Oracle8™ Parallel Server

Concepts & Administration

Release 8.0

November 14, 1997

Part No. A58238-01

ORACLE®

Enabling the Information Age™

Oracle8™ Parallel Server Concepts & Administration

Part No. A58238-01

Release 8.0

Primary Author: Rita Moran

Primary Contributors: Anjo Kolk, Graham Wood, Andrew Holdsworth

Contributors: Christina Anonuevo, Bill Bridge, Wilson Chan, Sandra Cheever, Carol Colrain, Mark Coyle, Connie Dialeris, Karl Dias, Jeff Fischer, John Frazzini, Anurag Gupta, Deepak Gupta, Mike Hartstein, Ken Jacobs, Ashok Joshi, Jonathan Klein, Jan Klokkers, Boris Klots, Tirthankar Lahiri, Bill Lee, Lefty Leverenz, Juan Loaiza, Sajjad Masud, Neil Macnaughton, Ravi Mirchandaney, Kant Patel, Mark Porter, Darryl Presley, Brian Quigley, Ann Rhee, Pat Ritto, Roger Sanders, Hari Sankar, Ekrem Soylemez, Vinay Srihari, Alex Tsukerman, Tak Wang, Betty Wu

Graphic Designer: Valarie Moore

Send Us Your Comments

Oracle8 Parallel Server Concepts & Administration, Release 8.0

Part No. A58238-01

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

- Did you find any errors?
- Is the information clearly presented?
- Do you need more information? If so, where?
- Are the examples correct? Do you need more examples?
- What features did you like most about this manual?

If you find any errors or have any other suggestions for improvement, please indicate the chapter, section, and page number (if available). You can send comments to us in the following ways:

- infodev@us.oracle.com
- FAX - 650-506-7228. Attn: Oracle8 Parallel Server
- postal service:
  Oracle Corporation
  Server Technologies Documentation
  500 Oracle Parkway, 4OP12
  Redwood Shores, CA 94065
  U.S.A.

If you would like a reply, please give your name, address, and telephone number below.

Contents

2 Successfully Implementing Parallel Processing

3 Parallel Hardware Architecture

5 Oracle Instance Architecture for the Parallel Server

6 Oracle Database Architecture for the Parallel Server

7 Overview of Locking Mechanisms

8 Integrated Distributed Lock Manager: Access to Resources

9 Parallel Cache Management Instance Locks

16 Ensuring IDLM Capacity for All Resources & Locks

17 Using Free List Groups to Partition Data

Part IV OPS System Maintenance Procedures

18 Administering Multiple Instances

19 Tuning the System to Optimize Performance

20 Monitoring Views & Tuning a Parallel Server

21 Backing Up the Database

22 Recovering the Database

23 Migrating from Single Instance to Parallel Server

Part V Reference

Preface

This manual describes the Oracle8 Parallel Server and supplements *Oracle8 Administrator's Guide* and *Oracle8 Concepts*.

This manual prepares you to successfully implement parallel processing by providing a thorough presentation of the concepts and procedures involved. Information in this manual applies to the Oracle8 Parallel Server running on all operating systems.

Note: *Oracle8 Parallel Server Concepts and Administration* contains information that describes the features and functionality of the Oracle8 and the Oracle8 Enterprise Edition products. Oracle8 and Oracle8 Enterprise Edition have the same basic features. However, several advanced features are available only with the Enterprise Edition, and some of these are optional. For example, to use application failover, you must have the Enterprise Edition and the Parallel Server Option.

For information about the differences between Oracle8 and the Oracle8 Enterprise Edition and the features and options that are available to you, please refer to *Getting to Know Oracle8 and the Oracle8 Enterprise Edition*.

Intended Audience

This manual is written for database administrators and application developers who work with a parallel server.

Structure

Part I: Parallel Processing Fundamentals

Chapter 1: Parallel Processing & Parallel Databases	This chapter introduces parallel processing and parallel database technologies, which offer great advantages for online transaction processing and decision support applications.
Chapter 2: Successfully Implementing Parallel Processing	This chapter explains how to attain the goals of speedup and scaleup, by effectively implementing parallel processing and parallel database technology.
Chapter 3: Parallel Hardware Architecture	This chapter describes the range of available hardware implementations which allow parallel processing, and surveys their advantages and disadvantages.

Part II: Oracle Parallel Server Concepts

Chapter 4: How Oracle Implements Parallel Processing	This chapter gives a high-level view of how the Oracle Parallel Server provides high performance parallel processing.
Chapter 5: Oracle Instance Architecture for the Parallel Server	This chapter explains features of Oracle multi-instance architecture which differ from an Oracle server in exclusive mode.
Chapter 6: Oracle Database Architecture for the Parallel Server	This chapter describes features of Oracle database architecture that pertain to the multiple instances of a parallel server.
Chapter 7: Overview of Locking Mechanisms	This chapter provides an overview of internal Oracle Parallel Server locking mechanisms.
Chapter 8: Integrated Distributed Lock Manager: Access to Resources	This chapter explains the role of the Integrated Distributed Lock Manager in controlling access to resources in a parallel server.

Chapter 9: Parallel Cache Management Instance Locks	This chapter provides a conceptual overview of PCM locks. The planning and allocation of PCM locks is one of the most complex tasks facing the Oracle Parallel Server database administrator.
Chapter 10: Non-PCM Instance Locks	This chapter describes some of the most common non-PCM instance locks.
Chapter 11: Space Management and Free List Groups	This chapter explains space management concepts.
Chapter 12: Application Analysis	This chapter provides a conceptual framework for optimizing OPS application design.

Part III: OPS System Development Procedures

Chapter 13: Designing a Database for Parallel Server	This chapter prescribes a general methodology for designing systems optimized for the Oracle Parallel Server.
Chapter 14: Creating a Database & Objects for Multiple Instances	This chapter describes aspects of database creation that are specific to a parallel server.
Chapter 15: Allocating PCM Instance Locks	This chapter explains how to allocate PCM locks to datafiles by specifying values for parameters in the initialization file of an instance.
Chapter 16: Ensuring IDLM Capacity for All Resources & Locks	This chapter explains how to reduce contention for shared resources and gain maximum performance from the parallel server by ensuring that adequate space is available in the Integrated Distributed Lock Manager for all the necessary locks and resources.
Chapter 17: Using Free List Groups to Partition Data	This chapter explains how to allocate free lists and free list groups to partition data. By doing this you can minimize contention for free space when using multiple instances.

Part IV: OPS System Maintenance Procedures

Chapter 18: Administering Multiple Instances	This chapter describes how to administer instances of a parallel server.
Chapter 19: Tuning the System to Optimize Performance	This chapter provides an overview of tuning issues.
Chapter 20: Monitoring Views & Tuning a Parallel Server	This chapter describes how to monitor performance of a parallel server by querying data dictionary views and dynamic performance views. It also explains how to tune a parallel server.
Chapter 21: Backing Up the Database	This chapter explains how to protect your data by archiving the online redo log files and periodically backing up the datafiles, the control file for your database, and the parameter files for your instances.
Chapter 22: Recovering the Database	This chapter describes Oracle recovery features on a parallel server.
Chapter 23: Migrating from Single Instance to Parallel Server	This chapter describes database conversion from a single instance Oracle8 database to a multi-instance Oracle8 database using the parallel server option.

Part V: Reference

Appendix A: Differences from Previous Versions	This appendix describes the differences between this release and previous releases of the Parallel Server Option.
Appendix B: Restrictions	This appendix lists restrictions for the parallel server.

Related Documents

This manual assumes you have already read *Oracle8 Concepts* and *Oracle8 Administrator's Guide*.

Conventions

This section explains the conventions used in this manual including the following:

- text
- syntax diagrams and notation
- code examples

Text

This section explains the conventions used within the text:

UPPERCASE Characters

Uppercase text is used to call attention to command keywords, object names, parameters, filenames, and so on.

For example, "If you create a private rollback segment, the name must be included in the ROLLBACK_SEGMENTS parameter of the parameter file."

Italicized Characters

Italicized words within text are book titles or emphasized words.

Syntax Diagrams and Notation

The syntax diagrams and notation in this manual show the syntax for SQL commands, functions, hints, and other elements. This section tells you how to read syntax diagrams and examples and write SQL statements based on them.

Keywords

Keywords are words that have special meanings in the SQL language. In the syntax diagrams in this manual, keywords appear in uppercase. You must use keywords in your SQL statements exactly as they appear in the syntax diagram, except that they can be either uppercase or lowercase. For example, you must use the CREATE keyword to begin your CREATE TABLE statements just as it appears in the CREATE TABLE syntax diagram.

Parameters

Parameters act as place holders in syntax diagrams. They appear in lowercase. Parameters are usually names of database objects, Oracle datatype names, or expressions. When you see a parameter in a syntax diagram, substitute an object or expression of the appropriate type in your SQL statement. For example, to write a CREATE TABLE statement, use the name of the table you want to create, such as EMP, in place of the *table* parameter in the syntax diagram. (Note that parameter names appear in italics in the text.)

This list shows parameters that appear in the syntax diagrams in this manual and examples of the values you might substitute for them in your statements:

Parameter	Description	Examples
table	The substitution value must be the name of an object of the type specified by the parameter.	emp
'text'	The substitution value must be a character literal in single quotes.	'Employee Records'
condition	The substitution value must be a condition that evaluates to TRUE or FALSE.	ename > 'A'
date *d*	The substitution value must be a date constant or an expression of DATE datatype.	TO_DATE ('01-Jan-1996', DD-MON-YYYY')
expr	The substitution value can be an expression of any datatype.	sal + 1000
integer	The substitution value must be an integer.	72
rowid	The substitution value must be an expression of datatype ROWID.	00000462.0001.0001
subquery	The substitution value must be a SELECT statement contained in another SQL statement.	SELECT ename FROM emp
statement_name *block_name*	The substitution value must be an identifier for a SQL statement or PL/SQL block.	s1 b1

Code Examples

SQL and SQL*Plus commands and statements appear separated from the text of paragraphs in a monospaced font. For example:

```
INSERT INTO emp (empno, ename) VALUES (1000, 'SMITH');
ALTER TABLESPACE users ADD DATAFILE 'users2.ora' SIZE 50K;
```

Example statements may include punctuation, such as commas or quotation marks. All punctuation in example statements is required. All example statements terminate with a semicolon (;). Depending on the application, a semicolon or other terminator may or may not be required to end a statement.

Uppercase words in example statements indicate the keywords within Oracle SQL. When you issue statements, however, keywords are not case sensitive.

Lowercase words in example statements indicate words supplied only for the context of the example. For example, lowercase words may indicate the name of a table, column, or file.

Your Comments Are Welcome

We value and appreciate your comments as an Oracle user and reader of the manuals. As we write, revise, and evaluate our documentation, your opinions are the most important input we receive. Please use the reader's comment form to tell us what you like and dislike about this manual or other Oracle manuals. If the form is not available, please use the following address:

- infodev@us.oracle.com
- FAX - 650-506-7228. Attn: Oracle8 Parallel Server
- postal service:

  Oracle Corporation
  Server Technologies Documentation
  500 Oracle Parkway, 4OP12
  Redwood Shores, CA 94065
  U.S.A.

Part I

Parallel Processing Fundamentals

1

Parallel Processing & Parallel Databases

This chapter introduces parallel processing and parallel database technologies, which offer great advantages for online transaction processing and decision support applications. The administrator's challenge is to selectively deploy this technology to fully use its multiprocessing power.

To do this successfully you must understand how multiprocessing works, what resources it requires, and when you can—and cannot—effectively apply it. This chapter answers the following questions:

- What Is Parallel Processing?
- What Is a Parallel Database?
- What Are the Key Elements of Parallel Processing?
- What Are the Benefits of Parallel Processing?
- What Are the Benefits of Parallel Database?
- How Does Parallel Execution Fit In?
- Is Parallel Server the Oracle Configuration You Need?

What Is Parallel Processing?

This section defines parallel processing and describes its use.

- Parallel Processing Defined
- Problems of Parallel Processing
- Characteristics of a Parallel System
- Parallel Processing for SMPs and MPPs
- Parallel Processing for Integrated Operations

Parallel Processing Defined

Parallel processing divides a large task into many smaller tasks, and executes the smaller tasks concurrently on several nodes. As a result, the larger task completes more quickly.

Note: A *node* is a separate processor, often on a separate machine. Multiple processors, however, can reside on a single machine.

Some tasks can be effectively divided, and thus are good candidates for parallel processing. Other tasks, however, do not lend themselves to this approach.

For example, in a bank with only one teller, all customers must form a single queue to be served. With two tellers, the task can be effectively split so that customers form two queues and are served twice as fast—or they can form a single queue to provide fairness. This is an instance in which parallel processing is an effective solution.

By contrast, if the bank manager must approve all loan requests, parallel processing will not necessarily speed up the flow of loans. No matter how many tellers are available to process loans, all the requests must form a single queue for bank manager approval. No amount of parallel processing can overcome this built-in bottleneck to the system.

Figure 1–1 and Figure 1–2 contrast sequential processing of a single parallel query with parallel processing of the same query.

Figure 1–1 Sequential Processing of a Large Task

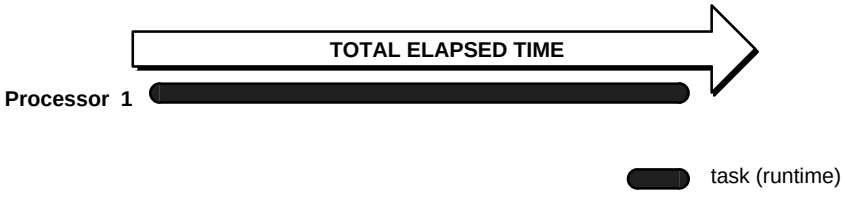
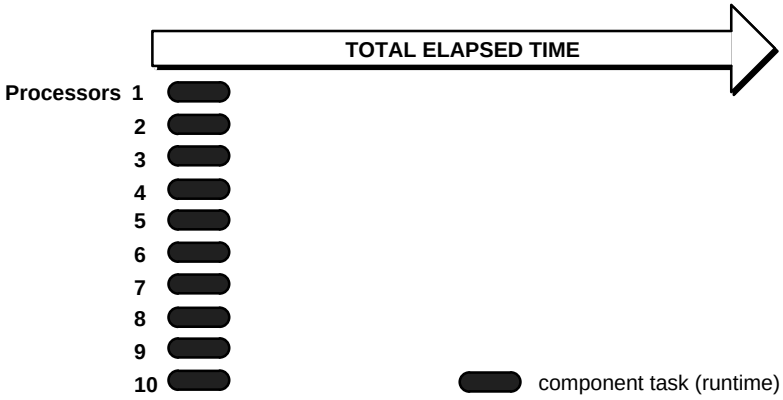


Figure 1–2 Parallel Processing: Executing Component Tasks in Parallel



In sequential processing, the query is executed as a single large task. In parallel processing, the query is divided into multiple smaller tasks, and each component task is executed on a separate node.

