating the index. If the DOP is not specified in the parallel clause of the CREATE INDEX statement, then the number of CPUs is used as the DOP. If there is no PARALLEL clause, index creation is done serially.

When creating an index in parallel, the STORAGE clause refers to the storage of each of the subindexes created by the query server processes. Therefore, an index created with an INITIAL value of 5 MB and a DOP of 12 consumes at least 60 MB of storage during index creation because each process starts with an extent of 5 MB. When the query coordinator process combines the sorted subindexes, some extents might be trimmed, and the resulting index might be smaller than the requested 60 MB.

When you add or enable a UNIQUE or PRIMARY KEY constraint on a table, you cannot automatically create the required index in parallel. Instead, manually create an index on the desired columns, using the CREATE INDEX statement and an appropriate PARALLEL clause, and then add or enable the constraint. Oracle Database then uses the existing index when enabling or adding the constraint.

Multiple constraints on the same table can be enabled concurrently and in parallel if all the constraints are in the ENABLE NOVALIDATE state. In the following example, the ALTER TABLE ENABLE CONSTRAINT statement performs the table scan that checks the constraint in parallel:

```
CREATE TABLE a (a1 NUMBER CONSTRAINT ach CHECK (a1 > 0) ENABLE NOVALIDATE)
PARALLEL;
INSERT INTO a values (1);
```

```
COMMIT;
ALTER TABLE a ENABLE CONSTRAINT ach;
```

8.8.6 Parallel DML Tips

The tips for parallel DML functionality are introduced in this topic.

The topics covered include:

- Parallel DML Tip 1: INSERT
- Parallel DML Tip 2: Direct-Path INSERT
- Parallel DML Tip 3: Parallelizing INSERT, MERGE, UPDATE, and DELETE

> **See Also:**
>
> - *Oracle Database Administrator's Guide* for information about improving load performance with direct-path insert
> - *Oracle Database SQL Language Reference* for information about the `INSERT` statement

8.8.6.1 Parallel DML Tip 1: INSERT

Parallel DML when using the SQL `INSERT` statement is discussed in this topic.

The functionality available using an `INSERT` statement can be summarized as shown in Table 8-5:

Table 8-5 Summary of INSERT Features

Insert Type	Parallel	Serial	NOLOGGING
Conventional	No See text in this section for information about using the `NOAPPEND` hint with parallel DML enabled to perform a parallel conventional insert.	Yes	No
Direct-path `INSERT` (`APPEND`)	Yes, but requires `ALTER SESSION ENABLE PARALLEL DML` or the `ENABLE_PARALLEL_DML` SQL hint to enable `PARALLEL DML` mode and one of the following: • Table `PARALLEL` attribute or `PARALLEL` hint to explicitly set parallelism • `APPEND` hint to explicitly set mode Or the following `ALTER SESSION FORCE PARALLEL DML` to force `PARALLEL DML` mode	Yes, but requires: `APPEND` hint	Yes, but requires: `NOLOGGING` attribute set for partition or table

If parallel DML is enabled and there is a `PARALLEL` hint or `PARALLEL` attribute set for the table in the data dictionary, then insert operations are parallel and appended, unless a restriction applies. If either the `PARALLEL` hint or `PARALLEL` attribute is missing, the insert operation is

performed serially. Automatic DOP only parallelizes the DML part of a SQL statement if and only if parallel DML is enabled or forced.

If parallel DML is enabled, then you can use the NOAPPEND hint to perform a parallel conventional insert operation. For example, you can use /*+ noappend parallel */ with the SQL INSERT statement to perform a parallel conventional insert.

```
SQL> INSERT /*+ NOAPPEND PARALLEL */ INTO sales_hist SELECT * FROM sales;
```

The advantage of the parallel conventional insert operation is the ability to perform online operations with none of the restrictions of direct-path INSERT. The disadvantage of the parallel conventional insert operation is that this process may be slower than direct-path INSERT.

8.8.6.2 Parallel DML Tip 2: Direct-Path INSERT

Parallel DML when using Direct-Path INSERT operations is discussed in this topic.

The append mode is the default during a parallel insert operation. Data is always inserted into a new block, which is allocated to the table. Using the APPEND hint is optional. You should use append mode to increase the speed of INSERT operations, but not when space utilization must be optimized. You can use NOAPPEND to override append mode.

The APPEND hint applies to both serial and parallel insert operation. Serial insertions are also faster if you use this hint. The APPEND hint, however, does require more space and locking overhead.

You can use NOLOGGING with APPEND to make the process even faster. NOLOGGING means that no redo log is generated for the operation. NOLOGGING is never the default; use it when you want to optimize performance. It should not typically be used when recovery is needed for the table or partition. If recovery is needed, be sure to perform a backup immediately after the operation. Use the ALTER TABLE [NO]LOGGING statement to set the appropriate value.

8.8.6.3 Parallel DML Tip 3: Parallelizing INSERT, MERGE, UPDATE, and DELETE

Parallel DML when using insert, merge, update, and delete operations is discussed in this topic.

When the table or partition has the PARALLEL attribute in the data dictionary, that attribute setting is used to determine parallelism of INSERT, UPDATE, and DELETE statements and queries. An explicit PARALLEL hint for a table in a statement overrides the effect of the PARALLEL attribute in the data dictionary.

You can use the NO_PARALLEL hint to override a PARALLEL attribute for the table in the data dictionary. In general, hints take precedence over attributes.

DML operations are considered for parallelization if the session has been enabled in the PARALLEL DML mode with the ALTER SESSION ENABLE PARALLEL DML statement or a specific SQL statement has been enabled in the PARALLEL DML mode with the ENABLE_PARALLEL_DML hint. The mode does not affect parallelization of queries or of the query portions of a DML statement.

8.8.6.3.1 Parallelizing INSERT SELECT

In the INSERT ... SELECT statement, you can specify a PARALLEL hint after the INSERT keyword, in addition to the hint after the SELECT keyword.

The PARALLEL hint after the INSERT keyword applies to the INSERT operation only, and the PARALLEL hint after the SELECT keyword applies to the SELECT operation only. Thus, parallelism of the INSERT and SELECT operations are independent of each other. If one operation cannot be performed in parallel, it has no effect on whether the other operation can be performed in parallel.

The ability to parallelize insert operations causes a change in existing behavior if the user has explicitly enabled the session for parallel DML and if the table in question has a PARALLEL attribute set in the data dictionary entry. In that case, existing INSERT SELECT statements that have the select operation parallelized can also have their insert operation parallelized.

If you query multiple tables, you can specify multiple SELECT PARALLEL hints and multiple PARALLEL attributes.

Example 8-5 shows the addition of the new employees who were hired after the acquisition of ACME.

Example 8-5 Parallelizing INSERT SELECT

```
INSERT /*+ PARALLEL(employees) */ INTO employees
SELECT /*+ PARALLEL(ACME_EMP) */ *  FROM ACME_EMP;
```

The APPEND keyword is not required in this example because it is implied by the PARALLEL hint.

8.8.6.3.2 Parallelizing UPDATE and DELETE

The PARALLEL hint (placed immediately after the UPDATE or DELETE keyword) applies not only to the underlying scan operation, but also to the UPDATE or DELETE operation.

Alternatively, you can specify UPDATE or DELETE parallelism in the PARALLEL clause specified in the definition of the table to be modified.

If you have explicitly enabled parallel DML for the session or transaction, UPDATE or DELETE statements that have their query operation parallelized can also have their UPDATE or DELETE operation parallelized. Any subqueries or updatable views in the statement can have their own separate PARALLEL hints or clauses, but these parallel directives do not affect the decision to parallelize the update or delete. If these operations cannot be performed in parallel, it has no effect on whether the UPDATE or DELETE portion can be performed in parallel.

Example 8-6 shows the update operation to give a 10 percent salary raise to all clerks in Dallas.

Example 8-6 Parallelizing UPDATE and DELETE

```
UPDATE /*+ PARALLEL(employees) */ employees
 SET salary=salary * 1.1 WHERE job_id='CLERK' AND department_id IN
  (SELECT department_id FROM DEPARTMENTS WHERE location_id = 'DALLAS');
```

The PARALLEL hint is applied to the UPDATE operation and to the scan.

Example 8-7 shows the removal of all products of category 39 because that business line was recently spun off into a separate company.

Example 8-7 Parallelizing UPDATE and DELETE

```
DELETE /*+ PARALLEL(PRODUCTS) */ FROM PRODUCTS
  WHERE category_id = 39;
```

Again, the parallelism is applied to the scan and `UPDATE` operations on the table `employees`.

8.8.7 Incremental Data Loading in Parallel

Parallel DML combined with the updatable join views facility provides an efficient solution for refreshing the tables of a data warehouse system.

To refresh tables is to update them with the differential data generated from the OLTP production system.

In the following example, assume a refresh of a table named `customers` that has columns `c_key`, `c_name`, and `c_addr`. The differential data contains either new rows or rows that have been updated since the last refresh of the data warehouse. In this example, the updated data is shipped from the production system to the data warehouse system by means of ASCII files. These files must be loaded into a temporary table, named `diff_customer`, before starting the refresh process. You can use SQL*Loader with both the parallel and direct options to efficiently perform this task. You can use the `APPEND` hint when loading in parallel as well.

After `diff_customer` is loaded, the refresh process can be started. It can be performed in two phases or by merging in parallel, as demonstrated in the following:

- Optimizing Performance for Updating the Table in Parallel
- Efficiently Inserting the New Rows into the Table in Parallel
- Optimizing Performance by Merging in Parallel

8.8.7.1 Optimizing Performance for Updating the Table in Parallel

How to optimize performance for updating a table in parallel is discussed in this topic.

The following statement is a straightforward SQL implementation of the update using subqueries:

```
UPDATE customers SET(c_name, c_addr) = (SELECT c_name, c_addr
  FROM diff_customer WHERE diff_customer.c_key = customer.c_key)
  WHERE c_key IN(SELECT c_key FROM diff_customer);
```

Unfortunately, the two subqueries in this statement affect performance.

An alternative is to rewrite this query using updatable join views. To rewrite the query, you must first add a primary key constraint to the `diff_customer` table to ensure that the modified columns map to a key-preserved table:

```
CREATE UNIQUE INDEX diff_pkey_ind ON diff_customer(c_key) PARALLEL NOLOGGING;

ALTER TABLE diff_customer ADD PRIMARY KEY (c_key);
```

You can then update the `customers` table with the following SQL statement:

```
UPDATE /*+ PARALLEL(cust_joinview) */
  (SELECT /*+ PARALLEL(customers) PARALLEL(diff_customer) */
  CUSTOMER.c_name AS c_name CUSTOMER.c_addr AS c_addr,
   diff_customer.c_name AS c_newname, diff_customer.c_addr AS c_newaddr
   FROM diff_customer
```

```
WHERE customers.c_key = diff_customer.c_key) cust_joinview
SET c_name = c_newname, c_addr = c_newaddr;
```

The underlying scans feeding the join view `cust_joinview` are done in parallel. You can then parallelize the update to further improve performance, but only if the `customers` table is partitioned.

8.8.7.2 Efficiently Inserting the New Rows into the Table in Parallel

How to efficiently insert new rows into a table in parallel is discussed in this topic.

The last phase of the refresh process consists of inserting the new rows from the `diff_customer` temporary table to the `customers` table. Unlike the update case, you cannot avoid having a subquery in the `INSERT` statement:

```
INSERT /*+PARALLEL(customers)*/ INTO customers SELECT * FROM diff_customer s);
```

However, you can guarantee that the subquery is transformed into an anti-hash join by using the `HASH_AJ` hint. Doing so enables you to use parallel `INSERT` to execute the preceding statement efficiently. Parallel `INSERT` is applicable even if the table is not partitioned.

8.8.7.3 Optimizing Performance by Merging in Parallel

How to optimize performance by merging in parallel is discussed in this topic.

You can combine update and insert operations into one statement, commonly known as a **merge**, as shown in the following example.

```
MERGE INTO customers USING diff_customer
ON (diff_customer.c_key = customer.c_key) WHEN MATCHED THEN
  UPDATE SET (c_name, c_addr) = (SELECT c_name, c_addr
  FROM diff_customer WHERE diff_customer.c_key = customers.c_key)
WHEN NOT MATCHED THEN
 INSERT VALUES (diff_customer.c_key,diff_customer.c_data);
```

The SQL statement in the previous example achieves the same result as all of the statements in Optimizing Performance for Updating the Table in Parallel and Efficiently Inserting the New Rows into the Table in Parallel.

9

Backing Up and Recovering VLDBs

Backup and recovery is a crucial and important job for a DBA to protect business data.

As data storage grows larger each year, DBAs are continually challenged to ensure that critical data is backed up and that it can be recovered quickly and easily to meet business needs. Very large databases are unique in that they are large and data may come from many resources. OLTP and data warehouse systems have some distinct characteristics. Generally, the availability considerations for a very large OLTP system are no different from the considerations for a small OLTP system. Assuming a fixed allowed downtime, a large OLTP system requires more hardware resources than a small OLTP system.

This chapter proposes an efficient backup and recovery strategy for very large databases to reduce the overall resources necessary to support backup and recovery by using some special characteristics that differentiate data warehouses from OLTP systems.

This chapter contains the following sections:

- Data Warehouses
- Oracle Backup and Recovery
- Data Warehouse Backup and Recovery
- The Data Warehouse Recovery Methodology

9.1 Data Warehouses

A data warehouse is a system that is designed to support analysis and decision-making.

In a typical enterprise, hundreds or thousands of users may rely on the data warehouse to provide the information to help them understand their business and make better decisions. Therefore, availability is a key requirement for data warehousing. This chapter discusses one key aspect of data warehouse availability: the recovery of data after a data loss.

Before looking at the backup and recovery techniques in detail, it is important to discuss specific techniques for backup and recovery of a data warehouse. In particular, one legitimate question might be: Should a data warehouse backup and recovery strategy be just like that of every other database system?

A DBA should initially approach the task of data warehouse backup and recovery by applying the same techniques that are used in OLTP systems: the DBA must decide what information to protect and quickly recover when media recovery is required, prioritizing data according to its importance and the degree to which it changes. However, the issue that commonly arises for data warehouses is that an approach that is efficient and cost-effective for a 100 GB OLTP system may not be viable for a 10 TB data warehouse. The backup and recovery may take 100 times longer or require 100 times more storage.

> **See Also:**
>
> *Oracle Database Data Warehousing Guide* for more information about data warehouses

9.1.1 Data Warehouse Characteristics

There are several key differences between data warehouses and OLTP systems that have significant impacts on backup and recovery.

These differences are:

1. A data warehouse is typically much larger than an OLTP system. Data warehouses over 10's of terabytes are not uncommon and the largest data warehouses grow to orders of magnitude larger. Thus, scalability is a particularly important consideration for data warehouse backup and recovery.
2. A data warehouse often has lower availability requirements than an OLTP system. While data warehouses are critical to businesses, there is also a significant cost associated with the ability to recover multiple terabytes in a few hours compared to recovering in a day. Some organizations may determine that in the unlikely event of a failure requiring the recovery of a significant portion of the data warehouse, they may tolerate an outage of a day or more if they can save significant expenditures in backup hardware and storage.
3. A data warehouse is typically updated through a controlled process called the ETL (Extract, Transform, Load) process, unlike in OLTP systems where users are modifying data themselves. Because the data modifications are done in a controlled process, the updates to a data warehouse are often known and reproducible from sources other than redo logs.
4. A data warehouse contains historical information, and often, significant portions of the older data in a data warehouse are static. For example, a data warehouse may track five years of historical sales data. While the most recent year of data may still be subject to modifications (due to returns, restatements, and so on), the last four years of data may be entirely static. The advantage of static data is that it does not need to be backed up frequently.

These four characteristics are key considerations when devising a backup and recovery strategy that is optimized for data warehouses.

9.2 Oracle Backup and Recovery

In general, backup and recovery refers to the various strategies and procedures involved in protecting your database against data loss and reconstructing the database after any kind of data loss.

A backup is a representative copy of data. This copy can include important parts of a database such as the control file, archived redo logs, and data files. A backup protects data from application error and acts as a safeguard against unexpected data loss, by providing a way to restore original data.

This section contains the following topics:

- Physical Database Structures Used in Recovering Data
- Backup Type

- Backup Tools

9.2.1 Physical Database Structures Used in Recovering Data

Before you begin to think seriously about a backup and recovery strategy, the physical data structures relevant for backup and recovery operations must be identified.

These components include the files and other structures that constitute data for an Oracle data store and safeguard the data store against possible failures. Three basic components are required for the recovery of Oracle Database:

- Data files
- Redo Logs
- Control Files

9.2.1.1 Data files

Oracle Database consists of one or more logical storage units called tablespaces. Each tablespace in Oracle Database consists of one or more files called data files, which are physical files located on or attached to the host operating system in which Oracle Database is running.

The data in a database is collectively stored in the data files that constitute each tablespace of the database. The simplest Oracle Database would have one tablespace, stored in one data file. Copies of the data files of a database are a critical part of any backup strategy. The sheer size of the data files is the main challenge from a VLDB backup and recovery perspective.

9.2.1.2 Redo Logs

Redo logs record all changes made to a database's data files.

With a complete set of redo logs and an older copy of a data file, Oracle can reapply the changes recorded in the redo logs to re-create the database at any point between the backup time and the end of the last redo log. Each time data is changed in Oracle Database, that change is recorded in the online redo log first, before it is applied to the data files.

Oracle Database requires at least two online redo log groups. In each group, there is at least one online redo log member, an individual redo log file where the changes are recorded. At intervals, Oracle Database rotates through the online redo log groups, storing changes in the current online redo log while the groups not in use can be copied to an archive location, where they are called archived redo logs (or, collectively, the archived redo log). For high availability reasons, production systems should always use multiple online redo members per group, preferably on different storage systems. Preserving the archived redo log is a major part of your backup strategy, as it contains a record of all updates to data files. Backup strategies often involve copying the archived redo logs to disk or tape for longer-term storage.

9.2.1.3 Control Files

The control file contains a crucial record of the physical structures of the database and their status.

Several types of information stored in the control file are related to backup and recovery:

- Database information required to recover from failures or to perform media recovery
- Database structure information, such as data file details

- Redo log details
- Archived log records
- A record of past RMAN backups

Oracle Database data file recovery process is in part guided by status information in the control file, such as the database checkpoints, current online redo log file, and the data file header checkpoints. Loss of the control file makes recovery from a data loss much more difficult. The control file should be backed up regularly, to preserve the latest database structural changes, and to simplify recovery.

9.2.2 Backup Type

Backups are divided into physical backups and logical backups.

- **Physical backups** are backups of the physical files used in storing and recovering your database, such as data files, control files, and archived redo logs. Ultimately, every physical backup is a copy of files storing database information to some other location, whether on disk or offline storage, such as tape.
- **Logical backups** contain logical data (for example, tables or stored procedures) extracted from a database with Oracle Data Pump (export/import) utilities. The data is stored in a binary file that can be imported into Oracle Database.

Physical backups are the foundation of any backup and recovery strategy. Logical backups are a useful supplement to physical backups in many circumstances but are not sufficient protection against data loss without physical backups.

Reconstructing the contents of all or part of a database from a backup typically involves two phases: retrieving a copy of the data file from a backup, and reapplying changes to the file since the backup, from the archived and online redo logs, to bring the database to the desired recovery point in time. To restore a data file or control file from backup is to retrieve the file from the backup location on tape, disk, or other media, and make it available to Oracle Database. To recover a data file, is to take a restored copy of the data file and apply to it the changes recorded in the database's redo logs. To recover a whole database is to perform recovery on each of its data files.

9.2.3 Backup Tools

Oracle Database provides several tools to manage backup and recovery of Oracle Databases.

Each tool gives you a choice of several basic methods for making backups. The methods include:

- Oracle Recovery Manager (RMAN)

  RMAN reduces the administration work associated with your backup strategy by maintaining an extensive record of metadata about all backups and needed recovery-related files. In restore and recovery operations, RMAN uses this information to eliminate the need for the user to identify needed files. RMAN is efficient, supporting file multiplexing and parallel streaming, and verifies blocks for physical and (optionally) logical corruptions, on backup and restore.

  Backup activity reports can be generated using `V$BACKUP` views.

- Oracle Data Pump

  Oracle Data Pump provides high speed, parallel, bulk data and metadata movement of Oracle Database contents. This utility makes logical backups by writing data from Oracle Database to operating system files. This data can later be imported into Oracle Database.

- User-Managed Backups

  The database is backed up manually by executing commands specific to your operating system.

9.2.3.1 Oracle Recovery Manager (RMAN)

Oracle Recovery Manager (RMAN), a command-line is the Oracle-preferred method for efficiently backing up and recovering Oracle Database.

RMAN is designed to work intimately with the server, providing block-level corruption detection during backup and recovery. RMAN optimizes performance and space consumption during backup with file multiplexing and backup set compression, and integrates with leading tape and storage media products with the supplied Media Management Library (MML) API.

RMAN takes care of all underlying database procedures before and after backup or recovery, freeing dependency on operating system and SQL*Plus scripts. It provides a common interface for backup tasks across different host operating systems, and offers features not available through user-managed methods, such as data file and tablespace-level backup and recovery, parallelization of backup and recovery data streams, incremental backups, automatic backup of the control file on database structural changes, backup retention policy, and detailed history of all backups.

> **See Also:**
>
> *Oracle Database Backup and Recovery User's Guide* for more information about RMAN

9.2.3.2 Oracle Data Pump

Physical backups can be supplemented by using the Oracle Data Pump (export/import) utilities to make logical backups of data.

Logical backups store information about the schema objects created for a database. Oracle Data Pump loads data and metadata into a set of operating system files that can be imported on the same system or moved to another system and imported there.

The dump file set is made up of one or more disk files that contain table data, database object metadata, and control information. The files are written in a binary format. During an import operation, the Data Pump Import utility uses these files to locate each database object in the dump file set.

9.2.3.3 User-Managed Backups

If you do not want to use Recovery Manager, you can use operating system commands, such as the UNIX `dd` or `tar` commands, to make backups.

To create a user-managed online backup, the database must manually be placed into hot backup mode. Hot backup mode causes additional write operations to the online log files, increasing their size.

Backup operations can also be automated by writing scripts. You can make a backup of the entire database immediately, or back up individual tablespaces, data files, control files, or archived logs. An entire database backup can be supplemented with backups of individual tablespaces, data files, control files, and archived logs.

Operating system commands or third-party backup software can perform database backups. Conversely, the third-party software must be used to restore the backups of the database.

9.3 Data Warehouse Backup and Recovery

Data warehouse recovery is similar to that of an OLTP system.

However, a data warehouse may not require all of the data to be recovered from a backup, or for a complete failure, restoring the entire database before user access can commence. An efficient and fast recovery of a data warehouse begins with a well-planned backup.

The next several sections help you to identify what data should be backed up and guide you to the method and tools that enable you to recover critical data in the shortest amount of time.

This section contains the following topics:

- Recovery Time Objective (RTO)
- Recovery Point Objective (RPO)

9.3.1 Recovery Time Objective (RTO)

A **Recovery Time Objective** (RTO) is the time duration in which you want to be able to recover your data.

Your backup and recovery plan should be designed to meet RTOs your company chooses for its data warehouse. For example, you may determine that 5% of the data must be available within 12 hours, 50% of the data must be available after a complete loss of the database within 2 days, and the remainder of the data be available within 5 days. In this case you have two RTOs. Your total RTO is 7.5 days.

To determine what your RTO should be, you must first identify the impact of the data not being available. To establish an RTO, follow these four steps:

1. Analyze and identify: Understand your recovery readiness, risk areas, and the business costs of unavailable data. In a data warehouse, you should identify critical data that must be recovered in the *n* days after an outage.
2. Design: Transform the recovery requirements into backup and recovery strategies. This can be accomplished by organizing the data into logical relationships and criticality.
3. Build and integrate: Deploy and integrate the solution into your environment to back up and recover your data. Document the backup and recovery plan.
4. Manage and evolve: Test your recovery plans at regular intervals. Implement change management processes to refine and update the solution as your data, IT infrastructure, and business processes change.

9.3.2 Recovery Point Objective (RPO)

A Recovery Point Objective, or RPO, is the maximum amount of data that can be lost before causing detrimental harm to the organization.

RPO indicates the data loss tolerance of a business process or an organization in general. This data loss is often measured in terms of time, for example, 5 hours or 2 days worth of data loss. A zero RPO means that no committed data should be lost when media loss occurs, while a 24 hour RPO can tolerate a day's worth of data loss.

This section contains the following topics:

- More Data Means a Longer Backup Window
- Divide and Conquer

9.3.2.1 More Data Means a Longer Backup Window

The most obvious characteristic of the data warehouse is the size of the database.

This can be upward of 100's of terabytes. Hardware is the limiting factor to a fast backup and recovery. However, today's tape storage continues to evolve to accommodate the amount of data that must be offloaded to tape (for example, advent of Virtual Tape Libraries which use disks internally with the standard tape access interface). RMAN can fully use, in parallel, all available tape devices to maximize backup and recovery performance.