Figure 1–3 and Figure 1–4 contrast sequential processing with parallel processing of multiple independent tasks from an online transaction processing (OLTP) environment.

Figure 1–3 Sequential Processing of Multiple Independent Tasks

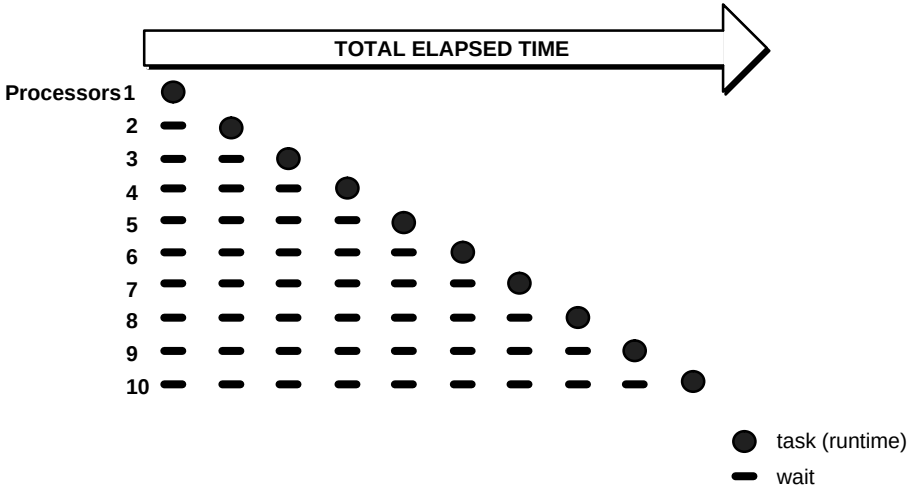
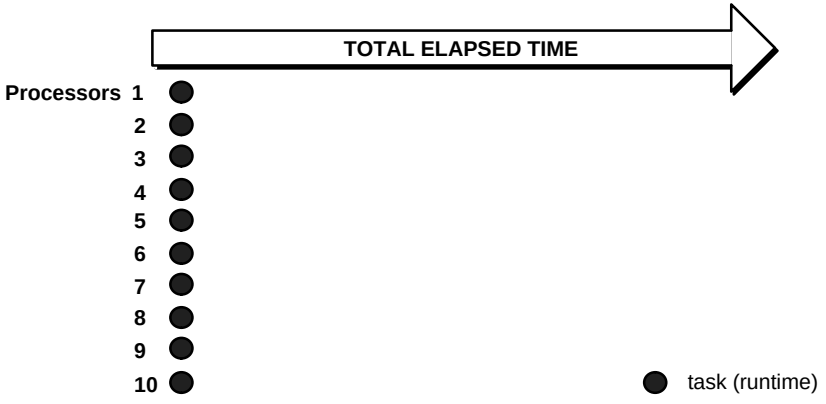


Figure 1–4 Parallel Processing: Executing Independent Tasks in Parallel



In sequential processing, independent tasks compete for a single resource. Only task 1 runs without having to wait. Task 2 must wait until task 1 has completed; task 3 must wait until tasks 1 and 2 have completed, and so on. (Although the figure shows the independent tasks as the same size, the size of the tasks will vary.) By contrast, in parallel processing (for example, a parallel server on a symmetric

multiprocessor), more CPU power is assigned to the tasks. Each independent task executes immediately on its own processor: no wait time is involved.

Problems of Parallel Processing

Effective implementation of parallel processing involves two challenges:

- structuring tasks so that certain tasks can execute at the same time (in parallel)
- preserving the sequencing of tasks which must be executed serially

Characteristics of a Parallel System

A parallel processing system has the following characteristics:

- Each processor in a system can perform tasks concurrently.
- Tasks may need to be synchronized.
- Nodes usually share resources, such as data, disks, and other devices.

Parallel Processing for SMPs and MPPs

Parallel processing architectures may support:

- clustered and massively parallel processing (MPP) hardware, in which each node has its own memory
- single memory systems—also known as symmetric multiprocessing (SMP) hardware, in which multiple processors use one memory resource

Clustered and MPP machines have multiple memories, with each CPU typically having its own memory. Such systems promise significant price/performance benefits by using commodity memory and bus components to eliminate memory bottlenecks.

Database management systems that support only one type of hardware limit the portability of applications, the potential to migrate applications to new hardware systems, and the scalability of applications. Oracle Parallel Server (OPS) exploits both clusters and MPP systems, and has no such limitations. Oracle without the Parallel Server Option exploits single CPU or SMP machines.

Parallel Processing for Integrated Operations

Parallel database software must effectively deploy the system's processing power to handle diverse applications: online transaction processing (OLTP) applications, decision support system (DSS) applications, as well as a mixed OLTP and DSS workload. OLTP applications are characterized by short transactions which have low CPU and I/O usage. DSS applications are characterized by long transactions, with high CPU and I/O usage.

Parallel database software is often specialized—usually to serve as query processors. Since they are designed to serve a single function, however, specialized servers do not provide a common foundation for integrated operations. These include online decision support, batch reporting, data warehousing, OLTP, distributed operations, and high availability systems. Specialized servers have been used most successfully in the area of very large databases: in DSS applications, for example.

Versatile parallel database software should offer excellent price/performance on open systems hardware, and be designed to serve a wide variety of enterprise computing needs. Features such as online backup, data replication, portability, interoperability, and support for a wide variety of client tools can enable a parallel server to support application integration, distributed operations, and mixed application workloads.

What Is a Parallel Database?

A variety of hardware architectures allow multiple computers to share access to data, software, or peripheral devices. A parallel database is designed to take advantage of such architectures by running multiple instances which “share” a single physical database. In appropriate applications, a parallel server can allow access to a single database by users on multiple machines, with increased performance.

A parallel server processes transactions in parallel by servicing a stream of transactions using multiple CPUs on different nodes, where each CPU processes an entire transaction. Using parallel data manipulation language you can have one transaction being performed by multiple nodes. This is an efficient approach because many applications consist of online insert and update transactions which tend to have short data access requirements. In addition to balancing the workload among CPUs, the parallel database provides for concurrent access to data and protects data integrity.

See Also: "Is Parallel Server the Oracle Configuration You Need?" on page 1-17 for a discussion of the available Oracle configurations.

What Are the Key Elements of Parallel Processing?

This section describes key elements of parallel processing:

- Speedup and Scaleup: the Goals of Parallel Processing
- Synchronization: A Critical Success Factor
- Locking
- Messaging

Speedup and Scaleup: the Goals of Parallel Processing

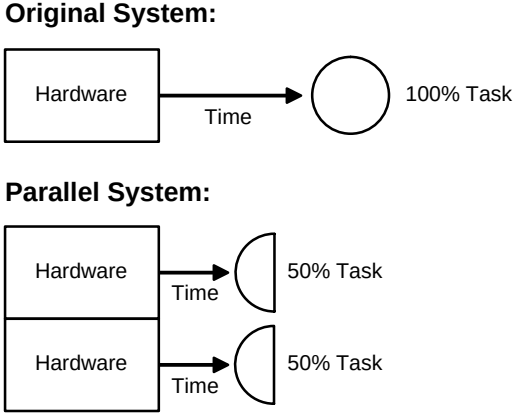
You can measure the performance goals of parallel processing in terms of two important properties:

- Speedup
- Scaleup

Speedup

Speedup is the extent to which more hardware can perform the same task in less time than the original system. With added hardware, speedup holds the task constant and measures time savings. Figure 1–5 shows how each parallel hardware system performs half of the original task in half the time required to perform it on a single system.

Figure 1–5 Speedup



With good speedup, additional processors reduce system response time. You can measure speedup using this formula:

$$Speedup = \frac{Time\_Original}{Time\_Parallel}$$

where

Time_Original is the elapsed time spent by a small system on the given task

Time_Parallel is the elapsed time spent by a larger, parallel system on the given task

For example, if the original system took 60 seconds to perform a task, and two parallel systems took 30 seconds, then the value of speedup would equal 2.

$$2 = \frac{60}{30}$$

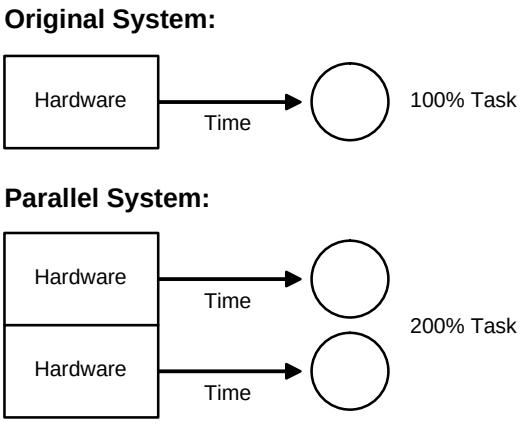
A value of *n*, where *n* times more hardware is used indicates the ideal of linear speedup: when twice as much hardware can perform the same task in half the time (or when three times as much hardware performs the same task in a third of the time, and so on).

Attention: For most OLTP applications, no speedup can be expected: only scaleup. The overhead due to synchronization may, in fact, cause speed-down.

Scaleup

Scaleup is the factor *m* that expresses how much more work can be done in the same time period by a system *n* times larger. With added hardware, a formula for scaleup holds the time constant, and measures the increased size of the job which can be done.

Figure 1–6 Scaleup



With good scaleup, if transaction volumes grow, you can keep response time constant by adding hardware resources such as CPUs.

You can measure scaleup using this formula:

$$Scaleup = \frac{Volume\_Parallel}{Volume\_Original}$$

where

Volume_Original	is the transaction volume processed in a given amount of time on a small system
Volume_Parallel	is the transaction volume processed in a given amount of time on a parallel system

For example, if the original system can process 100 transactions in a given amount of time, and the parallel system can process 200 transactions in this amount of time, then the value of scaleup would be equal to 2. That is, 200 / 100 = 2. A value of 2 indicates the ideal of linear scaleup: when twice as much hardware can process twice the data volume in the same amount of time.

Synchronization: A Critical Success Factor

Coordination of concurrent tasks is called *synchronization*. Synchronization is necessary for correctness. The key to successful parallel processing is to divide up tasks so that very little synchronization is necessary. The less synchronization necessary, the better the speedup and scaleup.

In parallel processing between nodes, a high-speed interconnect is required among the parallel processors. The overhead of this synchronization can be very expensive if a great deal of inter-node communication is necessary. For parallel processing within a node, messaging is not necessary: shared memory is used instead. Messaging and locking between nodes is handled by the Integrated Distributed Lock Manager (IDLM).

The amount of synchronization depends on the amount of resources and the number of users and tasks working on the resources. Little synchronization may be needed to coordinate a small number of concurrent tasks, but lots of synchronization may be necessary to coordinate many concurrent tasks.

Overhead

A great deal of time spent in synchronization indicates high contention for resources.

Attention: Too much time spent in synchronization can diminish the benefits of parallel processing. With less time spent in synchronization, better speedup and scaleup can be achieved.

Response time equals time spent waiting and time spent doing useful work. Table 1–1 illustrates how overhead increases as more concurrent processes are added. If 3 processes request a service at the same time, and they are served serially, then response time for process 1 is 1 second. Response time for process 2 is 2 seconds (waiting 1 second for process 1 to complete, then being serviced for 1 second). Response time for process 3 is 3 seconds (2 seconds waiting time plus 1 second service time).

Table 1–1 Increased Overhead with Increased Processes

Process Number	Service Time	Waiting Time	Response Time
1	1 second	0 seconds	1 second
2	1 second	1 second	2 seconds
3	1 second	2 seconds	3 seconds

One task, in fact, may require multiple messages. If tasks must continually wait to synchronize, then several messages may be needed per task.

Cost of Synchronization

While synchronization is a necessary element of parallel processing to preserve correctness, you need to manage its cost in terms of performance and system resources. Different kinds of parallel processing software may permit synchronization to be achieved, but a given approach may or may not be cost-effective.

Sometimes synchronization can be accomplished very cheaply. In other cases, however, the cost of synchronization may be too high. For example, if one table takes inserts from many nodes, a lot of synchronization is necessary. There will be high contention from the different nodes to insert into the same datablock: the datablock must be passed between the different nodes. This kind of synchronization can be done--but not efficiently.

See Also: Chapter 12, "Application Analysis"

Chapter 19, "Tuning the System to Optimize Performance"

Chapter 8, "Integrated Distributed Lock Manager: Access to Resources"

Locking

Locks are fundamentally a way of synchronizing tasks. Many different locking mechanisms are necessary to enable the synchronization of tasks required by parallel processing.

The Integrated Distributed Lock Manager (Integrated DLM, or IDLM) is the internal locking facility used with Oracle Parallel Server. It coordinates resource sharing between nodes running a parallel server. The instances of a parallel server use the Integrated Distributed Lock Manager to communicate with each other and coordinate modification of database resources. Each node operates independently of other nodes, except when contending for the same resource.

Note: In Oracle8 the Integrated Distributed Lock Manager facility replaces the external Distributed Lock Manager (DLM) which was used in previous releases. This enhancement frees Oracle performance from the limitations of external lock managers.

The IDLM allows applications to synchronize access to resources such as data, software, and peripheral devices, so that concurrent requests for the same resource are coordinated between applications running on different nodes.

The IDLM performs the following services for applications:

- keeps track of the current "ownership" of a resource
- accepts lock requests for resources from application processes
- notifies the requesting process when a lock on a resource is available
- gets access to a resource for a process

See Also: Chapter 7, "Overview of Locking Mechanisms", for a discussion of locking mechanisms internal to the Oracle database.

Chapter 8, "Integrated Distributed Lock Manager: Access to Resources"

Messaging

Parallel processing requires fast and efficient communication between nodes: a system with high bandwidth and low latency which efficiently communicates with the IDLM.

Bandwidth is the total size of messages which can be sent per second. *Latency* is the time (in seconds) it takes to place a message on the interconnect. Latency thus indicates the number of messages which can be put on the interconnect per second. An interconnect with high bandwidth is like a wide highway with many lanes to accommodate heavy traffic: the number of lanes affects the speed at which traffic can move. An interconnect with low latency is like a highway with an entrance ramp which permits vehicles to enter without delay: the cost of getting on the highway is low.

Most MPP systems and clusters are being designed with networks that have reasonably high bandwidth. Latency, on the other hand, is an operating system issue predominantly having to do with software. MPP systems and most clusters characteristically use interconnects with high bandwidth and low latency; other clusters may use Ethernet connections with relatively low bandwidth and high latency.