Essentially, the time required to back up a large database can be derived from the minimum throughput among: production disk, host bus adapter (HBA) and network to tape devices, and tape drive streaming specifications * the number of tape drives. The host CPU can also be a limiting factor to overall backup performance, if RMAN backup encryption or compression is used. Backup and recovery windows can be adjusted to fit any business requirements, given adequate hardware resources.

9.3.2.2 Divide and Conquer

In a data warehouse, there may be times when the database is not being fully used.

While this window of time may be several contiguous hours, it is not enough to back up the entire database. You may want to consider breaking up the database backup over several days. RMAN enables you to specify how long a given backup job is allowed to run. When using `BACKUP DURATION`, you can choose between running the backup to completion as quickly as possible and running it more slowly to minimize the load the backup may impose on your database.

In the following example, RMAN backs up all database files that have not been backed up in the last 7 days first, runs for 4 hours, and reads the blocks as fast as possible.

```
BACKUP DATABASE NOT BACKED UP SINCE 'sysdate - 7'
  PARTIAL DURATION 4:00 MINIMIZE TIME;
```

Each time this RMAN command is run, it backs up the data files that have not been backed up in the last 7 days first. You do not need to manually specify the tablespaces or data files to be backed up each night. Over the course of several days, all of your database files are backed up.

While this is a simplistic approach to database backup, it is easy to implement and provides more flexibility in backing up large amounts of data. During recovery, RMAN may point you to multiple different storage devices to perform the restore operation. Consequently, your recovery time may be longer.

9.4 The Data Warehouse Recovery Methodology

Devising a backup and recovery strategy can be a complicated and challenging task.

When you have hundreds of terabytes of data that must be protected and recovered for a failure, the strategy can be very complex. This section contains several best practices that can be implemented to ease the administration of backup and recovery.

This section contains the following topics:

- Best Practice 1: Use ARCHIVELOG Mode

- Best Practice 2: Use RMAN
- Best Practice 3: Use Block Change Tracking
- Best Practice 4: Use RMAN Multisection Backups
- Best Practice 5: Leverage Read-Only Tablespaces
- Best Practice 6: Plan for NOLOGGING Operations in Your Backup/Recovery Strategy
- Best Practice 7: Not All Tablespaces Should Be Treated Equally

9.4.1 Best Practice 1: Use ARCHIVELOG Mode

Archived redo logs are crucial for recovery when no data can be lost because they constitute a record of changes to the database.

Oracle Database can be run in either of two modes:

- `ARCHIVELOG`

  Oracle Database archives the filled online redo log files before reusing them in the cycle.

- `NOARCHIVELOG`

  Oracle Database does not archive the filled online redo log files before reusing them in the cycle.

Running the database in `ARCHIVELOG` mode has the following benefits:

- The database can be recovered from both instance and media failure.
- Backups can be performed while the database is open and available for use.
- Oracle Database supports multiplexed archive logs to avoid any possible single point of failure on the archive logs.
- More recovery options are available, such as the ability to perform tablespace point-in-time recovery (TSPITR).
- Archived redo logs can be transmitted and applied to the physical standby database, which is an exact replica of the primary database.

Running the database in `NOARCHIVELOG` mode has the following consequences:

- The database can be backed up only while it is closed after a clean shutdown.
- Typically, the only media recovery option is to restore the whole database to the point-in-time in which the full or incremental backups were made, which can result in the loss of recent transactions.

9.4.1.1 Is Downtime Acceptable?

It is important to design a backup plan to minimize database interruptions.

Oracle Database backups can be made while the database is open or closed. Planned downtime of the database can be disruptive to operations, especially in global enterprises that support users in multiple time zones, up to 24-hours per day.

Depending on the business, some enterprises can afford downtime. If the overall business strategy requires little or no downtime, then the backup strategy should implement an online backup. The database never needs to be taken down for a backup. An online backup requires the database to be in `ARCHIVELOG` mode.

Given the size of a data warehouse (and consequently the amount of time to back up a data warehouse), it is generally not viable to make an offline backup of a data warehouse, which would be necessitated if one were using `NOARCHIVELOG` mode.

9.4.2 Best Practice 2: Use RMAN

Many data warehouses, which were developed on earlier releases of Oracle Database, may not have integrated RMAN for backup and recovery.

However, just as there are many reasons to leverage `ARCHIVELOG` mode, there is a similarly compelling list of reasons to adopt RMAN. Consider the following:

1. Trouble-free backup and recovery
2. Corrupt block detection
3. Archive log validation and management
4. Block Media Recovery (BMR)
5. Easily integrates with Media Managers
6. Backup and restore optimization
7. Backup and restore validation
8. Downtime-free backups
9. Incremental backups
10. Extensive reporting

9.4.3 Best Practice 3: Use Block Change Tracking

Enabling block change tracking allows incremental backups to be completed faster, by reading and writing only the changed blocks since the last full or incremental backup.

For data warehouses, this can be extremely helpful if the database typically undergoes a low to medium percentage of changes.

> **See Also:**
>
> *Oracle Database Backup and Recovery User's Guide* for more information about block change tracking

9.4.4 Best Practice 4: Use RMAN Multisection Backups

With the advent of big file tablespaces, data warehouses have the opportunity to consolidate a large number of data files into fewer, better managed data files.

For backing up very large data files, RMAN provides multisection backups as a way to parallelize the backup operation within the file itself, such that sections of a file are backed up in parallel, rather than backing up on a per-file basis.

For example, a one TB data file can be sectioned into ten 100 GB backup pieces, with each section backed up in parallel, rather than the entire one TB file backed up as one file. The overall backup time for large data files can be dramatically reduced.

> **See Also:**
>
> *Oracle Database Backup and Recovery User's Guide* for more information about configuring multisection backups

9.4.5 Best Practice 5: Leverage Read-Only Tablespaces

An important issue facing a data warehouse is the sheer size of a typical data warehouse. Even with powerful backup hardware, backups may still take several hours.

One important consideration in improving backup performance is minimizing the amount of data to be backed up. Read-only tablespaces are the simplest mechanism to reduce the amount of data to be backed up in a data warehouse. Even with incremental backups, both backup and recovery are faster if tablespaces are set to read-only.

The advantage of a read-only tablespace is that data must be backed up only one time. If a data warehouse contains five years of historical data and the first four years of data can be made read-only, then theoretically the regular backup of the database would back up only 20% of the data. This can dramatically reduce the amount of time required to back up the data warehouse.

Most data warehouses store their data in tables that have been range-partitioned by time. In a typical data warehouse, data is generally active for a period ranging anywhere from 30 days to one year. During this period, the historical data can still be updated and changed (for example, a retailer may accept returns up to 30 days beyond the date of purchase, so that sales data records could change during this period). However, after data reaches a certain age, it is often known to be static.

By taking advantage of partitioning, users can make the static portions of their data read-only. Currently, Oracle supports read-only tablespaces rather than read-only partitions or tables. To take advantage of the read-only tablespaces and reduce the backup window, a strategy of storing constant data partitions in a read-only tablespace should be devised. Here are two strategies for implementing a rolling window:

1. When the data in a partition matures to the point where it is entirely static, implement a regularly scheduled process to move the partition from the current read-write tablespace to a tablespace that can then be made read-only.
2. Create a series of tablespaces, each containing a small number of partitions, and regularly modify a tablespace from read-write to read-only as the data in that tablespace ages.

One consideration is that backing up data is only half the recovery process. If you configure a tape system so that it can back up the read-write portions of a data warehouse in 4 hours, the corollary is that a tape system might take 20 hours to recover the database if a complete recovery is necessary when 80% of the database is read-only.

9.4.6 Best Practice 6: Plan for NOLOGGING Operations in Your Backup/Recovery Strategy

In general, a high priority for a data warehouse is performance. Not only must the data warehouse provide good query performance for online users, but the data warehouse must also be efficient during the extract, transform, and load (ETL) process so that large amounts of data can be loaded in the shortest amount of time. One common optimization used by data warehouses is to execute bulk-data operations using the `NOLOGGING` mode.

The database operations that support NOLOGGING modes are direct-path load and insert operations, index creation, and table creation. When an operation runs in NOLOGGING mode, data is not written to the redo log (or more precisely, only a small set of metadata is written to the redo log). This mode is widely used within data warehouses and can improve the performance of bulk data operations by up to 50%.

However, the tradeoff is that a NOLOGGING operation cannot be recovered using conventional recovery mechanisms, because the necessary data to support the recovery was never written to the log file. Moreover, subsequent operations to the data upon which a NOLOGGING operation has occurred also cannot be recovered even if those operations were not using NOLOGGING mode. Because of the performance gains provided by NOLOGGING operations, it is generally recommended that data warehouses use NOLOGGING mode in their ETL process.

The presence of NOLOGGING operations must be taken into account when devising the backup and recovery strategy. When a database is relying on NOLOGGING operations, the conventional recovery strategy (of recovering from the latest tape backup and applying the archived log files) is no longer applicable because the log files are not able to recover the NOLOGGING operation.

The first principle to remember is, do not make a backup when a NOLOGGING operation is occurring. Oracle Database does not currently enforce this rule, so DBAs must schedule the backup jobs and the ETL jobs such that the NOLOGGING operations do not overlap with backup operations.

There are two approaches to backup and recovery in the presence of NOLOGGING operations: ETL or incremental backups. If you are not using NOLOGGING operations in your data warehouse, then you do not have to choose either option: you can recover your data warehouse using archived logs. However, the options may offer some performance benefits over an archive log-based approach for a recovery. You can also use flashback logs and guaranteed restore points to flashback your database to a previous point in time.

This section contains the following topics:

- Extract, Transform, and Load
- The Extract, Transform, and Load Strategy
- Incremental Backup
- The Incremental Approach
- Flashback Database and Guaranteed Restore Points

9.4.6.1 Extract, Transform, and Load

The ETL process uses several Oracle features and a combination of methods to load (re-load) data into a data warehouse.

These features consist of:

- Transportable tablespaces

  Transportable tablespaces allow users to quickly move a tablespace across Oracle Databases. It is the most efficient way to move bulk data between databases. Oracle Database provides the ability to transport tablespaces across platforms. If the source platform and the target platform are of different endianness, then RMAN converts the tablespace being transported to the target format.

- SQL*Loader

SQL*Loader loads data from external flat files into tables of Oracle Database. It has a powerful data parsing engine that puts little limitation on the format of the data in the data file.

- Data Pump (export/import)

  Oracle Data Pump enables high-speed movement of data and metadata from one database to another. This technology is the basis for the Oracle Data Pump Export and Data Pump Import utilities.

- External tables

  The external tables feature is a complement to existing SQL*Loader functionality. It enables you to access data in external sources as if it were in a table in the database. External tables can also be used with the Data Pump driver to export data from a database, using `CREATE TABLE AS SELECT * FROM`, and then import data into Oracle Database.

9.4.6.2 The Extract, Transform, and Load Strategy

One approach is to take regular database backups and also store the necessary data files to re-create the ETL process for that entire week.

In the event where a recovery is necessary, the data warehouse could be recovered from the most recent backup. Then, instead of rolling forward by applying the archived redo logs (as would be done in a conventional recovery scenario), the data warehouse could be rolled forward by rerunning the ETL processes. This paradigm assumes that the ETL processes can be easily replayed, which would typically involve storing a set of extract files for each ETL process.

A sample implementation of this approach is to make a backup of the data warehouse every weekend, and then store the necessary files to support the ETL process each night. At most, 7 days of ETL processing must be reapplied to recover a database. The data warehouse administrator can easily project the length of time to recover the data warehouse, based upon the recovery speeds from tape and performance data from previous ETL runs.

Essentially, the data warehouse administrator is gaining better performance in the ETL process with `NOLOGGING` operations, at a price of slightly more complex and a less automated recovery process. Many data warehouse administrators have found that this is a desirable trade-off.

One downside to this approach is that the burden is on the data warehouse administrator to track all of the relevant changes that have occurred in the data warehouse. This approach does not capture changes that fall outside of the ETL process. For example, in some data warehouses, users may create their own tables and data structures. Those changes are lost during a recovery.

This restriction must be conveyed to the end-users. Alternatively, one could also mandate that end-users create all private database objects in a separate tablespace, and during recovery, the DBA could recover this tablespace using conventional recovery while recovering the rest of the database using the approach of rerunning the ETL process.

9.4.6.3 Incremental Backup

A more automated backup and recovery strategy in the presence of `NOLOGGING` operations uses RMAN's incremental backup capability.

Incremental backups provide the capability to back up only the changed blocks since the previous backup. Incremental backups of data files capture data changes on a block-by-block basis, rather than requiring the backup of all used blocks in a data file. The resulting backup

sets are generally smaller and more efficient than full data file backups, unless every block in the data file is changed.

When you enable block change tracking, Oracle Database tracks the physical location of all database changes. RMAN automatically uses the change tracking file to determine which blocks must be read during an incremental backup. The block change tracking file is approximately 1/30000 of the total size of the database.

> **See Also:**
>
> *Oracle Database Backup and Recovery User's Guide* for more information about block change tracking and how to enable it

9.4.6.4 The Incremental Approach

A typical backup and recovery strategy using this approach is to back up the data warehouse every weekend, and then take incremental backups of the data warehouse every night following the completion of the ETL process.

Incremental backups, like conventional backups, must not be run concurrently with `NOLOGGING` operations. To recover the data warehouse, the database backup would be restored, and then each night's incremental backups would be reapplied.

Although the `NOLOGGING` operations were not captured in the archive logs, the data from the `NOLOGGING` operations is present in the incremental backups. Moreover, unlike the previous approach, this backup and recovery strategy can be managed using RMAN.

9.4.6.5 Flashback Database and Guaranteed Restore Points

Flashback Database is a fast, continuous point-in-time recovery method to repair widespread logical errors.

Flashback Database relies on additional logging, called flashback logs, which are created in the fast recovery area and retained for a user-defined time interval according to the recovery needs. These logs track the original block images when they are updated.

When a Flashback Database operation is executed, just the block images corresponding to the changed data are restored and recovered, versus traditional data file restore where all blocks from the backup must be restored before recovery can start. Flashback logs are created proportionally to redo logs.

For very large and active databases, it may not be feasible to keep all needed flashback logs for continuous point-in-time recovery. However, there may be a requirement to create a specific point-in-time snapshot (for example, right before a nightly batch job) for logical errors during the batch run. For this scenario, guaranteed restore points can be created without enabling flashback logging.

When the guaranteed restore points are created, flashback logs are maintained just to satisfy Flashback Database to the guaranteed restore points and no other point in time, thus saving space. For example, guaranteed restore points can be created followed by a nologging batch job. As long as there are no previous nologging operations within the last hour of the creation time of the guaranteed restore points, Flashback Database to the guaranteed restore points undoes the nologging batch job. To flash back to a time after the nologging batch job finishes, then create the guaranteed restore points at least one hour away from the end of the batch job.

Estimating flashback log space for guaranteed restore points in this scenario depends on how much of the database changes over the number of days you intend to keep guaranteed restore points. For example, to keep guaranteed restore points for 2 days and you expect 100 GB of the database to change, then plan for 100 GB for the flashback logs. The 100 GB refers to the subset of the database changed after the guaranteed restore points are created and not the frequency of changes.

9.4.7 Best Practice 7: Not All Tablespaces Should Be Treated Equally

Not all of the tablespaces in a data warehouse are equally significant from a backup and recovery perspective.

Database administrators can use this information to devise more efficient backup and recovery strategies when necessary. The basic granularity of backup and recovery is a tablespace, so different tablespaces can potentially have different backup and recovery strategies.

On the most basic level, temporary tablespaces never need to be backed up (a rule which RMAN enforces). Moreover, in some data warehouses, there may be tablespaces dedicated to scratch space for users to store temporary tables and incremental results. These tablespaces are not explicit temporary tablespaces but are essentially functioning as temporary tablespaces. Depending upon the business requirements, these tablespaces may not need to be backed up and restored; instead, for a loss of these tablespaces, the users would re-create their own data objects.

In many data warehouses, some data is more important than other data. For example, the sales data in a data warehouse may be crucial and in a recovery situation this data must be online as soon as possible. But, in the same data warehouse, a table storing clickstream data from the corporate website may be much less critical to businesses. The business may tolerate this data being offline for a few days or may even be able to accommodate the loss of several days of clickstream data if there is a loss of database files. In this scenario, the tablespaces containing sales data must be backed up often, while the tablespaces containing clickstream data need to be backed up only once every week or two weeks.

While the simplest backup and recovery scenario is to treat every tablespace in the database the same, Oracle Database provides the flexibility for a DBA to devise a backup and recovery scenario for each tablespace as needed.

10

Storage Management for VLDBs

Storage management for the database files in a VLDB environment includes high availability, performance, and manageability aspects.

Storage performance in data warehouse environments often translates into I/O throughput (MB/s). For online transaction processing (OLTP) systems, the number of I/O requests per second (IOPS) is a key measure for performance.

This chapter discusses storage management for the database files in a VLDB environment only. Nondatabase files, including Oracle Database software, are not discussed because management of those files is no different from a non-VLDB environment. The focus is on the high availability, performance, and manageability aspects of storage systems for VLDB environments.

This chapter contains the following sections:

- High Availability
- Performance
- Scalability and Manageability
- Oracle ASM Settings Specific to VLDBs

> **Note:**
>
> Oracle Database supports the use of database files on raw devices and on file systems, and supports the use of Oracle Automatic Storage Management (Oracle ASM) on top of raw devices or logical volumes. Oracle ASM should be used whenever possible.

10.1 High Availability

High availability can be achieved by implementing storage redundancy.

In storage terms, these are mirroring techniques. There are three options for mirroring in a database environment:

- Hardware-based mirroring
- Using Oracle ASM for mirroring
- Software-based mirroring not using Oracle ASM

  Oracle recommends against software-based mirroring that does not use Oracle ASM.

This section contains the following topics:

- Hardware-Based Mirroring
- Mirroring Using Oracle ASM

> **Note:**
>
> In a cluster configuration, the software you use must support cluster capabilities. Oracle ASM is a cluster file system for Oracle Database files.

10.1.1 Hardware-Based Mirroring

Most external storage devices provide support for different RAID (Redundant Array of Independent Disks) levels.

The most commonly used high availability hardware RAID levels in VLDB environments are RAID 1 and RAID 5. Though less commonly used in VLDB environments, other high availability RAID levels can also be used.

This section contains the following topics:

- RAID 1 Mirroring
- RAID 5 Mirroring

10.1.1.1 RAID 1 Mirroring

RAID 1 is a basic mirroring technique.

Every storage block that has been written to storage is stored twice on different physical devices as defined by the RAID setup. RAID 1 provides fault tolerance because if one device fails, then there is another, mirrored, device that can respond to the request for data. The two write operations in a RAID 1 setup are generated at the storage level. RAID 1 requires at least two physical disks to be effective.

Storage devices generally provide capabilities to read either the primary or the mirror in case a request comes in, which may result in better performance compared to other RAID configurations designed for high availability. RAID 1 is the simplest hardware high availability implementation but requires double the amount of storage needed to store the data. RAID 1 is often combined with RAID 0 (striping) in RAID 0+1 configurations. In the simplest RAID 0+1 configuration, individual stripes are mirrored across two physical devices.

10.1.1.2 RAID 5 Mirroring

RAID 5 requires at least 3 storage devices, but commonly 4 to 6 devices are used in a RAID 5 group.

When using RAID 5, for every data block written to a device, parity is calculated and stored on a different device. On read operations, the parity is checked. The parity calculation takes place in the storage layer. RAID 5 provides high availability for a device failure because the device's contents can be rebuilt based on the parities stored on other devices.

RAID 5 provides good read performance. Write performance may be slowed down by the parity calculation in the storage layer. RAID 5 does not require double the amount of storage but rather a smaller percentage depending on the number of devices in the RAID 5 group. RAID 5 is relatively complex and consequently, not all storage devices support a RAID 5 setup.

10.1.2 Mirroring Using Oracle ASM

Oracle Automatic Storage Management (Oracle ASM) provides software-based mirroring capabilities.

Oracle ASM provides support for normal redundancy (mirroring) and high redundancy (triple mirroring). Oracle ASM also supports the use of external redundancy, in which case Oracle ASM does not perform additional mirroring. Oracle ASM normal redundancy can be compared to RAID 1 hardware mirroring.

With Oracle ASM mirroring, the mirror is produced by the database servers. Consequently, write operations require more I/O throughput when using Oracle ASM mirroring compared to using hardware-based mirroring. Depending on your configuration and the speed of the hardware RAID controllers, Oracle ASM mirroring or hardware RAID may introduce a bottleneck for data loads.

In Oracle ASM, the definition of *failure groups* enables redundancy, as Oracle ASM mirrors data across the boundaries of the failure group. For example, in a VLDB environment, you can define one failure group per disk array, in which case Oracle ASM ensures that mirrored data is stored on a different disk array. That way, you could not only survive a failure of a single disk in a disk array, but you could even survive the failure of an entire disk array or failure of all channels to that disk array. Hardware RAID configurations typically do not support this kind of fault tolerance.

Oracle ASM using normal redundancy requires double the amount of disk space needed to store the data. High redundancy requires triple the amount of disk space.

> **See Also:**
>
> *Oracle Automatic Storage Management Administrator's Guide*

10.2 Performance

To achieve the optimum throughput from storage devices, multiple disks must work in parallel.

Optimum throughput can be achieved using a technique called **striping**, which stores data blocks in equisized slices (stripes) across multiple devices. Striping enables storage configurations for good performance and throughput.

Optimum storage device performance is a trade-off between seek time and accessing consecutive blocks on disk. In a VLDB environment, a 1 MB stripe size provides a good balance for optimal performance and throughput, both for OLTP systems and data warehouse systems. There are three options for striping in a database environment:

- Hardware-based striping
- Software-based striping using Oracle ASM
- Software-based striping not using Oracle ASM

It is possible to use a combination of striping techniques, but you must ensure that you physically store stripes on different devices to get the performance advantages out of striping. From a conceptual perspective, software-based striping not using Oracle ASM is very similar to hardware-based striping.

This section contains the following topics:

- Hardware-Based Striping
- Striping Using Oracle ASM
- Information Lifecycle Management
- Partition Placement
- Bigfile Tablespaces
- Oracle Database File System (DBFS)

> **Note:**
>
> In a cluster configuration, the software you use must support cluster capabilities. Oracle ASM is a cluster file system for Oracle Database files.

10.2.1 Hardware-Based Striping

Most external storage devices provide striping capabilities. The most commonly used striping techniques to improve storage performance are RAID 0 and RAID 5.

This section contains the following topics:

- RAID 0 Striping
- RAID 5 Striping

10.2.1.1 RAID 0 Striping

RAID 0 requires at least two devices to implement.

Data blocks written to the devices are split up and alternatively stored across the devices using the stripe size. This technique enables the use of multiple devices and multiple channels to the devices.

RAID 0, despite its RAID name, is not redundant. Loss of a device in a RAID 0 configuration results in data loss, and should always be combined with some redundancy in a critical environment. Database implementations using RAID 0 are often combined with RAID 1, basic mirroring, in RAID 0+1 configurations.

10.2.1.2 RAID 5 Striping

RAID 5 configurations spread data across the available devices in the RAID group using a hardware-specific stripe size.

Consequently, multiple devices and channels are used to read and write data. Due to its more complex parity calculation, not all storage devices support RAID 5 configurations.

10.2.2 Striping Using Oracle ASM

Oracle Automatic Storage Management (Oracle ASM) always stripes across all devices presented to it as a disk group.

A disk group is a logical storage pool in which you create data files. The default Oracle ASM stripe size generally is a good stripe size for a VLDB.

Use disks with the same performance characteristics in a disk group. All disks in a disk group should also be the same size for optimum data distribution and hence optimum performance and throughput. The disk group should span as many physical spindles as possible to get the best performance. The disk group configuration for a VLDB does not have to be different from the disk group configuration for a non-VLDB.

Oracle ASM can be used on top of previously striped storage devices. If you use such a configuration, then ensure that you do not introduce hot spots by defining disk groups that span logical devices which physically may be using the same resource (disk, controller, or channel to disk) rather than other available resources. Always ensure that Oracle ASM stripes are distributed equally across all physical devices.

> **See Also:**
>
> *Oracle Automatic Storage Management Administrator's Guide* for more information about Oracle ASM striping

10.2.3 Information Lifecycle Management

In an Information Lifecycle Management (ILM) environment, you cannot use striping across all devices, because all data would then be distributed across all storage pools.

In an ILM environment, different storage pools typically have different performance characteristics. Tablespaces should not span storage pools, and data files for the same tablespace should not be stored in multiple storage pools.

Storage in an ILM environment should be configured to use striping across all devices in a storage pool. If you use Oracle ASM, then separate disk groups for different storage pools should be created. Using this approach, tablespaces do not store data files in different disk groups. Data can be moved online between tablespaces using partition movement operations for partitioned tables, or using the `DBMS_REDEFINITION` package when the tables are not partitioned.