What Are the Benefits of Parallel Processing?

Parallel processing can benefit certain kinds of applications by providing:

- Enhanced Throughput: Scaleup
- Improved Response Time: Speedup

Improved response time can be achieved either by breaking up a large task into smaller components or by reducing wait time, as was shown in Figure 1–3.

Table 1–2 shows which types of workload can attain speedup and scaleup with properly implemented parallel processing.

Table 1–2 Speedup and Scaleup with Different Workloads

Workload	Speedup	Scaleup
OLTP	No	Yes
DSS	Yes	Yes
Batch (Mixed)	Possible	Yes
Parallel Query	Yes	Yes

Enhanced Throughput: Scaleup

If tasks can run independently of one another, they can be distributed to different CPUs or nodes and there will be a scaleup: more processes will be able to run through the database in the same amount of time.

If processes can run ten times faster, then the system can accomplish ten times more in the original amount of time. The parallel query feature, for example, permits scaleup: a system might maintain the same response time if the data queried increases tenfold, or if more users can be served. Oracle Parallel Server without the parallel query feature also permits scaleup, but by running the same query sequentially on different nodes.

With a mixed workload of DSS, OLTP, and reporting applications, scaleup can be achieved by running multiple programs on different nodes. Speedup can also be achieved if you rewrite the batch programs, splitting them into a number of parallel streams to take advantage of the multiple CPUs which are now available.

Improved Response Time: Speedup

DSS applications and parallel query can attain speedup with parallel processing: each transaction can run faster.

For OLTP applications, however, no speedup can be expected: only scaleup. With OLTP applications each process is independent: even with parallel processing, each insert or update on an order table will still run at the same speed. In fact, the overhead due to synchronization may cause a slight speed-down. Since each of the operations being done is small, it is inappropriate to attempt to parallelize them; the overhead would be greater than the benefit.

Speedup can also be achieved with batch processing, but the degree of speedup depends on the synchronization between tasks.

What Are the Benefits of Parallel Database?

Parallel database technology can benefit certain kinds of applications by enabling:

- Higher Performance
- Higher Availability
- Greater Flexibility
- More Users

Higher Performance

With more CPUs available to an application, higher speedup and scaleup can be attained. The improvement in performance depends on the degree of inter-node locking and synchronization activities. Each lock operation is processor and message intensive; there can be a lot of latency. The volume of lock operations and database contention, as well as the throughput and performance of the IDLM, ultimately determine the scalability of the system.

Higher Availability

Nodes are isolated from each other, so a failure at one node does not bring the whole system down. The remaining nodes can recover the failed node and continue to provide data access to users. This means that data is much more available than it would be with a single node upon node failure, and amounts to significantly higher availability of the database.

Greater Flexibility

An Oracle Parallel Server environment is extremely flexible. Instances can be allocated or deallocated as necessary. When there is high demand for the database, more instances can be temporarily allocated. The instances can be deallocated and used for other purposes once they are no longer necessary.

More Users

Parallel database technology can make it possible to overcome memory limits, enabling a single system to serve thousands of users.

Is Parallel Server the Oracle Configuration You Need?

This section describes the following Oracle configurations, which can deliver high performance for different types of applications:

- Single Instance with Exclusive Access
- Multi-Instance Database System
- Distributed Database System
- Client-Server Systems

The parallel server is one of several Oracle options which provide a high-performance relational database serving many users. These configurations can be combined to suit your needs. A parallel server can be one of several servers in a distributed database environment, and the client-server configuration can combine various Oracle configurations into a hybrid system to meet specific application requirements.

Note: Support for any given Oracle configuration is platform-dependent; check to confirm that your platform supports the configuration you want.

For optimal performance, configure your system according to your particular application requirements and available resources, then design and tune the database and applications to make the best use of the configuration. Consider also the migration of existing hardware or software to the new system or to future systems.

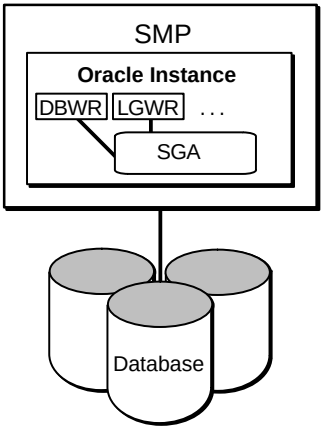
The following sections help you determine which Oracle configuration best meets your needs.

See Also: Chapter 3, “Parallel Hardware Architecture”

Single Instance with Exclusive Access

Figure 1–7 illustrates a single instance database system running on a symmetric multiprocessor (SMP). The database itself is located on a set of disks.

Figure 1–7 Single Instance Database System



A single instance accessing a single database can improve performance by running on a larger computer. A large single computer does not require coordination between several nodes and generally performs better than two small computers in a multinode system. However, two small computers often cost less than one large one.

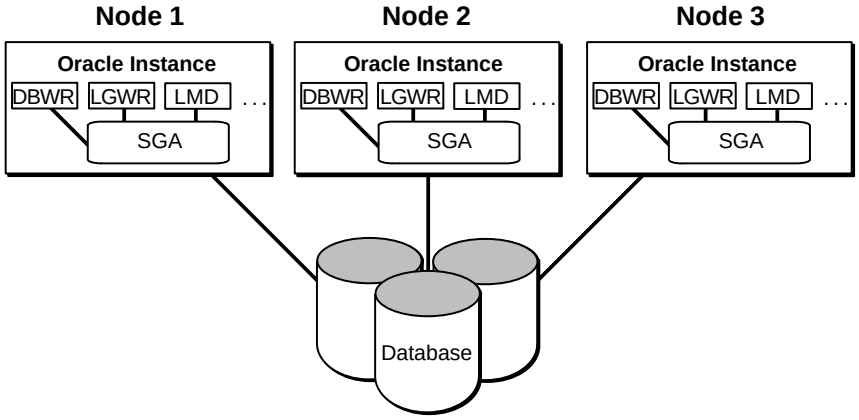
The cost of redesigning and tuning your database and applications for the Parallel Server Option might be significant if you want to migrate from a single computer to a multinode system. In situations like this, consider whether, a larger single computer might be a better solution than moving to a parallel server.

See Also: *Oracle8 Concepts* for complete information about single instance Oracle.

Multi-Instance Database System

Oracle with the Parallel Server Option running on a cluster or MPP is called a multi-instance database system, illustrated in Figure 1–8. This is an excellent solution for applications which can be configured to minimize the passing of data between instances on different nodes.

Figure 1–8 Multi-Instance Database System



Note that this database system requires the LMD process on each instance. These processes communicate with each other to coordinate global locking.

In a parallel server, instances are decoupled from databases. In exclusive mode, there is a one-to-one correspondence of instance to database. In shared (parallel) mode, however, there can be many instances to a single database.

In general, any single application performs best when it has exclusive access to a database on a larger system, as compared with its performance on a smaller node of a multinode environment. This is because the cost of synchronization may become too high if you go to a multinode environment. The performance difference depends on characteristics of that application and all other applications sharing access to the database.

Applications with one or both of the following characteristics are well suited to run on separate instances of a parallel server:

- applications which primarily query data
- applications which either change disjoint groups of datablocks or change the same datablocks at different times

See Also: "Enabling and Disabling Parallel Server" on page 4-2

Chapter 8, “Integrated Distributed Lock Manager: Access to Resources”

Oracle8 Concepts for more information on the DBWR, LGWR, and LMD background processes.

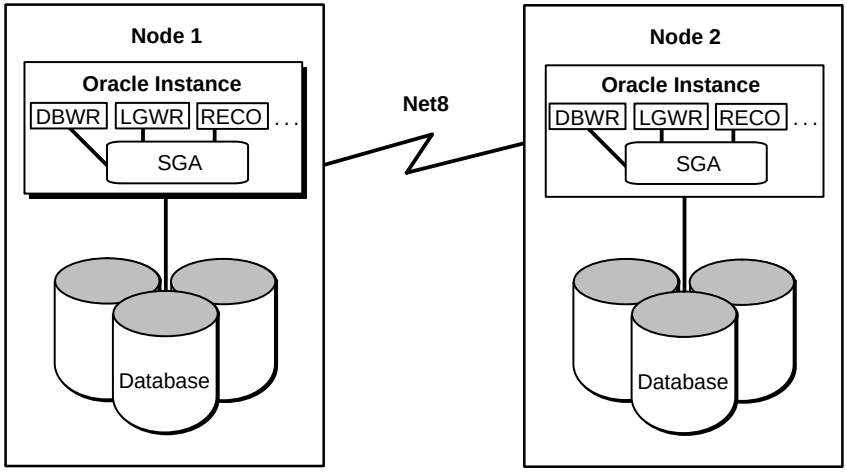
Distributed Database System

Several Oracle servers and databases can be linked to form a *distributed database* system. This configuration includes multiple databases, each of which is accessed directly by a single server and can be accessed indirectly by other instances through server-to-server cooperation. Each node can be used for database processing, but the data is permanently partitioned among the nodes. A parallel server, in contrast, has multiple instances which share direct access to one database.

Note: Oracle Parallel Server can be one of the constituents of a distributed database.

Figure 1–9 illustrates a distributed database system. This database system requires the RECO background process on each instance. There is no LCK, LMON, or LMD background process because this is not an Oracle Parallel Server configuration, and the Integrated Distributed Lock Manager is not needed.

Figure 1–9 Distributed Database System

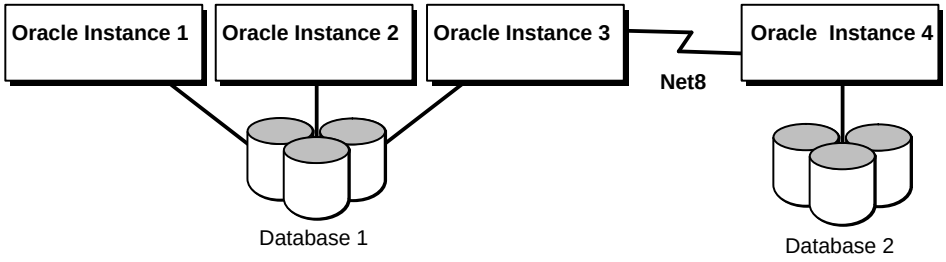


The multiple databases of a distributed system can be treated as one logical database, because servers can access remote databases transparently, using Net8.

If your data can be partitioned into multiple databases with minimal overlap, you can use a distributed database system instead of a parallel server, sharing data between the databases with Net8. A parallel server provides automatic data sharing among nodes through the common database.

A distributed database system allows you to keep your data at several widely separated sites. Users can access data from databases which are geographically distant, as long as network connections exist between the separate nodes. A parallel server requires all data to be at a single site because of the requirement for low latency, high bandwidth communication between nodes, but it can also be part of a distributed database system. Such a system is illustrated in Figure 1–10.

Figure 1–10 Oracle Parallel Server as Part of a Distributed Database



Multiple databases require separate database administration, and a distributed database system requires coordinated administration of the databases and network protocols. A parallel server can consolidate several databases to simplify administrative tasks.

Multiple databases can provide greater availability than a single instance accessing a single database, because an instance failure in a distributed database system does not prevent access to data in the other databases: only the database owned by the failed instance is inaccessible. A parallel server, however, allows continued access to all data when one instance fails, including data which was accessed by the instance running on the failed node.

A parallel server accessing a single consolidated database can avoid the need for distributed updates, inserts, or deletions and more expensive two-phase commits by allowing a transaction on any node to write to multiple tables simultaneously, regardless of which nodes usually write to those tables.

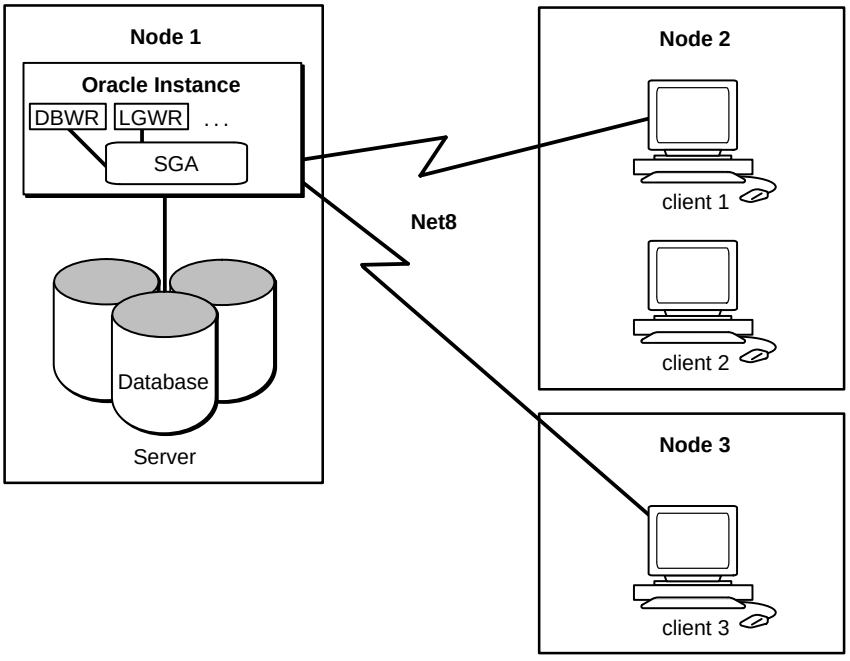
See Also: *Oracle8 Distributed Database Systems* for complete information about Oracle distributed database features.

Client-Server Systems

Any of the Oracle configurations can run in a client-server environment. In Oracle, a client application runs on a remote computer, using Net8 to access an Oracle server through a network. The performance of this configuration is typically limited to the power of the single server node.

Figure 1–11 illustrates an Oracle client-server system.

Figure 1–11 Client-Server System



Note: Client-server processing is suitable for any Oracle configuration. Check your Oracle platform-specific documentation to see whether it is implemented on your platform.

The client-server configuration allows you to off-load processing from the computer which runs an Oracle server. If you have too many applications running on one machine, you can off-load them to improve performance. However, if your database server is reaching its processing limits you might want to move either to a larger machine or to a multinode system.

For compute-intensive applications, you could run some applications on one node of a multinode system while running Oracle and other applications on another node, or on several other nodes. In this way you could effectively use various nodes of a parallel machine as client nodes, and one as a server node.

If the database consists of several distinct high-throughput parts, a parallel server running on high-performance nodes can provide quick processing for each part of the database while also handling occasional access across parts.

Remember that a client-server configuration requires that all communications between the client application and the database take place over the network. This may not be appropriate where a very high volume of such communications is required--as in many batch applications.