> **See Also:**
>
> Managing and Maintaining Time-Based Information for information about Information Lifecycle Management environment

10.2.4 Partition Placement

Partition placement is not a concern if you stripe across all available devices and distribute the load across all available resources.

If you cannot stripe data files across all available devices, then consider partition placement to optimize the use of all available hardware resources (physical disk spindles, disk controllers, and channels to disk).

I/O-intensive queries or DML operations should make optimal use of all available resources. Storing database object partitions in specific tablespaces, each of which uses a different set of

hardware resources, enables you to use all resources for operations against a single partitioned database object. Ensure that I/O-intensive operations can use all resources by using an appropriate partitioning technique.

Hash partitioning and hash subpartitioning on a unique or almost unique column or set of columns with the number of hash partitions equal to a power of 2 is the only technique likely to result in an even workload distribution when using partition placement to optimize I/O resource utilization. Other partitioning and subpartitioning techniques may yield similar benefits depending on your application.

10.2.5 Bigfile Tablespaces

Oracle Database enables the creation of bigfile tablespaces.

A **bigfile** tablespace consists of a single data or temporary file which can be up to 128 TB. The use of bigfile tablespaces can significantly reduce the number of data files for your database. Oracle Database supports parallel RMAN backup and restore on single data files.

Consequently, there is no disadvantage to using bigfile tablespaces and you may choose to use bigfile tablespaces to significantly reduce the number of data and temporary files.

File allocation is a serial process. If you use automatic allocation for your tables and automatically extensible data files, then a large data load can be impacted by the amount of time it takes to extend the file, regardless of whether you use bigfile tablespaces. However, if you preallocate data files and you use multiple data files, then multiple processes are spawned to add data files concurrently.

> **See Also:**
>
> *Oracle Database Backup and Recovery User's Guide*

10.2.6 Oracle Database File System (DBFS)

Oracle Database File System (DBFS) leverages the benefits of the database to store files, and the strengths of the database in efficiently managing relational data to implement a standard file system interface for files stored in the database.

With this interface, storing files in the database is no longer limited to programs specifically written to use BLOB and CLOB programmatic interfaces. Files in the database can now be transparently accessed using any operating system (OS) program that acts on files.

Oracle Database File System (DBFS) creates a standard file system interface on top of files and directories that are stored in database tables. With DBFS, the server is the database. Files are stored as Oracle SecureFiles LOBs in a database table. A set of PL/SQL procedures implement the file system access primitives such as create, open, read, write, and list directory. The implementation of the file system in the database is called the DBFS Content Store. The DBFS Content Store allows each database user to create one or more file systems that can be mounted by clients. Each file system has its own dedicated tables that hold the file system content.

> **See Also:**
>
> *Oracle Database SecureFiles and Large Objects Developer's Guide* for information about Oracle SecureFiles LOBs, stores, and Oracle Database File System

10.3 Scalability and Manageability

Storage scalability and management is an important factor in a VLDB environment.

A very important characteristic of a VLDB is its large size. The large size introduces the following challenges:

- Simple statistics suggest that storage components are more likely to fail because VLDBs use more components.
- A small relative growth in a VLDB may amount to a significant absolute growth, resulting in possibly many devices to be added.
- Despite its size, performance and (often) availability requirements are not different from smaller systems.

The storage configuration you choose should be able to handle these challenges. Regardless of whether storage is added or removed, deliberately or accidentally, your system should remain in an optimal state from a performance and high availability perspective.

This section contains the following topics:

- Stripe and Mirror Everything (SAME)
- SAME and Manageability

10.3.1 Stripe and Mirror Everything (SAME)

The stripe and mirror everything (SAME) methodology has been recommended by Oracle for many years and is a means to optimize high availability, performance, and manageability.

To simplify the configuration further, a fixed stripe size of 1 MB is recommended in the SAME methodology as a good starting point for both OLTP and data warehouse systems. Oracle ASM implements the SAME methodology and adds automation on top of it.

10.3.2 SAME and Manageability

To achieve maximum performance, the SAME methodology proposes to stripe across as many physical devices as possible.

This can be achieved without Oracle ASM, but if the storage configuration changes, for example, by adding or removing devices, then the layout of the database files on the devices should change. Oracle ASM performs this task automatically in the background. In most non-Oracle ASM environments, re-striping is a major task that often involves manual intervention.

In an ILM environment, you apply the SAME methodology to every storage pool.

10.4 Oracle ASM Settings Specific to VLDBs

Configuration of Oracle Automatic Storage Management for VLDBs is similar to Oracle ASM configuration for non-VLDBs.

Certain parameter values, such as the memory allocation to the Oracle ASM instance, may need a higher value.

Oracle Database supports Oracle ASM variable allocation units. Large variable allocation units are beneficial for environments that use large sequential I/O operations. VLDBs in general, and large data warehouses in particular, are good candidate environments to take advantage of large allocation units. Allocation units can be set between 1 MB and 64 MB in powers of two (that is, 1, 2, 4, 8, 16, 32, and 64). If your workload contains a significant number of queries scanning large tables, then you should use large Oracle ASM allocation units. Use 64 MB for a very large data warehouse system. Large allocation units also reduce the memory requirements for Oracle ASM and improve the Oracle ASM startup time.

See Also:

Oracle Automatic Storage Management Administrator's Guide for information about how to set up and configure Oracle ASM

Glossary

automatic data optimization

Automatic data optimization (ADO) is a component of Information Lifecycle Management (ILM) that automates the compression and movement of data between different tiers of storage within the database.

composite partitioning

Composite partitioning is a combination of the basic data distribution methods. After a table is partitioned by a data distribution method, then each partition is subdivided into subpartitions using the same or a different data distribution method.

degree of parallelism (DOP)

The degree of parallelism (DOP) is the number of parallel execution (PX) servers associated with a single operation.

distribution method (parallel execution)

A distribution method is the method by which data is sent, or redistributed, from one parallel execution (PX) server set to another.

hash partitioning

Hash partitioning maps data to partitions based on a hashing algorithm that Oracle applies to the partitioning key that you identify.

heat map

Heat map is a component of Information Lifecycle Management (ILM) that tracks data access at the segment-level and data modification at the segment and row level.

Information Lifecycle Management

Information Lifecycle Management (ILM) is a set of processes and policies for managing data throughout its useful life.

list partitioning

List partitioning maps rows to partitions by specifying a list of discrete values for the partitioning key in the description for each partition.

parallel execution (PX)

Parallel execution (PX) is the ability to apply multiple CPU and I/O resources to the execution of a single SQL statement with the use of multiple processes.

parallel execution server

The parallel execution (PX) servers are the individual processes that perform work in parallel on behalf of the initiating session.

parallelism

Parallelism is breaking down a task so that many processes simultaneously do part of the work in a query, rather than one process doing all of the work.

partition pruning

Partition pruning occurs when the results of a query can be achieved by accessing a subset of partitions, rather than the entire table.

partitioning

Partitioning is the process of subdividing objects, such as tables and indexes, into smaller and more manageable pieces.

partitioning key

The partitioning key consists of one or more columns that determine the partition where each row is stored.

query coordinator (QC)

The query coordinator (QC), or also called the parallel execution (PX) coordinator, is the session that initiates the parallel SQL statement.

range partitioning

Range partitioning maps data to partitions based on ranges of values of the partitioning key that have been specified for each partition.

very large database (VLDB)

A very large database is a database that contains a very large number of rows, or occupies an extremely large physical storage space.

Index

A

adaptive algorithm, *8-18*
ADD PARTITION clause, *4-57*
ADD SUBPARTITION clause, *4-59–4-61*
adding ILM policies
  for Automatic Data Optimization, *5-20*
adding index partitions, *4-62*
adding multiple partitions, *4-62*
adding partitions
  composite hash-partitioned tables, *4-59*
  composite list-partitioned tables, *4-59*
  composite range-partitioned tables, *4-60*
  hash-partitioned tables, *4-57*
  interval-partitioned tables, *4-58*
  list-partitioned tables, *4-57*
  partitioned tables, *4-56*
  range-partitioned tables, *4-57*
  reference-partitioned tables, *4-61*
ALTER INDEX statement
  partition attributes, *3-38*
ALTER SESSION statement
  ENABLE PARALLEL DML clause, *8-38*
  FORCE PARALLEL DDL clause, *8-31*, *8-35*
  FORCE PARALLEL DML clause, *8-39*, *8-40*
ALTER TABLE statement
  MODIFY DEFAULT ATTRIBUTES clause, *4-88*
  MODIFY DEFAULT ATTRIBUTES FOR PARTITION clause, *4-88*
applications
  decision support system (DSS)
    parallel SQL, *8-33*
  direct-path INSERT, *8-38*
  parallel DML operations, *8-37*
asynchronous communication
  parallel execution servers, *8-9*
asynchronous I/O, *8-65*
Automatic big table caching
  about, *8-19*
Automatic Data Optimization
  adding ILM policies, *5-20*
  and heat map, *5-12*
  DBMS_ILM package, *5-24*
  DBMS_ILM_ADMIN package, *5-24*
  deleting ILM policies, *5-21*
Automatic Data Optimization *(continued)*
  disabling ILM policies, *5-21*
  ILM ADO parameters, *5-23*
  limitations, *5-26*
  managing ILM policies, *5-17*
  managing with Oracle Enterprise Manager, *5-39*
  monitoring DBA and ILM policy views, *5-25*
  row-level compression tiering, *5-22*
  segment-level compression tiering, *5-21*
  views for ILM policies, *5-25*
Automatic Data Optimization (ADO)
  for Information Lifecycle Management strategy, *5-16*
automatic list partitioning
  creating tables using, *4-10*

B

backing up and recovering
  very large databases (VLDBs), *9-1*
bigfile tablespaces
  very large databases (VLDBs), *10-6*
binary XML tables
  partitioning of XMLIndex, *4-30*

C

COALESCE PARTITION clause, *4-64*
collection tables
  performing PMOs on partitions, *4-29*
collections
  tables, *4-29*
  XMLType, *2-9*, *4-28*
composite hash partitioned tables
  creating, *4-31*
composite hash-hash partitioning, *2-15*
composite hash-list partitioning, *2-15*
composite hash-partitioned tables
  adding partitions, *4-59*
composite hash-range partitioning, *2-15*
composite interval partitioning
  creating tables using, *4-31*
composite list partitioning
  creating tables using, *4-34*

composite list-hash partitioning, 2-15
  performance considerations, 3-48
composite list-list partitioning, 2-15
  performance considerations, 3-49
composite list-partitioned tables
  adding partitions, 4-59
composite list-range partitioning, 2-15
  performance considerations, 3-50
composite partitioned tables
  creating, 4-30
composite partitioning, 2-13
  default partition, 4-41
  interval-hash, 4-32
  interval-list, 4-33
  interval-range, 4-34
  list-hash, 4-35
  list-list, 4-35
  list-range, 4-36
  performance considerations, 3-45
  range-hash, 4-38
  range-list, 4-39
  range-range, 4-42
  subpartition template, modifying, 4-53
composite range-* partitioned tables
  creating, 4-37
composite range-hash partitioning, 2-14
  performance considerations, 3-46
composite range-interval partitioning
  creating tables using, 4-31
composite range-list partitioned tables
  creating, 4-40
composite range-list partitioning, 2-15
  performance considerations, 3-47
composite range-partitioned tables
  adding partitions, 4-60
composite range-range partitioning, 2-14
  performance considerations, 3-48
compression
  partitioning, 3-39
compression table
  partitioning, 3-39
concurrent execution of union all, 8-45
constraints
  parallel create table, 8-36
consumer operations, 8-4
CREATE INDEX statement
  partition attributes, 3-38
  partitioned indexes, 4-5
CREATE TABLE AS SELECT statement
  decision support system, 8-33
CREATE TABLE statement
  AS SELECT
    rules of parallelism, 8-36
    space fragmentation, 8-34
    temporary storage space, 8-34
  parallelism, 8-33
creating hash partitioned tables
  examples, 4-6
creating indexes on partitioned tables
  restrictions, 2-23
creating interval partitions
  INTERVAL clause of CREATE TABLE, 4-5
creating partitions, 4-1
creating segments on demand
  maintenance procedures, 4-25
critical consumer group
  specifying for parallel statement queuing, 8-25