See Also: "Client-Server Architecture" in *Oracle8 Concepts*

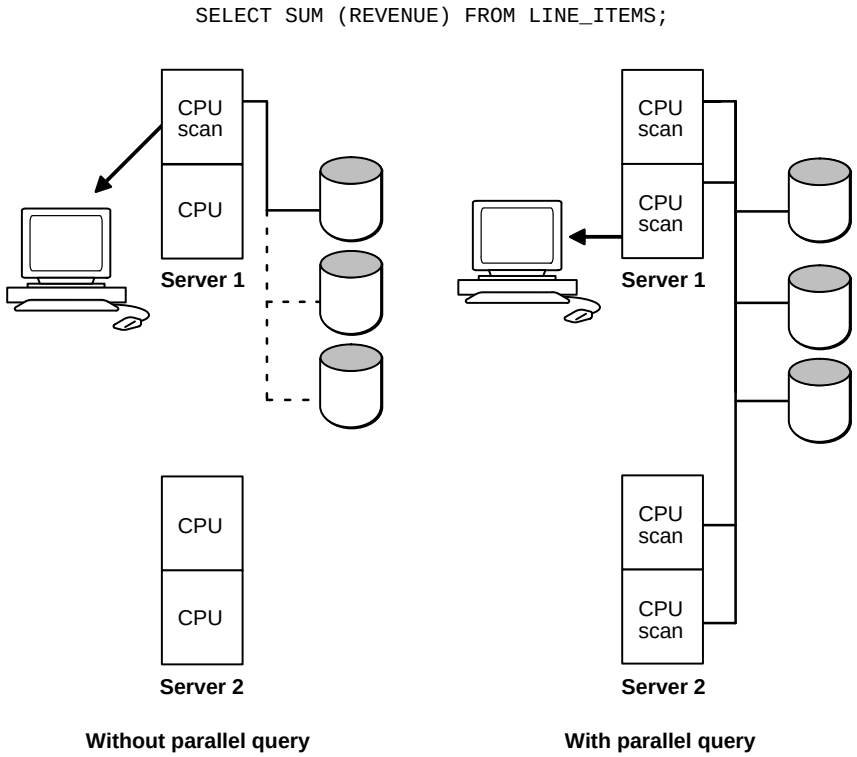
How Does Parallel Execution Fit In?

With its parallel execution features, Oracle can divide the work of processing certain types of SQL statements among multiple query server processes.

Oracle Parallel Server provides the framework for parallel execution to work between nodes. Parallel execution features behave the same way in Oracle with or without the Parallel Server Option. The only difference is that OPS enables multiple nodes to execute on behalf of a single query or other parallel operation.

In some applications (notably data warehousing applications), an individual query consumes a great deal of CPU resource and disk I/O, unlike most online insert or update transactions. To take advantage of multiprocessing systems, the data server must parallelize individual queries into units of work which can be processed simultaneously. Figure 1–12 shows an example of parallel query processing.

Figure 1–12 Example of Parallel Query Processing



If the query were not processed in parallel, disks would be read serially with a single I/O. A single CPU would have to scan all rows in the LINE_ITEMS table and total the revenues across all rows. With the query parallelized, disks are read in parallel, with multiple I/Os. Several CPUs can each scan a part of the table in parallel, and aggregate the results. Parallel query benefits not only from multiple CPUs but also from more of the available I/O bandwidth.

See Also: *Oracle8 Concepts* and *Oracle8 Tuning* for a detailed treatment of parallel execution.

2

Successfully Implementing Parallel Processing

There is an old network saying: Bandwidth problems can be cured with money. Latency problems are harder because the speed of light is fixed—you can't bribe God.

— David Clark, MIT

To attain the goals of speedup and scaleup, you must effectively implement parallel processing and parallel database technology. This means designing and building your system for parallel processing from the start. This chapter covers the following issues:

- The Four Levels of Scalability You Need
- When Is Parallel Processing Advantageous?
- When Is Parallel Processing Not Advantageous?
- Guidelines for Effective Partitioning
- Common Misconceptions about Parallel Processing

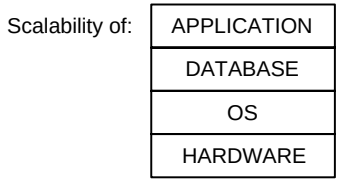
The Four Levels of Scalability You Need

Successful implementation of parallel processing and parallel database requires optimal scalability on four levels:

- Scalability of Hardware
- Scalability of Operating System
- Scalability of Database Management System
- Scalability of Application

Attention: An inappropriately designed application may not fully utilize the potential scalability of the system. Likewise, no matter how well your application scales, you will not get the desired performance if you try to run it on hardware that does not scale.

Figure 2–1 Levels of Scalability



Scalability of Hardware

Interconnect is key to hardware scalability. Every system must have some means of connecting the CPUs, whether it be a high speed bus or a low speed Ethernet connection. Bandwidth and latency of the interconnect determine the scalability of the hardware.

See Also: "Required Hardware and Operating System Software" on page 3-3.

Bandwidth and Latency

Most interconnects have sufficient bandwidth. A high bandwidth may, in fact, disguise high latency.

Hardware scalability depends heavily on very low latency. Lock coordination traffic communication is characterized by a large number of very small messages among the LMD processes.

Consider the difference between conveying a hundred passengers on a single bus, compared to a hundred individual cars. In the latter case, efficiency depends largely upon the capacity for cars to quickly enter and exit the highway.

Other operations between nodes, such as parallel query, rely on high bandwidth.

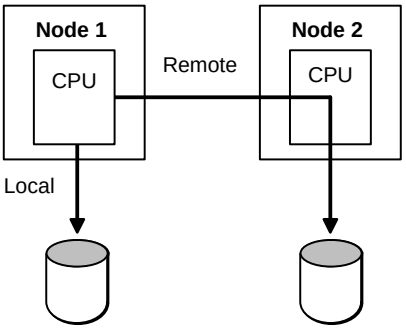
Disk Input and Output

Local I/Os are faster than remote I/Os (those which occur between nodes). If a great deal of remote I/O is needed, the system loses scalability. In this case you can partition data so that the data is local. Figure 2–2 illustrates the difference.

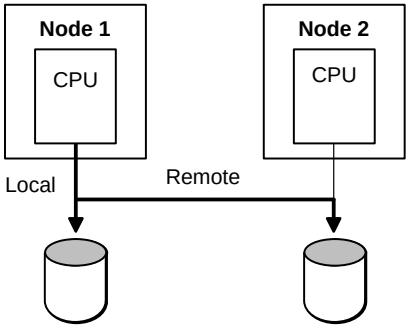
Note: Various clustering implementations are available from different hardware vendors. On shared disk clusters with dual ported controllers, there is the same latency from all nodes.

Figure 2–2 Local and Remote I/O on Shared Nothing and Shared Disk

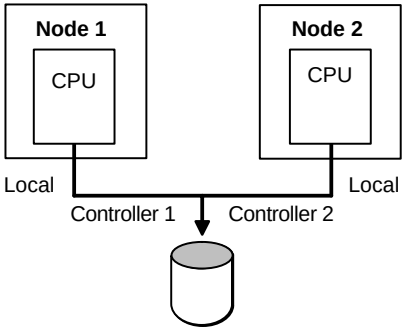
Shared Nothing



Shared Disk Cluster



Shared Disk Cluster with Dual Ported Controller

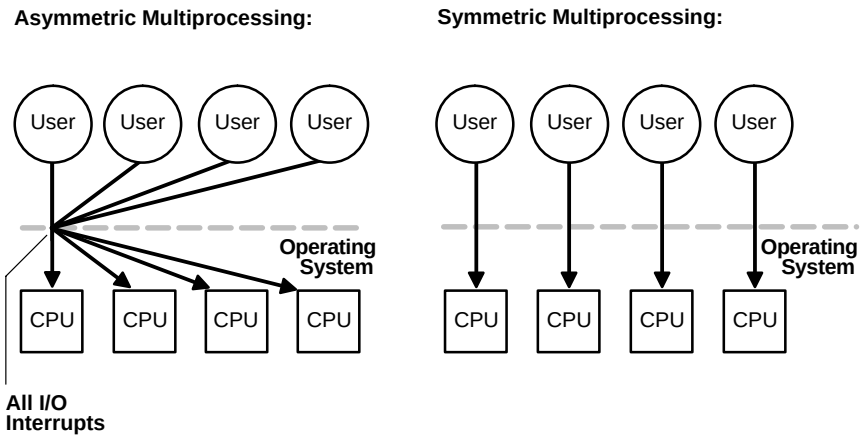


Scalability of Operating System

The ultimate scalability of your system also depends upon the scalability of the operating system. This section explains how to analyze this factor.

Software scalability can be an important issue if one node is a shared memory system (that is, a system in which multiple CPUs connect to a symmetric multiprocessor single memory). Methods of synchronization in the operating system can determine the scalability of the system. In asymmetrical multiprocessing, for example, only a single CPU can handle I/O interrupts. Consider a system in which multiple user processes all need to request a resource from the operating system:

Figure 2–3 Asymmetric Multiprocessing vs. Symmetric Multiprocessing



Here, the potential scalability of the hardware is lost because the operating system can only process one resource request at a time. Each time one request enters the operating system, a lock is held to exclude the others. In symmetrical multiprocessing, by contrast, there is no such bottleneck.

Scalability of Database Management System

An important distinction in parallel server architectures is internal versus external parallelism; this has a strong effect on scalability. The key difference is whether the object-relational database management system (ORDBMS) parallelizes the query, or an external process parallelizes the query.

Disk affinity can improve performance by ensuring that nodes mainly access local, rather than remote, devices. An efficient synchronization mechanism enables better speedup and scaleup.

See Also: "Disk Affinity" on page 4-9.

“Parallel Execution” in *Oracle8 Tuning*.

Scalability of Application

Application design is key to taking advantage of the scalability of the other elements of the system.

Attention: Applications must be specifically designed to be scalable!

No matter how scalable the hardware, software, and database may be, a table with only one row which every node is updating will synchronize on one datablock. Consider the process of generating a unique sequence number:

```
UPDATE ORDER_NUM
SET NEXT_ORDER_NUM = NEXT_ORDER_NUM + 1;
COMMIT;
```

Every node which needs to update this sequence number will have to wait to access the same row of this table: the situation is inherently unscalable. A better approach would be to use sequences to improve scalability:

```
INSERT INTO ORDERS VALUES
   (order_sequence.nextval, ... )
```

Note: Clients must be connected to server machines in a scalable manner: this means that your network must also be scalable!

See Also: Chapter 13, “Designing a Database for Parallel Server”.

Chapter 12, “Application Analysis”.

When Is Parallel Processing Advantageous?

This section describes applications which commonly benefit from a parallel server.

- Data Warehousing Applications
- Applications in Which Updated Data Blocks Do Not Overlap
- Failover and High Availability
- Summary

Data Warehousing Applications

Data warehousing applications which infrequently update, insert, or delete data are often appropriate for the parallel server. Query-intensive applications and other applications with low update activity can access the database through different instances with little additional overhead.

If the data blocks are not modified, multiple copies of the blocks can be read into the Oracle buffer caches on several nodes and queried without additional I/O or lock operations. As long as the instances are only reading data and not modifying it, a block can be read into multiple buffer caches and one instance never has to write the block to disk before another instance can read it.

Decision support applications are good candidates for a parallel server because they only occasionally modify data, as in a database of financial transactions which is mostly accessed by queries during the day and is updated during off-peak hours.

Applications in Which Updated Data Blocks Do Not Overlap

Applications which either update disjoint data blocks or update the same data blocks at different times are also well suited to the parallel server. Applications can run efficiently on a parallel server if the set of data blocks regularly updated by one instance does not overlap with the set of blocks simultaneously updated by other instances. An example is a time-sharing environment where each user primarily owns and uses one set of tables.

An instance which needs to update blocks held in its buffer cache must hold one or more instance locks in exclusive mode while modifying those buffers. You should tune a parallel server and the applications which run on it, so as to reduce contention for instance locks.

OLTP with Partitioned Data

Online transaction processing applications which modify disjoint sets of data benefit the most from the parallel server architecture. One example is a branch banking system where each branch (node) accesses its own accounts and only occasionally accesses accounts from other branches.

OLTP with Random Access to a Large Database

Applications which access a database in a mostly random pattern also benefit from the parallel server architecture, if the database is significantly larger than any node's buffer cache. One example is a Department of Motor Vehicles system where individual records are unlikely to be accessed by different nodes at the same time. Another example would be archived tax records or research data. In cases like these, most of the accesses would result in I/O even if the instance had exclusive access to the database. Oracle features such as fine grained locking can further improve performance of such applications.

Departmentalized Applications

Applications which primarily modify different tables in the same database are also suitable for Oracle Parallel Server. An example is a system where one node is dedicated to inventory processing, another is dedicated to personnel processing, and a third is dedicated to sales processing. Note that there is only one database to administer, not three.

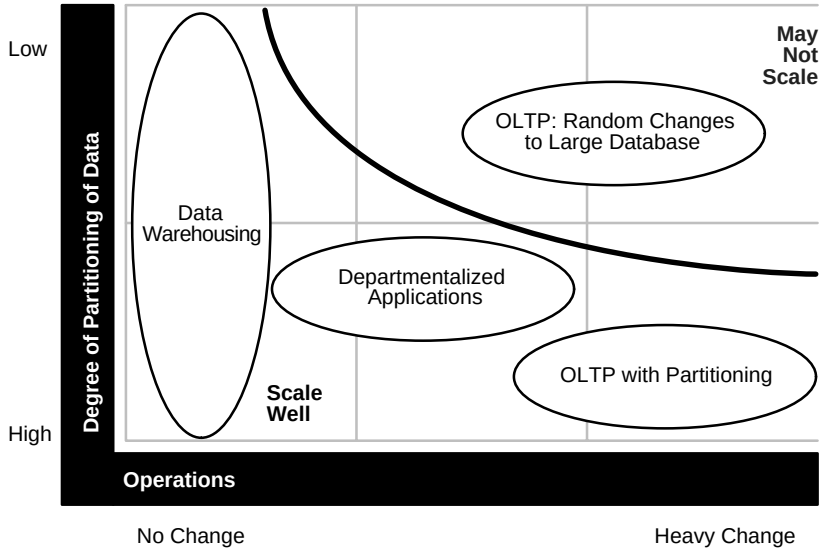
Failover and High Availability

Applications which require high availability benefit from the Oracle parallel server's failover capability. If the connection to the database is broken, applications can automatically reconnect.

Summary

Figure 2–4 illustrates the relative scalability of different kinds of applications. Online transaction processing applications which have a very high volume of inserts or updates from multiple nodes on the same set of data may require partitioning if they are to scale well. OLTP applications with a very low insert and update load may not require partitioning at all to be successful.