D

data
  parallel DML restrictions and integrity rules, 8-43
data loading
  incremental in parallel, 8-82
data manipulation language
  parallel DML operations, 8-36
  transaction model for parallel DML operations, 8-40
data segment compression
  bitmap indexes, 3-40
  example, 3-41
  partitioning, 3-39
data warehouses
  about, 6-1
  advanced partition pruning, 6-4
  ARCHIVELOG mode for recovery, 9-8
  backing up and recovering, 9-1
  backing up and recovering characteristics, 9-2
  backing up tables on individual basis, 9-14
  backup and recovery, 9-6
  basic partition pruning, 6-3
  block change tracking for backups, 9-9
  data compression and partitioning, 6-15
  differences with online transaction processing backups, 9-2
  extract, transform, and load for backup and recovery, 9-11
  extract, transform, and load strategy, 9-12
  flashback database and guaranteed restore points, 9-13
  incremental backup strategy, 9-13
  incremental backups, 9-12
  leverage read-only tablespaces for backups, 9-10
  manageability, 6-14
  manageability with partition exchange load, 6-14
  materialized views and partitioning, 6-12
  more complex queries, 6-2
  more users querying the system, 6-2

data warehouses *(continued)*
NOLOGGING mode for backup and recovery, *9-10*
partition pruning, *6-3*
partitioned tables, *3-42*
partitioning, *6-1*
partitioning and removing data from tables, *6-15*
partitioning for large databases, *6-2*
partitioning for large tables, *6-2*
partitioning for scalability, *6-1*
partitioning materialized views, *6-13*
recovery methodology, *9-7*
recovery point object (RPO), *9-6*
recovery time object (RTO), *9-6*
refreshing table data, *8-38*
RMAN for backup and recovery, *9-9*
RMAN multi-section backups, *9-9*
database writer process (DBWn)
tuning, *8-77*
databases
partitioning, and, *1-3*
scalability, *8-37*
DB_BLOCK_SIZE initialization parameter
parallel query, *8-64*
DB_CACHE_SIZE initialization parameter
parallel query, *8-64*
DB_FILE_MULTIBLOCK_READ_COUNT initialization parameter
parallel query, *8-64*
DBMS_HEAT_MAP package
subprograms for Heat MAP, *5-16*
DBMS_ILM package
Automatic Data Optimization, *5-24*
DBMS_ILM_ADMIN package
Automatic Data Optimization, *5-24*
decision support system (DSS)
parallel DML operations, *8-37*
parallel SQL, *8-33*, *8-37*
performance, *8-37*
scoring tables, *8-38*
default partitions, *4-9*
default subpartition, *4-41*
deferred segments
partitioning, *4-24*
degree of parallelism
adaptive parallelism, *8-18*
automatic, *8-15*
between query operations, *8-4*
controlling with initialization parameters and hints, *8-17*
determining for auto DOP, *8-15*
in-memory parallel execution, *8-18*
manually specifying, *8-14*
parallel execution servers, *8-13*
specifying a limit for a consumer group, *8-25*
DELETE statement
parallel DELETE statement, *8-39*
deleting ILM policies
for Automatic Data Optimization, *5-21*
direct-path INSERT
restrictions, *8-42*
DISABLE ROW MOVEMENT clause, *4-1*
DISABLE_PARALLEL_DML SQL hint, *8-38*
disabling ILM policies
for Automatic Data Optimization, *5-21*
DISK_ASYNCH_IO initialization parameter
parallel query, *8-65*
distributed transactions
parallel DML restrictions, *8-45*
DML_LOCKS
parallel DML, *8-62*
DROP PARTITION clause, *4-65*
dropping multiple partitions, *4-68*
dropping partitioned tables, *4-118*
DSS database
partitioning indexes, *3-37*

E

ENABLE ROW MOVEMENT clause, *4-1*, *4-4*
ENABLE_PARALLEL_DML SQL hint, *8-38*
equipartitioning
examples, *3-25*
local indexes, *3-24*
EXCHANGE PARTITION clause, *4-76*–*4-78*
EXCHANGE SUBPARTITION clause, *4-76*
exchanging partitions
cascade option, *4-78*
of a referenced-partition table, *4-74*
extents
parallel DDL statements, *8-34*
extract, transform, and load
data warehouses, *9-11*

F

filtering
maintenance operations on partitions, *4-53*
FOR PARTITION clause, *4-88*
fragmentation
parallel DDL, *8-34*
full partition-wise join
querying, *3-15*
full partition-wise joins, *3-14*, *6-6*
composite - composite, *3-19*
composite - single-level, *3-17*
single-level - single-level, *3-15*
full table scans
parallel execution, *8-2*
functions
parallel DML and DDL statements, *8-47*

functions *(continued)*
parallel execution, *8-46*
parallel queries, *8-47*

G

global hash partitioned indexes
about, *2-21*
global indexes
partitioning, *3-34*
summary of index types, *3-35*
global nonpartitioned indexes
about, *2-22*
global partitioned indexes
about, *2-20*
maintenance, *2-21*
global range partitioned indexes
about, *2-21*
granuless
parallelism, *8-5*
groups
instance, *8-12*

H

hardware-based mirroring
very large databases (VLDBs), *10-2*
hardware-based striping
very large databases (VLDBs), *10-4*
hash partitioning, *2-13*
creating global indexes, *4-7*
creating tables examples, *4-6*
creating tables using, *4-6*
index-organized tables, *4-26*
multicolumn partitioning keys, *4-19*
performance considerations, *3-43*
hash partitions
splitting, *4-104*
hash-partitioned tables
adding partitions, *4-57*
heap-organized partitioned tables
table compression, *4-23*
Heat Map
ALL, DBA, USER, and V$ views, *5-14*
and automatic data optimization, *5-12*
disabling, *5-13*
enabling, *5-13*
for Information Lifecycle Management strategy, *5-12*
limitations, *5-26*
managing with DBMS_HEAT_MAP subprograms, *5-16*
managing with Oracle Enterprise Manager, *5-39*
viewing tracking information, *5-14*
heat map and automatic data optimization
implementing an ILM strategy, *5-12*
HEAT_MAP initialization parameter
disabling, *5-13*
enabling, *5-13*
hints
parallel statement queuing, *8-29*
Hybrid Columnar Compression
example, *3-41*
hybrid partitioned tables
about, *2-6*
converting from, *4-127*
converting to, *4-126*
creating, *4-123*
exchanging a partition with a nonpartitioned external table, *4-130*
managing, *4-122*
splitting partitions, *4-129*
using with ADO, *4-128*

I

I/O
asynchronous, *8-65*
parallel execution, *8-2*
ILM
See Information Lifecycle Management
ILM policies
for Automatic Data Optimization, *5-17*
implementing an ILM system
manually with partitioning, *5-36*
using Oracle Database, *5-4*
In-Database Archiving
limitations, *5-35*
index partitions
adding, *4-62*
index-organized tables
hash-partitioned, *4-26*
list-partitioned, *4-27*
parallel queries, *8-30*
partitioning, *4-1*, *4-25*
partitioning secondary indexes, *4-26*
range-partitioned, *4-26*
indexes
advanced compression with partitioning, *3-37*
creating in parallel, *8-78*
global partitioned, *6-12*
global partitioned indexes, *3-34*
managing partitions, *3-34*
local indexes, *3-24*
local partitioned, *6-11*
manageability with partitioning, *6-15*
nonpartitioned, *6-11*
parallel creation, *8-78*
parallel DDL storage, *8-34*
parallel local, *8-78*

indexes *(continued)*
partitioned, *6-10*
partitioning, *3-23*
partitioning guidelines, *3-37*
partitions, *1-1*
updating automatically, *4-50*
updating global indexes, *4-50*
when to partition, *2-4*
Information Lifecycle Management, *5-1*, *5-12*
about, *5-1*
and HEAT_MAP initialization parameter, *5-12*
application transparency, *5-2*
assigning classes to storage tiers, *5-8*
auditing, *5-11*
benefits of an online archive, *5-3*
controlling access to data, *5-10*
creating data access, *5-9*
creating migration policies, *5-9*
creating storage tiers, *5-7*
data retention, *5-11*
defining compliance policies, *5-11*
defining data classes, *5-4*
enforceable compliance policies, *5-2*
enforcing compliance policies, *5-11*
expiration, *5-12*
fine-grained, *5-2*
heat map and automatic data optimization, *5-12*
immutability, *5-11*
implemented with Automatic Data Optimization, *5-16*
implementing a system manually with partitioning, *5-36*
implementing using Oracle Database, *5-4*
implementing with Heat Map, *5-12*
introduction, *5-1*
lifecycle of data, *5-7*
limitations, *5-26*
low-cost storage, *5-2*
moving data using partitioning, *5-10*
Oracle Database, and, *5-2*
partitioning, *5-5*
partitioning, and, *1-3*
privacy, *5-11*
regulatory requirements, *5-3*
striping, *10-5*
structured and unstructured data, *5-2*
time-based information, *5-1*
initialization parameters
MEMORY_MAX_TARGET, *8-59*
MEMORY_TARGET, *8-59*
PARALLEL_EXECUTION_MESSAGE_SIZE, *8-60*, *8-61*
PARALLEL_FORCE_LOCAL, *8-53*
PARALLEL_MAX_SERVERS, *8-53*
PARALLEL_MIN_PERCENT, *8-54*
initialization parameters *(continued)*
PARALLEL_MIN_SERVERS, *8-10*, *8-55*
PARALLEL_MIN_TIME_THRESHOLD, *8-55*
PARALLEL_SERVERS_TARGET, *8-55*
SHARED_POOL_SIZE, *8-56*
INSERT statement
parallelizing INSERT SELECT, *8-40*
instance groups
for parallel operations, *8-12*
limiting the number of instances, *8-12*
integrity rules
parallel DML restrictions, *8-43*
interval partitioned tables
dropping partitions, *4-68*
interval partitioning
creating tables using, *4-5*
manageability, *2-16*
performance considerations, *3-42*, *3-51*
interval-hash partitioning
creating tables using, *4-32*
subpartitioning template, *4-44*
interval-list partitioning
creating tables using, *4-33*
subpartitioning template, *4-45*
interval-partitioned tables
adding partitions, *4-58*
splitting partitions, *4-104*
interval-range partitioning
creating tables using, *4-34*
interval-reference partitioned tables
creating, *4-15*

J

joins
full partition-wise, *3-14*
partial partition-wise, *3-19*
partition-wise, *3-14*

K

key compression
partitioning indexes, *4-23*

L

list partitioning, *2-13*
adding values to value list, *4-90*
creating tables using, *4-8*
dropping values from value-list, *4-91*
index-organized tables, *4-27*
modifying, *4-90*
performance considerations, *3-45*
list-hash partitioning
creating tables using, *4-35*
subpartitioning template, *4-44*

list-list partitioning
creating tables using, 4-35
subpartitioning template, 4-45
list-partitioned tables
adding partitions, 4-57
creating, 4-8
splitting partitions, 4-101, 4-105
list-range partitioning
creating tables using, 4-36
LOB data types
restrictions on parallel DDL statements, 8-32
restrictions on parallel DML operations, 8-42
local indexes, 3-24, 3-35
equipartitioning, 3-24
local partitioned indexes
about, 2-19
LOGGING clause, 8-77
logging mode
parallel DDL, 8-32, 8-34

M

maintenance operations
supported on index partitions, 4-45
supported on partitioned tables, 4-45
maintenance operations on partitions
filtering, 4-53
manageability
data warehouses, 6-14
managing data validity
Temporal Validity, 5-32
managing ILM policies
for Automatic Data Optimization, 5-17
memory
configure at 2 levels, 8-59
MEMORY_MAX_TARGET initialization parameter, 8-59
MEMORY_TARGET initialization parameter, 8-59
MERGE PARTITION clause, 4-80
MERGE statement
parallel MERGE statement, 8-39
MERGE SUBPARTITION clause, 4-80
merging multiple partitions, 4-86
MINIMUM EXTENT parameter, 8-34
mirroring with Oracle ASM
very large databases (VLDBs), 10-3
MODIFY DEFAULT ATTRIBUTES clause, 4-88
using for partitioned tables, 4-88
MODIFY DEFAULT ATTRIBUTES FOR PARTITION clause, 4-88
of ALTER TABLE statement, 4-88
MODIFY PARTITION clause, 4-88, 4-89, 4-94, 4-97
MODIFY SUBPARTITION clause, 4-89
modifying
partitioning, 4-92
monitoring
parallel processing, 8-65, 8-66
MOVE PARTITION clause, 4-88, 4-94
MOVE SUBPARTITION clause, 4-88, 4-95
multi-column list partitioning
creating tables using, 4-11
multiple archiver processes, 8-77
multiple block sizes
restrictions on partitioning, 4-27
multiple parallelizers, 8-11
multiple partitions
adding, 4-62
dropping, 4-68
merging, 4-86
splitting, 4-108
truncating, 4-113