Figure 2–4 Scalability of Applications



When Is Parallel Processing Not Advantageous?

The following guidelines describe situations in which parallel processing is *not* advantageous.

- In general, parallel processing ceases to be advantageous when the cost of synchronization becomes too high and therefore the throughput decreases.

  If many users on a large number of nodes are modifying a small set of data, then synchronization is likely to be very high. However, if they are just reading the data then no synchronization is required.

- Parallel processing is not advantageous when there is contention between instances on a single block or row.

  For example, it would not be effective to use a table with one row used primarily as a sequence numbering tool. Such a table would be a bottleneck because every process would have to select the row, update it, and release it sequentially.

Guidelines for Effective Partitioning

This section provides general guidelines to make partitioning decisions which will decrease synchronization and add to your system's performance.

- Overview
- Vertical Partitioning
- Horizontal Partitioning

Overview

You can partition any of the three elements of processing, depending on function, location, and so on, such that they do not interfere with each other. These elements are:

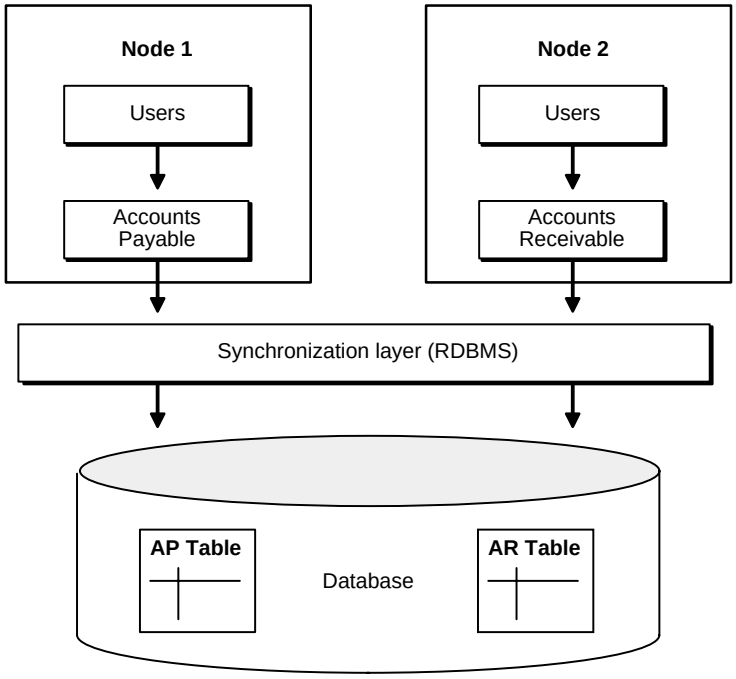
- users
- applications
- data

You can partition data, based on groups of users who access it; partition applications into groups which access the same data. You can also consider partitioning by location (geographic partitioning).

Vertical Partitioning

With vertical partitioning, a large number of tasks can run on a large number of resources without much synchronization. Figure 2–5 illustrates the concept of vertical partitioning.

Figure 2–5 Vertical Partitioning



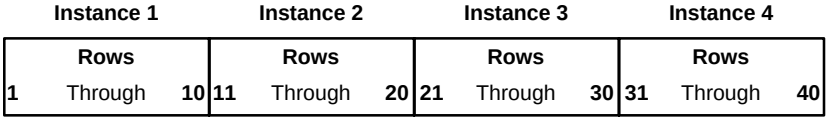
Here, a company's accounts payable and accounts receivable functions have been partitioned by users, application, and data. They have been placed on two separate nodes. Here, most of the synchronization takes place on the same node, which is very efficient. The cost of synchronization on the local node is cheaper than the cost of synchronization between nodes.

Partition tasks on a subset of resources to reduce synchronization. When you partition, you have a smaller set of tasks working on a smaller resource.

Horizontal Partitioning

To illustrate the concept of horizontal partitioning, Figure 2–6 represents the rows of a stock table. If the Oracle Parallel Server has four instances on a single node, then the data can be partitioned such that each instance accesses only a subset of the data.

Figure 2–6 Horizontal Partitioning



In this example, very little synchronization is necessary because the instances access different sets of rows. Similarly, users partitioned by location can often run almost independently: very little synchronization is necessary if the users do not access the same data.

Common Misconceptions about Parallel Processing

Various mistaken notions can lead to unrealistic expectations about parallel processing. Consider the following:

- Do not assume that you can switch to parallel processing and it will automatically work the way you expect. A good deal of application tuning and database design and tuning is required.
- Scalability is not determined just by the number of nodes or CPUs involved, but also by interconnect (bandwidth/latency) and by the amount and cost of synchronization.

  In some applications a single synchronization may be so expensive as to constitute a problem; in other applications, many cheap synchronizations may be perfectly acceptable.
- Just because you have parallel processing does not mean you automatically have higher availability: this depends on the system architecture.

  For example, on some MPP systems if one of the CPUs dies, the whole machine dies. On a cluster, by contrast, if one of the nodes dies the other nodes survive.
- All applications may not have been designed to scale up effectively.

3

Parallel Hardware Architecture

The parallel database server can use various machine architectures which allow parallel processing. This chapter describes the range of available hardware implementations and surveys their advantages and disadvantages.

- Overview
- Required Hardware and Operating System Software
- Shared Memory Systems
- Shared Disk Systems
- Shared Nothing Systems
- Shared Nothing /Shared Disk Combined Systems

Overview

This section covers the following topics:

- Parallel Processing Hardware Implementations
- Application Profiles

Oracle configurations support parallel processing within a machine, between machines, and between nodes. There is no advantage to running Oracle Parallel Server on a single node and a single system image--you would incur overhead and receive no benefit. With standard Oracle you do not have to do anything special on shared memory configurations to take advantage of some parallel processing capabilities.

Although this manual focuses on Oracle Parallel Server with shared nothing / shared disk architecture, the application design issues discussed in this book may also be relevant to standard Oracle systems.

Parallel Processing Hardware Implementations

Parallel processing hardware implementations are often categorized according to the particular resources which are shared. The following categories are described in this chapter:

- shared memory systems
- shared disk systems
- shared nothing systems

These implementations can also be described as “tightly coupled” or “loosely coupled,” according to the way in which communication between nodes is accomplished.

Attention: Oracle supports *all* these different implementations of parallel processing, assuming that in a shared nothing system the software enables a node to access a disk from another node. For example, the IBM SP2 features a virtual shared disk: the disk is shared through software.

Note: Support for any given Oracle configuration is platform-dependent; check to confirm that your platform supports the configuration you want.

Application Profiles

Online transaction processing (OLTP) applications tend to perform best on symmetric multiprocessors; they perform well con clusters and MPP systems if they can be well partitioned. Decision support (DSS) applications tend to perform well on SMPs, clusters, and massively parallel systems. Choose the implementation that provides the power you need for the application(s) you require.

Required Hardware and Operating System Software

Each hardware vendor implements parallel processing in its own way, but the following common elements are required for Oracle Parallel Server:

- High Speed Interconnect
- Globally Accessible Disk or Shared Disk Subsystem

High Speed Interconnect

This is a high bandwidth, low latency communication facility between the various nodes for lock manager and cluster manager traffic. The interconnect can be Ethernet, FDDI, or some other proprietary interconnect method. If the primary interconnect fails, a back-up interconnect is usually available. The back-up interconnect will ensure high availability, and prevent a single point of failure.

Globally Accessible Disk or Shared Disk Subsystem

All nodes in a loosely coupled or massively parallel system have simultaneous access to shared disks. This gives multiple instances of Oracle8 concurrent access to the same database. These shared disk subsystems are most often implemented via a shared SCSI or twintailed SCSI (common in UNIX) connected to a disk farm. On some MPP platforms, such as IBM SP, disks are associated to nodes and a virtual shared disk software layer enables global access to all nodes.

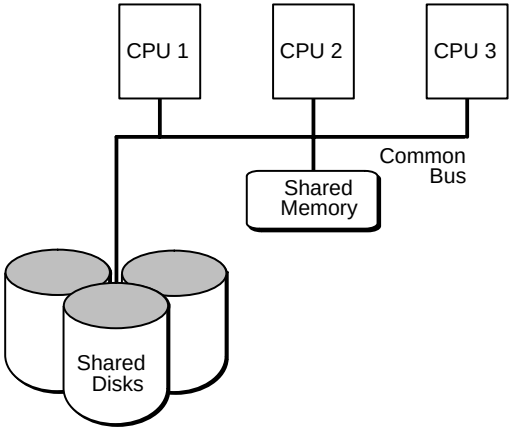
Note: The Integrated Distributed Lock Manager coordinates modifications of data blocks, maintenance of cache consistency, recovery of failed nodes, transaction locks, dictionary locks, and SCN locks.

Shared Memory Systems

Tightly coupled shared memory systems, illustrated in Figure 3–1, have the following characteristics:

- Multiple CPUs share memory.
- Each CPU has full access to all shared memory through a common bus.
- Communication between nodes occurs via shared memory.
- Performance is limited by the bandwidth of the memory bus.

Figure 3–1 Tightly Coupled Shared Memory System



Symmetric multiprocessor (SMP) machines are often nodes in a cluster. Multiple SMP nodes can be used with Oracle Parallel Server in a tightly coupled system, where memory is shared among the multiple CPUs, and is accessible by all the CPUs through a memory bus. Examples of tightly coupled systems include the Pyramid, Sequent, and Sun SparcServer.

It does not make sense to run Oracle Parallel Server on a single SMP machine, because the system would incur a great deal of unnecessary overhead from IDLM accesses.

Performance is potentially limited in a tightly coupled system by a number of factors. These include various system components such as the memory bandwidth, CPU to CPU communication bandwidth, the memory available on the system, the I/O bandwidth, and the bandwidth of the common bus.

Parallel processing advantages of shared memory systems are these:

- Memory access is cheaper than inter-node communication. This means that internal synchronization is faster than using the Lock Manager.
- Shared memory systems are easier to administer than a cluster.

A disadvantage of shared memory systems for parallel processing is as follows:

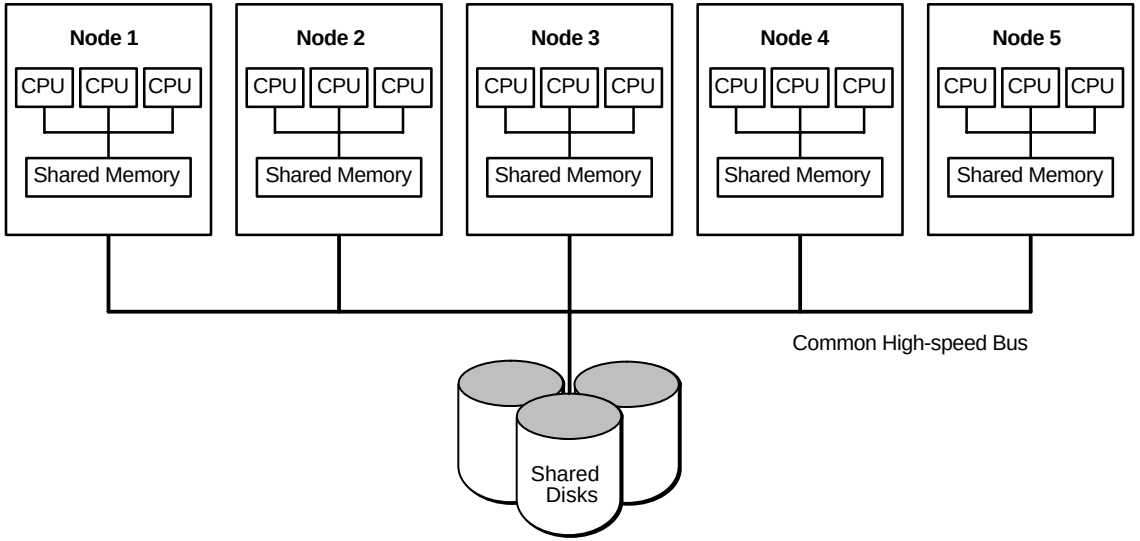
- Scalability is limited by bus bandwidth and latency, and by available memory.

Shared Disk Systems

Shared disk systems are typically loosely coupled. Such systems, illustrated in Figure 3–2, have the following characteristics:

- Each node consists of one or more CPUs and associated memory.
- Memory is not shared between nodes.
- Communication occurs over a common high-speed bus.
- Each node has access to the same disks and other resources.
- A node can be an SMP if the hardware supports it.
- Bandwidth of the high-speed bus limits the number of nodes (scalability) of the system.

Figure 3–2 Loosely Coupled Shared Disk System



The cluster illustrated in Figure 3–2 is composed of multiple tightly coupled nodes. The IDLM is required. Examples of loosely coupled systems are VAXclusters or Sun clusters.

Since the memory is not shared among the nodes, each node has its own data cache. Cache consistency must be maintained across the nodes and a lock manager is needed to maintain the consistency. Additionally, instance locks using the IDLM

on the Oracle level must be maintained to ensure that all nodes in the cluster see identical data.

There is additional overhead in maintaining the locks and ensuring that the data caches are consistent. The performance impact is dependent on the hardware and software components, such as the bandwidth of the high-speed bus through which the nodes communicate, and IDLM performance.

Parallel processing advantages of shared disk systems are as follows:

- Shared disk systems permit high availability. All data is accessible even if one node dies.
- These systems have the concept of one database, which is an advantage over shared nothing systems.
- Shared disk systems provide for incremental growth.

Parallel processing disadvantages of shared disk systems are these:

- Inter-node synchronization is required, involving IDLM overhead and greater dependency on high-speed interconnect.
- If the workload is not partitioned well, there may be high synchronization overhead.
- There is operating system overhead of running shared disk software.

Shared Nothing Systems

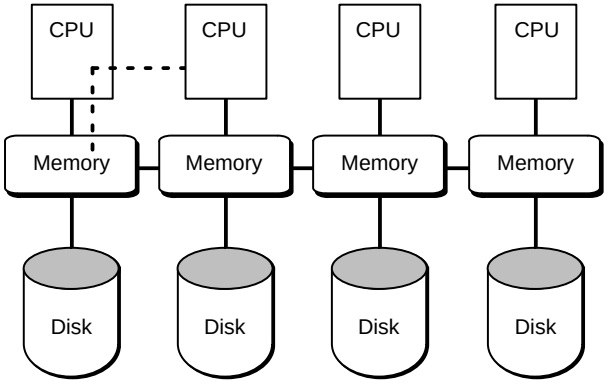
Shared nothing systems are typically loosely coupled. This section describes:

- Overview of Shared Nothing Systems
- Massively Parallel Systems
- Summary: Shared Nothing Systems

Overview of Shared Nothing Systems

In shared nothing systems only one CPU is connected to a given disk. If a table or database is located on that disk, access depends entirely on the CPU which owns it. Shared nothing systems can be represented as follows:

Figure 3–3 Shared Nothing System



Shared nothing systems are concerned with access to disks, not access to memory. Nonetheless, adding more CPUs and disks can improve scaleup. Oracle Parallel Server can access the disks on a shared nothing system as long as the operating system provides transparent disk access, but this access is expensive in terms of latency.

Massively Parallel Systems

Massively parallel (MPP) systems have the following characteristics:

- From only a few nodes, up to thousands of nodes are supported.
- The cost per processor may be extremely low because each node is an inexpensive processor.
- Each node has associated non-shared memory.
- Each node may have its own devices, but in case of failure other nodes can access the devices of the failed node.
- Nodes are organized in a grid, mesh, or hypercube arrangement.
- Oracle instances can potentially reside on any or all nodes.

A massively parallel system may have as many as several thousand nodes. Each node may have its own Oracle instance, with all the standard facilities of an instance. (An Oracle instance comprises the System Global Area and all the background processes.)

An MPP has access to a huge amount of real memory for all database operations (such as sorts or the buffer cache), since each node has its own associated memory. To avoid disk I/O, this advantage will be significant in long running queries and sorts. This is not possible for 32 bit machines which have a 2 GB addressing limit; the total amount of memory on an MPP system may well be over 2 GB. As with loosely coupled systems, cache consistency on MPPs must still be maintained across all nodes in the system. Thus, the overhead for cache management is still present. Examples of massively parallel systems are the nCUBE2 Scalar Supercomputer, the Unisys OPUS, Amdahl, Meiko, and the IBM SP.

Summary: Shared Nothing Systems

Shared nothing systems have advantages and disadvantages for parallel processing:

Advantages

- Shared nothing systems provide for incremental growth.
- System growth is practically unlimited.
- MPPs are good for read-only databases and decision support applications.
- Failure is local: if one node fails, the others stay up.

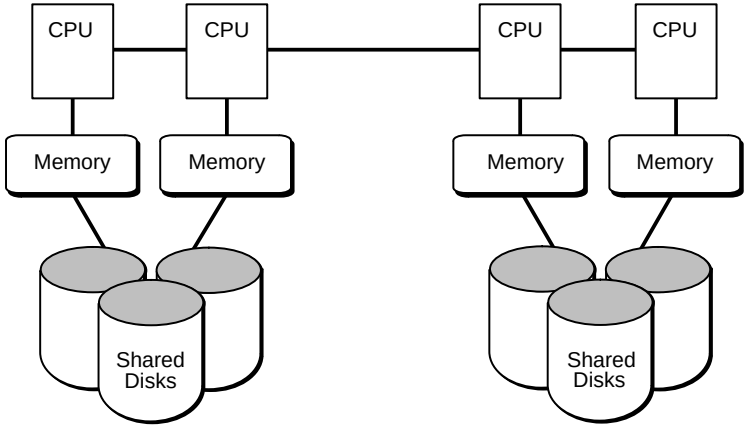
Disadvantages

- More coordination is required.
- More overhead is required for a process working on a disk belonging to another node.
- If there is a heavy workload of updates or inserts, as in an online transaction processing system, it may be worthwhile to consider data-dependent routing to alleviate contention.

Shared Nothing /Shared Disk Combined Systems

A combined system can be very advantageous—one which brings together the advantages of shared nothing and shared disk, while overcoming their respective limitations. Such a combined system can be represented as follows:

Figure 3–4 Two Shared Disk Systems Forming a Shared Nothing System



Here, two shared disk systems are linked to form a system with the same hardware redundancies as a shared nothing system. If one CPU fails, the other CPUs can still access all disks.

Part II

Oracle Parallel Server Concepts

4

How Oracle Implements Parallel Processing

This chapter gives a high-level view of how the Oracle Parallel Server (OPS) provides high performance parallel processing. Key issues include:

- Enabling and Disabling Parallel Server
- Synchronization
- High Performance Features
- Cache Coherency

See Also: Chapter 7, "Overview of Locking Mechanisms", for an understanding of lock hierarchy in Oracle.

Enabling and Disabling Parallel Server

Oracle Parallel Server can be enabled or disabled:

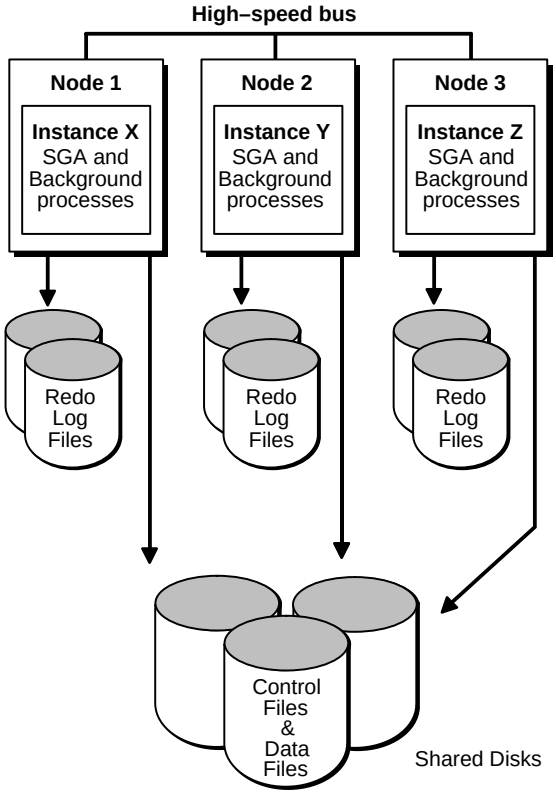
Oracle + Option	Parallel Server Disabled	Parallel Server Enabled	
		Single Node	Multiple Nodes
OPS not installed	Yes: default	No	No
OPS installed	Yes: default	Yes: Single Shared	Yes: Multiple Shared

When parallel server is disabled, only one Oracle instance can mount or open the database. This mode is necessary to create and completely recover a database. It is useful to implement Oracle Parallel Server but leave it disabled if standard Oracle functionality can meet your current needs, but you want your system to be parallel-server ready.

When parallel server is enabled, one or more instances of a parallel server mount the same database. All instances mount the database and read from and write to the same datafiles. *Single shared mode* describes an Oracle Parallel Server configuration in which only one instance is running. Global operations exist, but are not needed at the moment. The instance operates as though it is in a cluster (with Integrated DLM overhead, and so on), although there is no contention for resources. *Multiple shared mode* describes an Oracle Parallel Server configuration with multiple instances running.

Figure 4–1 illustrates a typical configuration in which Oracle Parallel Server is enabled with three instances on separate nodes accessing the database.

Figure 4–1 Shared Mode Sharing Disks



Note: Each instance can access the redo log files of the other instances.

See Also: "Enabling Parallel Server and Starting Instances" on page 18-12

Synchronization

Inter-node synchronization is an issue that does not need to be addressed in standard Oracle. But with Oracle Parallel Server you must have a broad understanding of the dimensions in which synchronization must occur. Some of these include:

- Block Level Locking
- Row Level Locking
- Space Management
- System Change Number

In Oracle Parallel Server exclusive mode, all synchronization is done within the instance. In shared mode, synchronization is accomplished with the help of the Integrated Distributed Lock Manager component.

Block Level Locking

Block access between instances is done on a per-block level. When certain blocks are locked by an instance, other instances are not permitted to access them. Every time Oracle tries to read a block from the database it needs to get an instance lock. Ownership of the lock is thus assigned to the instance.

Since Oracle Parallel Server runs in an environment having multiple memories, there can be multiple copies of the same data block in the multiple memories. Inter-node synchronization using the Integrated DLM is used to ensure that all copies of the block are valid: these block-level locks are the buffer cache locks.

Block level locking occurs only when parallel server is enabled. It is transparent to the user and to the application. (Row level locking also operates, whether parallel server is enabled or disabled.)

See Also: Chapter 9, “Parallel Cache Management Instance Locks”

Row Level Locking

Oracle Parallel Server provides row level locking in addition to block level locking in the buffer cache. In fact, row level locks are stored within the block.

Consider the following example. Instance 1 reads file 2, block 10 in order to update row 1. Instance 2 also reads file 2, block 10, in order to update row 2. Here, instance 1 obtains an instance lock on block 10, then locks and updates row 1. (The row lock is implicit because of the UPDATE statement.)

Instance 2 will then force instance 1 to write the updated block to disk, and instance 1 will give up ownership of the lock on block 10 so that instance 2 can have ownership of it. Instance 2 will then lock row 2 and perform its own UPDATE.

Space Management

Free lists and free list groups are used to optimize space management in Oracle Parallel Server.

The problem of allocating space for inserts illustrates space management issues. When a table uses more space, how can you make sure that no one else uses the same space? How can you make sure that two nodes are not inserting into the same space on the same disk, in the same file?

Consider the following example. Instance 1 reads file 2, block 10 in order to insert a row. Instance 2 reads file 3, block 20, in order to insert another row. Each instance proceeds to insert rows as needed. If one particular block were responsible for assigning enough space for all these inserts, that block would constantly ping between the instances. Instance 1 would lose ownership of the block when instance 2 needs to make an insert, and so forth. The situation would involve a great deal of contention, and performance would suffer.

By contrast, free list groups make good space management possible. If two instances are inserting into the same object (such as a table), but each instance has its own set of free lists for that object, then contention for a single block would be avoided. Each instance would insert into a different block belonging to the object.

System Change Number

In standard Oracle, the system change number (SCN) is maintained and incremented in the SGA by an exclusive mode instance. In Oracle Parallel Server shared mode, the SCN must be maintained globally. Its implementation may vary from platform to platform. The SCN may be handled by the Integrated DLM, by the Lamport SCN scheme, or by using a hardware clock or dedicated SCN server.

See Also: Your Oracle system-specific documentation.
"Lamport SCN Generation" on page 4-7
"System Change Number (SC)" on page 10-4

High Performance Features

A parallel server takes advantage of systems of linked processors sharing resources without sacrificing any transaction processing features of Oracle. The following sections discuss in more detail certain features that optimize performance on the Oracle Parallel Server.

- Fast Commits, Group Commits, and Deferred Writes
- Row Locking and Multiversion Read Consistency
- Online Backup and Archiving
- Sequence Number Generators
- Lamport SCN Generation
- Free Lists
- Free List Groups
- Disk Affinity

Within a single instance, Oracle uses a buffer cache in memory to reduce the amount of disk I/O necessary for database operations. Since each node in the parallel server has its own memory that is not shared with other nodes, Oracle Parallel Server must coordinate the buffer caches of different nodes while minimizing additional disk I/O that could reduce performance. The Oracle parallel cache management technology maintains the high-performance features of Oracle while coordinating multiple buffer caches.

See Also: *Oracle8 Concepts* for further information about each of these high-performance features.

Fast Commits, Group Commits, and Deferred Writes

Fast commits, group commits, and deferred writes operate on a per-instance basis in Oracle and work the same whether in exclusive or shared mode.

Oracle only reads data blocks from disk if they are not already in the buffer cache of the instance that needs the data. Because data block writes are deferred, they often contain modifications from multiple transactions.

Optimally, Oracle writes modified data blocks to disk only when necessary:

- when the blocks have not been used recently and new data requires buffer cache space (in shared or exclusive mode)
- during checkpoints (shared or exclusive mode)

- when another instance needs the blocks (only in shared mode)

Oracle may also perform unnecessary writes to disk caused by forced reads or forced writes.

See Also: "How to Detect False Pinging" on page 15-16

Row Locking and Multiversion Read Consistency

The Oracle row locking feature allows multiple transactions on separate nodes to lock and update different rows of the same data block, without any of the transactions waiting for the others to commit. If a row has been modified but not yet committed, the original row values are available to all instances for read access (this is called *multiversion read consistency*).

Online Backup and Archiving

A parallel server supports all of the backup features of Oracle in exclusive mode, including both online and offline backups of either an entire database or individual tablespaces.

If you operate Oracle in ARCHIVELOG mode, online redo log files are archived before they can be overwritten. In a parallel server, each instance can automatically archive its own redo log files or one or more instances can archive the redo log files manually for all instances.

In ARCHIVELOG mode, you can make both online and offline backups. If you operate Oracle in NOARCHIVELOG mode, you can only make offline backups. Operating production databases in ARCHIVELOG mode is strongly recommended.

Sequence Number Generators

A parallel server allows users on multiple instances to generate unique sequence numbers with minimal cooperation or contention among instances.

The sequence number generator allows multiple instances to access and increment a sequence without contention among instances for sequence numbers and without waiting for any transactions to commit. Each instance can have its own sequence cache for faster access to sequence numbers. Integrated DLM locks coordinate sequences across instances in a parallel server.

Lamport SCN Generation

The System Change Number (SCN) is a logical time stamp Oracle uses to order events within a single instance, and across all instances. For example, Oracle

assigns an SCN to each transaction. Conceptually, there is a global serial point that generates SCNs. In practice, however, SCNs can be read and generated in parallel. One of the SCN generation schemes is called the Lamport SCN generation scheme.

The Lamport SCN generation scheme is fast and scalable because it can generate SCNs in parallel on all instances. In this scheme, all messages across instances, including lock messages, piggyback SCNs. These piggybacked SCNs propagate causalities within Oracle. As long as causalities are respected in this way, multiple instances can generate SCNs in parallel, with no need for extra communication among these instances.

On most platforms, Oracle uses the Lamport SCN generation scheme when the MAX_COMMIT_PROPAGATION_DELAY is larger than a platform-specific threshold (typically 7 seconds). You can examine the alert log after an instance is started to see whether the Lamport SCN generation scheme has been picked.

See Also: Your Oracle system-specific documentation.

Free Lists

Standard Oracle can use multiple use free lists as a way to reduce contention on blocks. A free list is a list of data blocks, located in extents, that contain free space. These data blocks are used when inserts or updates are made to a database object such as a table or a cluster. No contention among instances occurs when different instances' transactions insert data into the same table. This is achieved by locating free space for the new rows using *free space lists* that are associated with one or more instances. The free list may be from a common pool of blocks, or multiple free lists may be partitioned so that specific extents in files are allocated to objects.

With a single free list, when multiple inserts are taking place, single threading occurs as these processes try to allocate space from the free list. The advantage of using multiple free lists is that it allows processes to search a specific pool of blocks when space is needed, thus reducing contention among users for free space.

See Also: Chapter 11, "Space Management and Free List Groups"

Free List Groups

Oracle Parallel Server can use free list groups to eliminate contention between instances for access to a single block containing free lists.