N

NO_STATEMENT_QUEUING
parallel statement queuing hint, 8-29
NOLOGGING clause, 8-77
NOLOGGING mode
parallel DDL, 8-32, 8-34
non-partitioned tables
converting to partitioned tables, 4-121
nonpartitioned indexes, 6-11
nonpartitioned tables
changing to partitioned tables, 4-119
nonprefixed indexes, 2-19, 3-26
global partitioned indexes, 3-34
nonprefixed indexes_importance, 3-36

O

object types
parallel queries, 8-31
restrictions on parallel DDL statements, 8-32
restrictions on parallel DML operations, 8-42
restrictions on parallel queries, 8-31
of ALTER TABLE statement, 4-88
OLTP database
batch jobs, 8-38
parallel DML operations, 8-37
partitioning indexes, 3-37
Online Transaction Processing (OLTP)
about, 7-1
common partition maintenance operations, 7-6
partitioning, and, 7-1
when to partition indexes, 7-3
operating system statistics
monitoring for parallel processing, 8-72
operations
partition-wise, 3-14

optimization
partition pruning and indexes, 3-36
partitioned indexes, 3-36
optimizations
parallel SQL, 8-4
Oracle Automatic Storage Management settings
very large databases (VLDBs), 10-7
Oracle Database File System
very large databases (VLDBs), 10-6
Oracle Database Resource Manager
managing parallel statement queue, 8-21
Oracle Real Application Clusters
instance groups, 8-12

P

PARALLEL clause, 8-79
parallel DDL statements, 8-32
extent allocation, 8-34
partitioned tables and indexes, 8-32
restrictions on LOBs, 8-32
restrictions on object types, 8-31, 8-32
parallel delete, 8-39
parallel DELETE statement, 8-39
parallel DML
considerations for parallel execution, 8-75
parallel DML and DDL statements
functions, 8-47
parallel DML operations, 8-36
applications, 8-37
enabling PARALLEL DML, 8-38
recovery, 8-41
restrictions, 8-42
restrictions on LOB data types, 8-42
restrictions on object types, 8-31, 8-42
restrictions on remote transactions, 8-45
transaction model, 8-40
parallel execution
about, 8-1, 8-3
adaptive parallelism, 8-18
bandwidth, 8-2
benefits, 8-2
considerations for parallel DML, 8-75
CPU utilization, 8-2
CREATE TABLE AS SELECT statement, 8-73
DB_BLOCK_SIZE initialization parameter, 8-64
DB_CACHE_SIZE initialization parameter, 8-64
DB_FILE_MULTIBLOCK_READ_COUNT initialization parameter, 8-64
default parameter settings, 8-50
DISK_ASYNCH_IO initialization parameter, 8-65
forcing for a session, 8-52
full table scans, 8-2
parallel execution *(continued)*
functions, 8-46
fundamental hardware requirements, 8-3
I/O, 8-2
I/O parameters, 8-64
in-memory, 8-18
index creation, 8-78
initializing parameters, 8-50
inter-operator parallelism, 8-4
intra-operator parallelism, 8-4
massively parallel systems, 8-2
Oracle RAC, 8-12
parallel load, 8-48
parallel propagation, 8-48
parallel recovery, 8-48
parallel replication, 8-48
parameters for establishing resource limits, 8-52
resource parameters, 8-59
symmetric multiprocessors, 8-2
TAPE_ASYNCH_IO initialization parameter, 8-65
tips for tuning, 8-72
tuning general parameters, 8-52
tuning parameters, 8-50
using, 8-1
when not to use, 8-2
parallel execution strategy
implementing, 8-73
PARALLEL hint
UPDATE, MERGE, and DELETE, 8-39
parallel partition-wise joins
performance considerations, 6-10
parallel processing
monitoring, 8-65
monitoring operating system statistics, 8-72
monitoring session statistics, 8-70
monitoring system statistics, 8-71
monitoring with GV$FILESTAT view, 8-66
monitoring with performance views, 8-66
parallel queries, 8-30
functions, 8-47
index-organized tables, 8-30
object types, 8-31
restrictions on object types, 8-31
parallel query
integrating with the automatic big table caching, 8-19
parallelism type, 8-30
parallel server resources
limiting for a consumer group, 8-23
parallel servers
asynchronous communication, 8-9
parallel SQL
allocating rows to parallel execution servers, 8-7

parallel SQL *(continued)*
distribution methods, *8-7*
instance groups, *8-12*
number of parallel execution servers, *8-10*
optimizer, *8-4*
parallel statement queue
about, *8-20*
grouping parallel statements, *8-28*
hints, *8-29*
limiting parallel server resources, *8-23*
managing for consumer groups, *8-21*
managing the order of dequeuing, *8-23*
managing with Oracle Database Resource Manager, *8-21*
NO_STATEMENT_QUEUING hint, *8-29*
PARALLEL_DEGREE_POLICY, *8-20*
sample scenario for managing parallel statements, *8-26*
setting order of parallel statements, *8-21*
specifying a critical consumer group, *8-25*
specifying a DOP limit for a consumer group, *8-25*
specifying a timeout for a consumer group, *8-24*
STATEMENT_QUEUING hint, *8-29*
using BEGIN_SQL_BLOCK to group statements, *8-28*
parallel update, *8-39*
parallel UPDATE statement, *8-39*
PARALLEL_DEGREE_POLICY initialization parameter
automatic degree of parallelism, *8-15*
controlling automatic DOP, *8-17*
PARALLEL_EXECUTION_MESSAGE_SIZE initialization parameter, *8-60*, *8-61*
PARALLEL_FORCE_LOCAL initialization parameter, *8-53*
PARALLEL_MAX_SERVERS initialization parameter, *8-53*
parallel execution, *8-53*
PARALLEL_MIN_PERCENT initialization parameter, *8-54*
PARALLEL_MIN_SERVERS initialization parameter, *8-10*, *8-55*
PARALLEL_MIN_TIME_THRESHOLD initialization parameter, *8-55*
PARALLEL_SERVERS_TARGET initialization parameter, *8-55*
parallelism
about, *8-3*
adaptive, *8-18*
degree, *8-13*
inter-operator, *8-4*
intra-operator, *8-4*
other types, *8-29*
parallel DDL statements, *8-29*
parallelism *(continued)*
parallel DML operations, *8-29*
parallel execution of functions, *8-29*
parallel queries, *8-29*
types, *8-29*
parallelization
methods for specifying precedence, *8-48*
rules for SQL operations, *8-48*
parameters
Automatic Data Optimization, *5-23*
partial indexes
on partitioned tables, *2-23*
partial partition-wise joins, *6-7*
about, *3-19*
composite, *3-22*
single-level, *3-20*
Partition Advisor
manageability, *2-16*
partition bound
range-partitioned tables, *4-3*
PARTITION BY HASH clause, *4-6*
PARTITION BY LIST clause, *4-8*
PARTITION BY RANGE clause, *4-3*
for composite-partitioned tables, *4-30*
PARTITION BY REFERENCE clause, *4-13*
PARTITION clause
for composite-partitioned tables, *4-30*
for hash partitions, *4-6*
for list partitions, *4-8*
for range partitions, *4-3*
partition exchange load
manageability, *6-14*
partition granules, *8-6*
partition maintenance operations, *7-5*, *7-6*
merging older partitions, *7-7*
moving older partitions, *7-7*
Online Transaction Processing (OLTP), *7-6*
removing old data, *7-6*
partition pruning
about, *3-1*
benefits, *3-1*
collection tables, *3-13*
data type conversions, *3-9*
dynamic, *3-4*
dynamic with bind variables, *3-4*
dynamic with nested loop joins, *3-7*
dynamic with star transformation, *3-5*
dynamic with subqueries, *3-5*
function calls, *3-12*
identifying, *3-3*
information for pruning, *3-2*
PARTITION_START, *3-3*
PARTITION_STOP, *3-3*
static, *3-3*
tips and considerations, *3-9*
with zone maps, *3-7*

PARTITION_START
partition pruning, 3-3
PARTITION_STOP
partition pruning, 3-3
partition-wise joins, 3-14
benefits, 6-5, 6-9
full, 3-14, 6-6
parallel execution, 6-10
partial, 3-19, 6-7
partition-wise operations, 3-14
partitioned external tables
creating, 4-18
partitioned indexes
about, 2-18
adding partitions, 4-62
administration, 4-1
composite partitions, 2-24
creating hash-partitioned global, 4-7
creating local index on composite partitioned table, 4-39
creating local index on hash partitioned table, 4-6
creating range partitions, 4-5
dropping partitions, 4-68
key compression, 4-23
maintenance operations, 4-49, 4-54
maintenance operations that can be performed, 4-45
modifying partition default attributes, 4-88
modifying real attributes of partitions, 4-90
moving partitions, 4-96
Online Transaction Processing (OLTP), 7-3
rebuilding index partitions, 4-96
renaming index partitions/subpartitions, 4-98
secondary indexes on index-organized tables, 4-26
splitting partitions, 4-108
views, 4-133
which type to use, 2-19
partitioned tables
adding partitions, 4-56
adding subpartitions, 4-59–4-61
administration, 4-1
coalescing partitions, 4-63
converting to from non-partitioned tables, 4-121
creating automatic list partitions, 4-10
creating composite, 4-30
creating composite interval, 4-31
creating composite list, 4-34
creating hash partitions, 4-6
creating interval partitions, 4-5
creating interval-hash partitions, 4-32
creating interval-list partitions, 4-33
creating interval-range partitions, 4-34
creating list partitions, 4-8
partitioned tables *(continued)*
creating list-hash partitions, 4-35
creating list-list partitions, 4-35
creating list-range partitions, 4-36
creating multi-column list partitions, 4-11
creating range partitions, 4-3, 4-5
creating range-hash partitions, 4-38
creating range-list partitions, 4-39
creating range-range partitions, 4-42
creating reference partitions, 4-13
data warehouses, 3-42
DISABLE ROW MOVEMENT, 4-1
dropping, 4-118
dropping partitions, 4-65
ENABLE ROW MOVEMENT, 4-1
exchanging partitions and subpartitions, 4-69
exchanging partitions of a referenced-partition table, 4-74
exchanging partitions with a cascade option, 4-78
exchanging subpartitions, 4-76–4-78
filtering maintenance operations, 4-53
FOR EXCHANGE WITH, 4-71
global indexes, 7-6
in-memory column store, 4-16
incremental statistics and partition exchange operations, 4-69
index-organized tables, 4-1, 4-26, 4-27
INTERVAL clause of CREATE TABLE, 4-5
interval-reference, 4-15
local indexes, 7-5
maintenance operations, 4-54
maintenance operations that can be performed, 4-45
maintenance operations with global indexes, 7-6
maintenance operations with local indexes, 7-5
marking indexes UNUSABLE, 4-98
merging partitions, 4-80
modifying default attributes, 4-87
modifying real attributes of partitions, 4-88
modifying real attributes of subpartitions, 4-89
moving partitions, 4-94
moving subpartitions, 4-95
multicolumn partitioning keys, 4-19
partition bound, 4-3
partitioning columns, 4-3
partitioning keys, 4-3
read-only status, 4-17
rebuilding index partitions, 4-96
redefining partitions online, 4-119
renaming partitions, 4-97
renaming subpartitions, 4-98
splitting partitions, 4-98
truncating partitions, 4-111