Even if multiple free lists reside in a single block, on Oracle Parallel Server the block containing the free lists would have forced reads/writes between all the instances all the time. To avoid this problem, free lists can be grouped, with one group assigned to each instance. Each instance then has its own block containing

free lists. Since each instance uses its own free lists, there is no contention between instances to access the same block containing free lists.

See Also: Chapter 17, “Using Free List Groups to Partition Data” regarding proper use of free lists to achieve optimal performance in an Oracle Parallel Server environment.
"Backing Up the Database" on page 21-12

Disk Affinity

Disk affinity determines on which instances or processes to perform a parallelized DML or query operation. Affinity is especially important for parallel DML when running in Oracle Parallel Server configurations. Affinity information which persists across statements can improve the buffer cache hit ratio and reduce forced reads / writes on blocks between instances.

The granularity of parallelism for most PDML operations is by partition. For parallel query, granularity is by rowid. Parallel DML operations need a partition-to-instance mapping to implement affinity. The segment header of the partition is used to determine the affinity of the partition for MPPs. Better performance is achieved by having nodes mainly access local devices, with a better buffer cache hit ratio for every node.

For other Oracle Parallel Server configurations, a deterministic mapping of partitions to instances is used. Partition-to-instance affinity information is used to determine slave allocation and work assignment for all OPS / MPP configurations.

See Also: Parallel Data Manipulation Language (parallel DML) and degree of parallelism are discussed at length in *Oracle8 Concepts*. For a discussion of PDML tuning and optimizer hints, please see *Oracle8 Tuning*.

Client-Side Application Failover

Application failover enables the application to automatically reconnect to the database if the connection is broken. Any active transaction will be rolled back, but the new database connection will otherwise be identical to the original one. This is true regardless of whether the connection was lost because the instance died or for some other reason.

With application failover, a client sees no loss of connection as long as there is one instance left serving the application. The DBA controls which applications run on particular instances, and creates a failover order for each application.

See Also: "Client-side Application Failover" on page 22-2

Cache Coherency

Cache coherency is the technique of keeping multiple copies of an object consistent. This section describes:

- Parallel Cache Management Issues
- Non-PCM Cache Management Issues

Parallel Cache Management Issues

With the Oracle Parallel Server option, separate Oracle instances run simultaneously on one or more nodes using a technology called *parallel cache management*.

Parallel cache management uses *Integrated Distributed Lock Manager locks* to coordinate access to resources required by the instances of a parallel server. Rollback segments, dictionary entries, and data blocks are some examples of database resources. The most often required database resources are data blocks.

Cache coherency is provided by the Parallel Cache Manager for the buffer caches of instances located on separate nodes. The set of global constant (GC_*) initialization parameters associated with PCM buffer cache locks are *not* used with the dictionary cache, library cache, and so on.

The Parallel Cache Manager ensures that a master copy data block in an SGA has identical copies in other SGAs that require a copy of the master. Thus, the most recent copy of the block in all SGAs contains all changes made to that block by all instances in the system, regardless of whether any of the transactions on those instances have committed.

If a data block is modified in one buffer cache, then all existing copies in other buffer caches are no longer current. New copies can be obtained after the modification operation completes.

Parallel cache management enforces cache coherency while minimizing I/O and use of the Integrated DLM. I/O and lock operations for cache coherency are only done when the current version of a data block is in one instance's buffer cache and another instance requests that block for update.

Multiple transactions running on a single instance of a parallel server can share access to a set of data blocks without additional instance lock operations, as long as the blocks are not needed by transactions running on other instances.

In shared mode, the Integrated Distributed Lock Manager maintains the status of instance locks. In exclusive mode, all locks are local and the IDLM is not used to coordinate database resources.

Instances use instance locks simply to indicate the ownership of a master copy of a resource. When an instance becomes the owner of a master copy of a database resource, it also inherently becomes the owner of the instance lock covering the resource, with fixed locking. (Releasable locks are, of course, released.) A master copy indicates that it is an updatable copy of the resource. The instance only *disowns* the instance lock when another instance requests the resource for update. Once another instance *owns* the master copy of the resource, it becomes the owner of the instance lock.

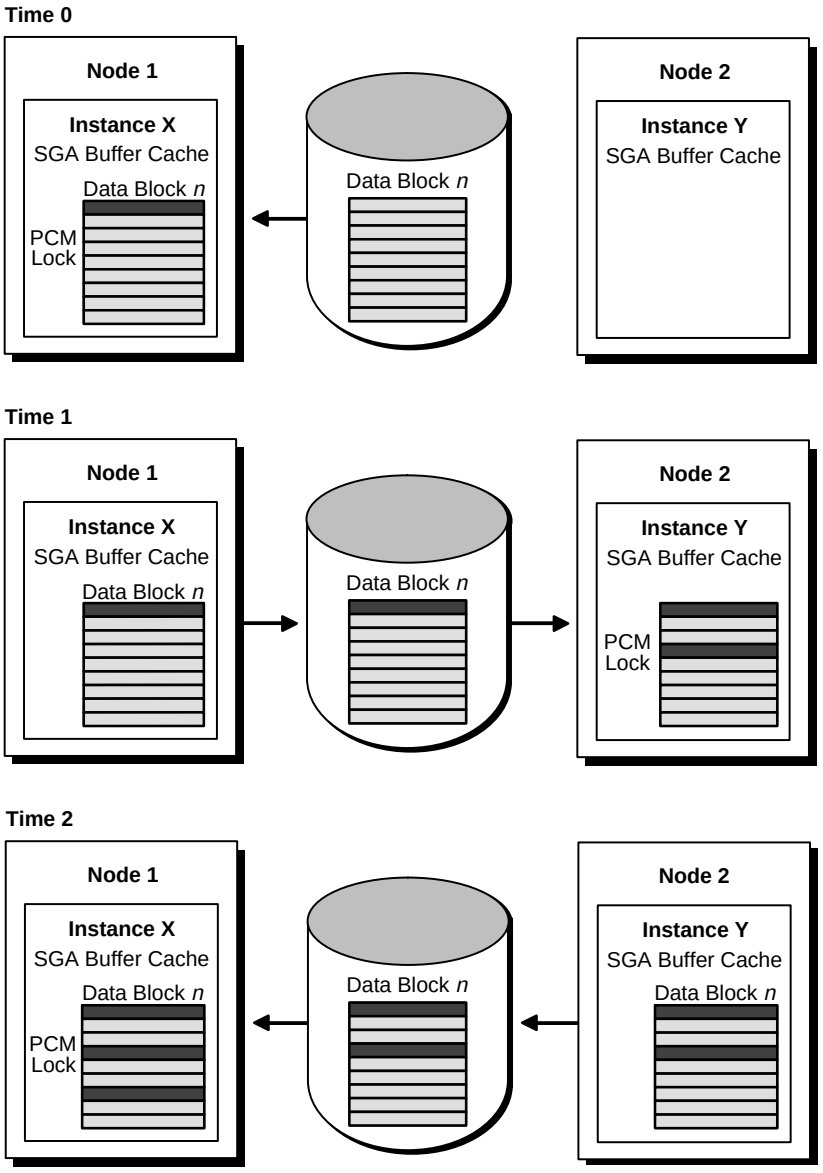
Attention: Transactions and parallel cache management are autonomous mechanisms in Oracle. PCM locks function independently of any form of transaction lock.

Example Consider the following example and the illustrations in Figure 4–2. (This example assumes that one PCM lock covers one block--although many blocks could be covered.)

- Instance X becomes the owner of the PCM lock covering data block *n* containing row 1 and updates the row.
- Instance Y requests the block to update row 4.
- Instance X writes the data block to disk and releases the PCM lock.
- Instance Y becomes the owner of the block and the PCM lock, and then updates row 4.
- Instance X requests the block to update row 7.
- Instance Y writes the data block to disk and releases the block and the PCM lock.
- Instance X becomes the owner of the block and PCM lock and updates row 7.
- Instance X commits its transaction and still owns the PCM lock and the master copy of the block until another instance requests the block.

See Also: "How Buffer State and Lock Mode Change" on page 9-11

Figure 4–2 Multiple Instances Updating the Same Data Block



Independence of PCM Locks and Row Locks

PCM locks and row locks operate independently of each other. An instance can disown a PCM lock without affecting row locks held in the set of blocks covered by the PCM lock. A row lock is acquired during a transaction. A database resource, such as a data block, acquires a PCM lock when it is read for update by an instance. During a transaction, a PCM lock can therefore be disowned and owned many times if the blocks are needed in other instances.

In contrast, transactions do not release row locks until changes to the rows are either committed or rolled back. Oracle uses internal mechanisms for concurrency control to isolate transactions, so that modifications to data made by one transaction are not visible to other transactions until the transaction modifying the data commits. The row lock concurrency control mechanisms are independent of parallel cache management: concurrency control does not require PCM locks, and PCM lock operations do not depend on individual transactions committing or rolling back.

Instance Lock Modes

An instance can acquire the instance lock that covers a set of data blocks in either shared or exclusive mode, depending on the type of access required.

- *Exclusive lock mode* allows the instance to update a set of blocks.

  If one instance needs to update a data block and a second instance already holds the instance lock that covers the block, the first instance uses the IDLM lock to request that the second instance disown the instance lock, writing the block(s) to disk if necessary.

- *Read lock mode* only allows the instance to read blocks.

  Multiple instances can own an instance lock as long as they only need to read, not modify, the blocks covered by that instance lock. Thus, all instances can be sure that their memory-resident copy of the block is a current copy, or that they can read the current copy from disk without any instance lock operations to request the block from another instance. This means that instances do not have to disown instance locks for the portion of a database accessed for read-only use, which may be a substantial portion of the time in many applications.

- *Null lock mode* allows instances to keep a lock without any permissions on the block(s).

  This mode is used so that locks need not be continually obtained and released—locks are just converted from one mode to another.

See Also: Chapter 15, “Allocating PCM Instance Locks”, for a detailed description of allocating PCM locks for datafiles.

Non-PCM Cache Management Issues

Oracle Parallel Server ensures that all of the standard Oracle caches are synchronized across the instances. Changing a block on one node, and its ramifications for the other nodes, is a familiar example of synchronization. Synchronization has broader implications, however.

Understanding the way that caches are synchronized across instances can help you to understand the ongoing overhead which affects the performance of your system. Consider a five-node parallel server in which someone drops a table on one node. Each of the five dictionary caches has a copy of the definition of that particular table, thus the node that drops the table from its own dictionary cache must also flush the other four dictionary caches. It does this automatically through the Integrated DLM. Users on the other nodes will be notified of the change in lock status.

There are big advantages to having each node cache the library and table information. Occasionally, a command like DROP TABLE will force other caches to be flushed, but the brief effect this may have on performance does not diminish the advantage of having multiple caches.

See Also: "Space Management" on page 4-5
"System Change Number" on page 4-5 for additional examples of non-PCM cache management issues.

5

Oracle Instance Architecture for the Parallel Server

*[Architecture] is music in space, as it were a frozen mu*sic...

— Schelling, *Philosophie der Kunst*

This chapter explains features of Oracle multi-instance architecture which differ from an Oracle server in exclusive mode.

- Overview
- Characteristics of OPS Multi-instance Architecture
- System Global Area
- Background Processes and LCKn
- Configuration Guidelines for Oracle Parallel Server

Overview

Each Oracle instance in a parallel server architecture has its own

- system global area (SGA)
- background processes
- ORACLE_SID
- set of redo logs

All instances in a parallel server database share

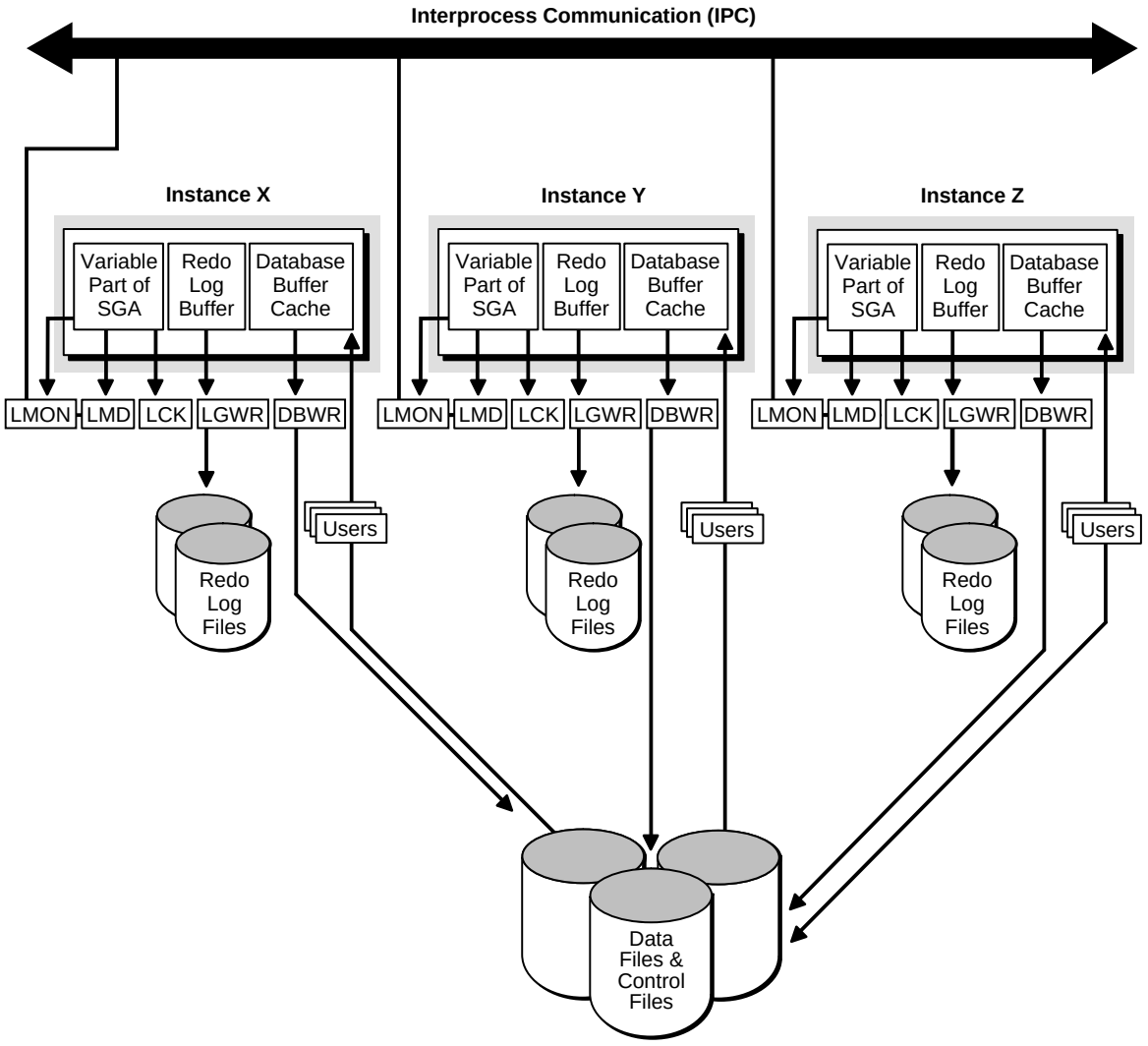
- the same set of data files
- the same set of control files

The Oracle Parallel Server (OPS) instance contains

- an additional PCM lock area in its SGA to coordinate the use of shared resources (also known as lock elements)
- the Integrated Distributed Lock Manager component, an area for global locks and resources. This area was formerly part of the external distributed lock manager (DLM).
- additional background processes LCK*n* to coordinate the locking of shared resources among the multiple instances in a parallel server
- additional background processes LMON and LMD0 to manage global locks and resources

Basic elements of the Oracle Parallel Server are illustrated in Figure 5–1. DBWR processes are shown writing data, users reading data.