partitioned tables *(continued)*
truncating partitions with the cascade option, *4-117*
truncating subpartitions, *4-115*
updating global indexes automatically, *4-50*
views, *4-133*
partitioning
about, *1-1*
administration of indexes, *4-1*
administration of tables, *4-1*
advanced index compression, *3-37*
advantages, *1-1*
availability, *2-11*, *3-1*
basics, *2-2*
benefits, *2-10*
bitmap indexes, *3-40*
collections in XMLType and object data, *2-9*
composite, *2-13*
composite list-hash, *2-15*
composite list-list, *2-15*
composite list-range, *2-15*
composite range-hash, *2-14*
composite range-list, *2-15*
composite range-range, *2-14*
concepts, *2-1*
creating a partitioned index, *4-1*
creating a partitioned table, *4-1*
creating indexes on partitioned tables, *2-23*
data segment compression, *3-39*, *3-40*
data segment compression example, *3-41*
data warehouses, *6-1*
data warehouses and scalability, *6-1*
databases, and, *1-3*
default partition, *4-9*
default subpartition, *4-41*
deferred segments, *4-24*
EXCHANGE PARTITION clause, *4-72*
exchanging a hash partitioned table, *4-75*
exchanging a range partitioned table, *4-77*
exchanging interval partitions, *4-73*
extensions, *2-15*
global hash partitioned indexes, *2-21*
global indexes, *3-34*
global nonpartitioned indexes, *2-22*
global partitioned indexes, *2-20*
global range partitioned indexes, *2-21*
guidelines for indexes, *3-37*
hash, *2-13*
Hybrid Columnar Compression example, *3-41*
index-organized tables, *2-4*, *4-1*, *4-26*, *4-27*
indexes, *2-4*, *2-18*, *3-23*
Information Lifecycle Management, *2-5*
Information Lifecycle Management, and, *1-3*
interval, *2-16*
interval-hash, *4-32*
interval-list, *4-33*
partitioning *(continued)*
interval-range, *4-34*
key, *2-3*
key extensions, *2-17*
list, *2-13*, *4-90*, *4-91*
list-hash, *4-35*
list-list, *4-35*
list-range, *4-36*
LOB data, *2-6*
local indexes, *3-24*
local partitioned indexes, *2-19*
maintaining partitions, *4-54*
maintenance procedures for segment creation, *4-25*
manageability, *2-11*, *3-1*
manageability extensions, *2-16*
manageability with indexes, *6-15*
managing partitions, *3-34*
modifying attributes, *4-87*
modifying list partitions, *4-90*
modifying the strategy, *4-92*
nonprefixed indexes, *3-26*, *3-34*, *3-36*
Online Transaction Processing (OLTP), *7-1*
overview, *2-1*
partial indexes on partitioned tables, *2-23*
Partition Advisor, *2-16*
partition-wise joins, *2-11*
partitioned indexes on composite partitions, *2-24*
performance, *2-10*, *3-1*, *3-42*
performance considerations, *3-42*
performance considerations for composite, *3-45*
performance considerations for composite list-hash, *3-48*
performance considerations for composite list-list, *3-49*
performance considerations for composite list-range, *3-50*
performance considerations for composite range-hash, *3-46*
performance considerations for composite range-list, *3-47*
performance considerations for composite range-range, *3-48*
performance considerations for hash, *3-43*
performance considerations for interval, *3-51*
performance considerations for list, *3-45*
performance considerations for range, *3-51*
performance considerations for virtual columns, *3-52*
placement with striping, *10-5*
prefixed indexes, *3-25*, *3-34*
pruning, *2-10*, *3-1*
range, *2-13*
range-hash, *4-38*

partitioning *(continued)*
range-list, *4-39*
range-range, *4-42*
reference, *2-17*
removing data from tables, *6-15*
restrictions for multiple block sizes, *4-27*
segments, *4-23*
single-level, *2-12*
strategies, *2-11*, *3-42*
subpartition templates, *4-43*
system, *2-5*, *2-15*–*2-17*
tables, *2-3*, *2-4*
truncating segments, *4-24*
type of index to use, *2-19*
very large databases (VLDBs), and, *1-2*
virtual columns, *2-18*
partitioning and data compression
data warehouses, *6-15*
partitioning and materialized views
data warehouses, *6-12*
partitioning columns
range-partitioned tables, *4-3*
partitioning keys
range-partitioned tables, *4-3*
partitioning materialized views
data warehouses, *6-13*
partitioning of XMLIndex
binary XML tables, *4-30*
partitions, *1-1*
advanced index compression, *3-37*
equipartitioning
examples, *3-25*
local indexes, *3-24*
global indexes, *3-34*, *6-12*
guidelines for partitioning indexes, *3-37*
indexes, *3-23*
local indexes, *3-24*, *6-11*
nonprefixed indexes, *2-19*, *3-26*, *3-36*
on indexes, *6-10*
parallel DDL statements, *8-32*
physical attributes, *3-38*
prefixed indexes, *3-25*
PARTITIONS clause
for hash partitions, *4-6*
performance
DSS database, *8-37*
prefixed and nonprefixed indexes, *3-36*
very large databases (VLDBs), *10-3*
predicates
index partition pruning, *3-36*
prefixed indexes, *3-25*, *3-35*
partition pruning, *3-36*
process monitor process (PMON)
parallel DML process recovery, *8-41*
processes
memory contention in parallel processing, *8-53*
producer operations, *8-4*
pruning partitions
about, *3-1*
benefits, *3-1*
indexes and performance, *3-36*

Q

queries
ad hoc, *8-33*
queuing
parallel statements, *8-20*

R

range partitioning, *2-13*
creating tables using, *4-3*
index-organized tables, *4-26*
multicolumn partitioning keys, *4-19*
performance considerations, *3-42*, *3-51*
range-hash partitioning
creating tables using, *4-38*
subpartitioning template, *4-44*
range-list partitioning
creating tables using, *4-39*
subpartitioning template, *4-45*
range-partitioned tables
adding partitions, *4-57*
splitting partitions, *4-99*, *4-107*
range-range partitioning
creating tables using, *4-42*
read-only status
tables, partitions, and subpartitions, *4-17*
read-only tablespaces
performance considerations, *3-53*
REBUILD PARTITION clause, *4-96*, *4-97*
REBUILD UNUSABLE LOCAL INDEXES clause, *4-97*
recovery
parallel DML operations, *8-41*
reference partitioning
creating tables using, *4-13*
key extension, *2-17*
reference-partitioned tables
adding partitions, *4-61*
RENAME PARTITION clause, *4-98*
RENAME SUBPARTITION clause, *4-98*
replication
restrictions on parallel DML, *8-42*
resources
consumption, parameters affecting, *8-59*, *8-61*
limiting for users, *8-53*
limits, *8-53*

resources *(continued)*
parallel query usage, *8-59*
restrictions
parallel DDL statements, *8-32*
parallel DML operations, *8-42*
parallel DML operations and remote transactions, *8-45*
row movement clause for partitioned tables, *4-1*
row-level compression tiering
Automatic Data Optimization, *5-22*

S

scalability
batch jobs, *8-38*
parallel DML operations, *8-37*
scalability and manageability
very large databases (VLDBs), *10-7*
scans
parallel query on full table, *8-2*
segment-level compression tiering
Automatic Data Optimization, *5-21*
segments
creating on demand, *4-25*
deferred, *4-24*
partitioning, *4-23*
truncating, *4-24*
session statistics
monitoring for parallel processing, *8-70*
sessions
enabling parallel DML operations, *8-38*
SET INTERVAL clause, *4-58*
SHARED_POOL_SIZE initialization parameter, *8-56*
single-level partitioning, *2-12*
skewing parallel DML workload, *8-11*
SORT_AREA_SIZE initialization parameter
parallel execution, *8-61*
space management
MINIMUM EXTENT parameter, *8-34*
parallel DDL, *8-34*
SPLIT PARTITION clause, *4-57*, *4-98*
SPLIT PARTITION operations
optimizing, *4-110*
SPLIT SUBPARTITION operations
optimizing, *4-110*
splitting multiple partitions, *4-108*
splitting partitions and subpartitions, *4-98*
SQL statementsSQL statements
data flow operations, *8-4*
parallelizing, *8-4*
STATEMENT_QUEUING
parallel statement queuing hint, *8-29*
statistics
operating system, *8-72*
storage
fragmentation in parallel DDL, *8-34*
index partitions, *3-38*
STORAGE clause
parallel execution, *8-34*
storage management
very large databases (VLDBs), *10-1*
STORE IN clause
partitions, *4-39*
stripe and mirror everything
very large databases (VLDBs), *10-7*
striping
Information Lifecycle Management, *10-5*
partitioning placement, *10-5*
striping with Oracle ASM
very large databases (VLDBs), *10-4*
SUBPARTITION BY HASH clause
for composite-partitioned tables, *4-30*
SUBPARTITION clause, *4-59*–*4-61*, *4-104*
for composite-partitioned tables, *4-30*
subpartition templates, *4-43*
modifying, *4-53*
SUBPARTITIONS clause, *4-59*, *4-104*
for composite-partitioned tables, *4-30*
subqueries
in DDL statements, *8-33*
system monitor process (SMON)
parallel DML system recovery, *8-42*
system partitioning, *2-5*
system statistics
monitoring for parallel processing, *8-71*

T

table compression
partitioning, *4-23*
table queues
monitoring parallel processing, *8-68*
tables
creating and populating in parallel, *8-73*
creating composite partitioned, *4-30*
full partition-wise joins, *3-14*, *6-6*
historical, *8-38*
index-organized, partitioning, *4-25*
parallel creation, *8-33*
parallel DDL storage, *8-34*
partial partition-wise joins, *3-19*, *6-7*
partitioning, *2-3*
partitions, *1-1*
refreshing in data warehouse, *8-38*
STORAGE clause with parallel execution, *8-34*
summary, *8-33*
when to partition, *2-4*
tables for exchange
with partitioned tables, *4-71*

TAPE_ASYNCH_IO initialization parameter
parallel query, 8-65
temporal validity
creating a table with, 5-34
Temporal Validity
DBMS_FLASHBACK_ARCHIVE package
ENABLE_AT_VALID_TIME
procedure, 5-32
limitations, 5-35
managing data validity, 5-32
valid-time period, 5-32
temporary segments
parallel DDL, 8-34
time-based information
Information Lifecycle Management, 5-1
transactions
distributed and parallel DML restrictions, 8-45
triggers
restrictions, 8-44
restrictions on parallel DML, 8-42
TRUNCATE PARTITION clause, 4-111, 4-112
TRUNCATE SUBPARTITION clause, 4-115
truncating multiple partitions, 4-113
truncating partitions
cascade option, 4-117
marking indexes UNUSABLE, 4-111
truncating segments
partitioning, 4-24
two-phase commit, 8-62
types of parallelism, 8-29

U

UNION ALL
concurrent execution, 8-45
UPDATE GLOBAL INDEX clause
of ALTER TABLE, 4-50
UPDATE statement
parallel UPDATE statement, 8-39
updating indexes automatically, 4-50
user resources
limiting, 8-53

V

V$PQ_SESSTAT view
monitoring parallel processing, 8-67
V$PQ_TQSTAT view
monitoring parallel processing, 8-68
V$PX_BUFFER_ADVICE view
monitoring parallel processing, 8-66
V$PX_PROCESS view
monitoring parallel processing, 8-67
V$PX_PROCESS_SYSSTAT view
monitoring parallel processing, 8-67
V$PX_SESSION view
monitoring parallel processing, 8-66
V$PX_SESSTAT view
monitoring parallel processing, 8-66
V$RSRC_CONS_GROUP_HISTORY view
monitoring parallel processing, 8-68
V$RSRC_CONSUMER_GROUP view
monitoring parallel processing, 8-69
V$RSRC_PLAN view
monitoring parallel processing, 8-69
V$RSRC_PLAN_HISTORY view
monitoring parallel processing, 8-69
V$RSRC_SESSION_INFO view
parallel statement queuing metrics, 8-69
V$RSRCMGRMETRIC view
parallel statement queuing statistics, 8-69
V$SESSTAT view, 8-72
V$SYSSTAT view, 8-77
valid-time period
Temporal Validity, 5-32
very large databases (VLDBs)
about, 1-1
backing up and recovering, 9-1
backup tools, 9-4
backup types, 9-4
bigfile tablespaces, 10-6
database structures for recovering data, 9-3
hardware-based mirroring, 10-2
hardware-based striping, 10-4
high availability, 10-1
introduction, 1-1
mirroring with Oracle ASM, 10-3
Oracle Automatic Storage Management
settings, 10-7
Oracle Backup and Recovery, 9-2
Oracle Data Pump, 9-4, 9-5
Oracle Database File System, 10-6
Oracle Recovery Manager, 9-5
partitioning, and, 1-2
performance, 10-3
physical and logical backups, 9-4
RAID 0 striping, 10-4
RAID 1 mirroring, 10-2
RAID 5 mirroring, 10-2
RAID 5 striping, 10-4
RMAN, 9-4
scalability and manageability, 10-7
storage management, 10-1
stripe and mirror everything, 10-7
striping with Oracle ASM, 10-4
user-managed backups, 9-4, 9-5
views
parallel processing monitoring, 8-66
partitioned tables and indexes, 4-133
V$SESSTAT, 8-72
V$SYSSTAT, 8-77

views for ILM policies
Automatic Data Optimization, *5-25*
virtual column partitioning
performance considerations, *3-52*
virtual column-based partitioning
about, *4-22*
key extension, *2-18*
using for the subpartitioning key, *4-22*
VLDBs
very large databases, *1-1*

W

workloads
skewing, *8-11*

X

XMLType collections
partitioning, *4-28*
XMLType objects, *2-9*