Figure 5–1 Basic Elements of Oracle Parallel Server



See Also: "Memory Structures and Processes" in *Oracle8 Concepts.*

Characteristics of OPS Multi-instance Architecture

Characteristics of an Oracle parallel server can be summarized as follows:

- An Oracle instance can be started on one or more nodes in the network.
- Each instance has a separate System Global Area (SGA) and set of background processes.
- All instances share the same datafiles and control file.
- Each instance has its own set of redo log files.

  Note: The redo logs must be accessible to all instances in case of instance failure. On some MPP platforms, a redo server exists so that only one set of redo logs is necessary for the whole OPS system.
- Archived logs are private, but must be accessible to all instances for media recovery.
- All instances can execute transactions concurrently against the same database, and each instance can have multiple users executing transactions.
- Row level locking is preserved.

A parallel server is administered in the same manner as a non-parallel server, except that you must connect to a particular instance to perform administration.

Applications that access the database can run on the same nodes as instances of a parallel server or on separate nodes, using the client-server architecture. A parallel server can be part of a distributed database system. Distributed transactions access the data in a remote database in the same manner, regardless of whether the datafiles are owned by an Oracle Server (in exclusive mode) or a parallel server (in exclusive or shared mode).

Other non-Oracle processes can run on each node of the system, or you can dedicate the entire system or part of the system to Oracle. For example, a parallel server and its applications might occupy three nodes of a five-node configuration, while the other two nodes are used for non-Oracle applications.

System Global Area

Each instance of a parallel server has its own System Global Area (SGA). The SGA includes the following memory structures:

- buffer cache for data blocks
- dictionary cache for data dictionary information
- redo log buffer for redo entries
- shared pool containing the shared SQL and shared PL/SQL areas
- instance lock area (only in a parallel server)

Data sharing between SGAs in a parallel server is controlled by *parallel cache management*, which uses parallel cache management (PCM) locks.

A data block can be present in several SGAs at the same time. PCM locks ensure that the database buffer cache is kept consistent for all the instances. It thus ensures readability by one instance of changes made by other instances.

Each instance has a shared pool that can only be used by the user applications connected to that instance. If the same SQL statement is submitted by different applications using the same instance, it is parsed and stored once in that instance's SGA. If that same SQL statement is also submitted by an application on a different instance, then this different instance also parses and stores the statement.

See Also: Chapter 9, "Parallel Cache Management Instance Locks".

Background Processes and LCK*n*

Each instance in a parallel server has its own set of background processes, which are identical to the background processes of a single server in exclusive mode. The DBWR, LGWR, PMON, and SMON processes are present for every instance; the optional processes, ARCH, CKPT, D*nnn* and RECO, can be enabled by setting the appropriate initialization parameters. In addition to the standard background processes, each instance of a parallel server has at least one lock process, LCK0. Additional lock processes can be enabled if necessary.

In OPS the Integrated Distributed Lock Manager also uses the LMON and LMD0 processes. LMON is used to manage instance and process deaths and associated recovery for the Integrated DLM. In particular, LMON handles the part of recovery that is associated with the global locks. The LMD processes are used to handle remote lock requests (those which originate from other instances).

The Lock process (LCK*n*) manages the locks used by an instance and coordinates requests for those locks by other instances. Additional lock processes, LCK1 through LCK9, are available for systems that require exceptionally high throughput of instance lock requests. The single lock process per instance, LCK0, is usually sufficient for most systems.

All instances in a parallel server must have the same number of lock processes. Lock processes use the Integrated DLM to coordinate the buffer caches of the different SGAs in a parallel server.

When an instance fails in shared mode, another instance's SMON detects the failure and recovers for the failed instance. The lock processes of the instance doing the recovery clean up any outstanding PCM locks for the failed instance.

All the lock processes for one instance are functionally equivalent. Typically, one lock process is sufficient. Occasionally, more than one lock process may maximize throughput of locking requests. Although lock requests are asynchronous, each request is blocked until the lock process knows if the lock can be granted immediately. Since multiple lock processes are functionally equivalent, this manual refers only to *the lock process*.

See Also: "The LCKn Processes" on page 7-6
"GC_* Initialization Parameters" on page 9-13

Configuration Guidelines for Oracle Parallel Server

When setting up an Oracle Parallel Server environment, observe the guidelines presented in Table 5–1:

Table 5–1 Parallel Server Configuration Guidelines

Configuration Issue	Guidelines
Version	Ensure that the same Oracle version exists on all the nodes.
Links	UNIX soft or hard inks (“aliases”) to executables are not recommended for a parallel server. If the single node containing the executables fails, none of the nodes will be able to operate.
Initialization parameters	Keep in a single file the initialization parameters which should be identical across all instances in a parallel server. Include this file in the individual initialization files of the different instances using the IFILE option.
Control files	Must be accessible from all instances.
Data files	Must be accessible from all instances.
Log files	Must be located on the same set of disks as control files and data files. Although the redo log files are independent per instance, each of the log files must still be accessible by all the instances so that recovery is provided for.
NFS	You can use NFS to enable access to Oracle executables, but not access to database files or log files. Note that if you are using NFS, the serving node is a single point of failure.
Archived redo log files	Must be accessible from all instances.

6

Oracle Database Architecture for the Parallel Server

This chapter describes features of Oracle database architecture that pertain to the multiple instances of a parallel server.

- File Structures
- The Data Dictionary
- The Sequence Generator
- Rollback Segments

File Structures

The following sections describe the features of control files, datafiles, and redo log files that apply to a parallel server.

- Control Files
- Datafiles
- Redo Log Files

Control Files

All instances of a parallel server access the same control files. The control files hold the values of *global constant* initialization parameters, such as GC_DB_LOCKS, some of which must be identical for all instances running concurrently. As each instance starts up, Oracle compares the global constant initialization values in a common parameter file (or in parameter files for each instance) with those in the control file, and generates a message if the values are different.

See Also: *Oracle8 Concepts*
"Parameters Which Must Be Identical on Multiple Instances" on page 18-10.
"Using a Common Parameter File for Multiple Instances" on page 18-3.
“Initialization Parameters” on page A-18.

Datafiles

All instances of a parallel server access the same datafiles. Database files are the same for Oracle in parallel mode as for Oracle in exclusive mode. You do not have to change the datafiles to start Oracle in exclusive or parallel mode.

To improve performance, you can control the physical placement of data so that the various instances use disjoint sets of data blocks. Free lists, for example, enable you to allocate space for inserts to particular instances.

Whenever an instance starts up, it verifies access to all online datafiles. The first instance to start up in a parallel server must verify access to all online files so that it can determine if media recovery is required. Additional instances can operate without access to all of the online datafiles, but any attempt to use an unverified file fails and a message is generated.

When an instance adds a datafile or brings an offline datafile online, all instances verify access to the file. If an instance adds a new datafile on a disk that other instances cannot access, verification fails, but the instances continue running. Verification can also fail if instances access different copies of the same datafile.

If verification fails for any instance, you need to diagnose and fix the problem, then use the ALTER SYSTEM CHECK DATAFILES statement to verify access. This statement has a GLOBAL option (the default), which makes all instances verify access to the online datafiles, and a LOCAL option, which makes the current instance verify access.

ALTER SYSTEM CHECK DATAFILES makes the online datafiles available to the instance or instances for which access is verified.

Oracle cannot recover from instance failure or media failure unless the instance that performs recovery can verify access to all of the required online datafiles.

Oracle automatically maps absolute file numbers to relative file numbers. Use of a parallel server does not affect these values. Query the V$DATAFILE view to see both numbers for your datafiles.

See Also: Chapter 17, “Using Free List Groups to Partition Data”.
"Access to Datafiles for Instance Recovery" on page 22-13.
"Setting GC_FILES_TO_LOCKS: PCM Locks for Each Datafile" on page 15-7.
For more information about relative file numbers, see *Oracle8 Concepts*.

Redo Log Files

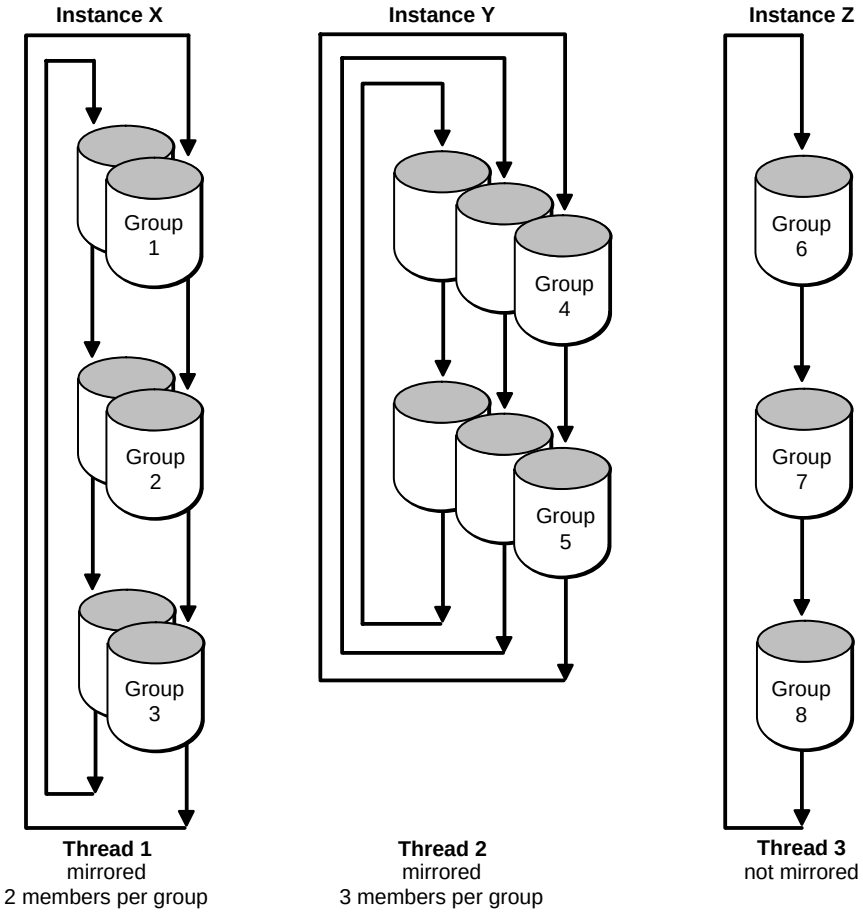
In a parallel server, each instance writes to its own set of online redo log files. The redo written by a single instance is called a *thread* of redo. Each online redo log file is associated with a particular thread number. When an online redo log is archived, its thread number is recorded to identify it during recovery.

A *private thread* is a redo log created using the ALTER DATABASE ADD LOGFILE command with the THREAD clause. A *public thread* is a redo log created using the ALTER DATABASE ADD LOGFILE, but no THREAD clause is specified.

If the THREAD initialization parameter is specified, the instance starting up acquires the thread identified by that value as a private thread. If THREAD is the default of zero, the instance acquires a public thread. Once acquired, a redo thread is used exclusively by the acquiring instance.

Online redo log files can be multiplexed (or “mirrored”). A multiplexed redo log consists of two or more groups of files, and all members of a group are written to concurrently when that group is active. Figure 6–1 shows the threads of redo for three instances of a parallel server.

Figure 6–1 Threads of Redo



- Instance X uses thread 1, which contains three *groups* of online redo log files (groups 1, 2, and 3). Thread 1 is multiplexed, that is, each group has two copies (*members*) of the redo log file.
- Instance Y uses thread 2, which contains two groups of online redo log files (groups 4 and 5). Thread 2 is multiplexed, with three members per group.
- Instance Z uses thread 3, which contains three groups of online redo log files (groups 6, 7, and 8) that are not multiplexed.

Group numbers must be unique within the database, but the order of assigning groups to instances is arbitrary. For example, although in Figure 6–1 thread 1 contains groups 1, 2, and 3 while thread 2 contains groups 4 and 5, you could instead assign groups 2, 4, and 5 to thread 1 while assigning groups 1 and 3 to thread 2. The V$LOGFILE view displays the group number associated with each redo log file.

Although it is possible to have different numbers of groups and members per thread, Oracle Corporation recommends that all threads be configured to a common standard.

Different instances of a parallel server can have different degrees of mirroring (different numbers of members per group), and can also have different numbers of groups. For example, one instance could have three groups with two members per group, a second instance could have four log files that are not multiplexed, and a third instance could have two groups with four members per group. While such a configuration may be inconvenient to administer, it may be necessary in order to realize the full potential of the system.

Each instance must have at least two groups of online redo log files. When the current group fills, an instance begins writing to the next log file group. At a log switch, information is written to the control file that can be used to identify the filled group and its thread number after it has been archived.

The number of redo log files about which the control file can keep information is limited by the value of the MAXLOGHISTORY option of the CREATE DATABASE statement. Note that only one member per group is needed. MAXLOGHISTORY can be extremely useful for sites with very demanding availability requirements. This option can assist you in administration of recovery, especially when there are many instances and many log files.

Attention: In Oracle Parallel Server, you should set the value of MAXLOGHISTORY higher than in single instance Oracle, because in OPS the history of multiple redo log files must be tracked.

See Also: “Recovery Structures” in *Oracle8 Concepts* for a full description of multiplexed redo log files.
"Archiving the Redo Log Files" on page 21-2.
"Checkpoints and Log Switches" on page 21-8.
"Recovery from Media Failure" on page 22-15.

The Data Dictionary

Each instance of a parallel server has a dictionary cache (row cache) containing data dictionary information in its SGA. The data dictionary structure is the same for Oracle instances in parallel mode as for an instance in exclusive mode. Instance locks coordinate the data dictionary activity of multiple instances.

The Sequence Generator

This section describes the CREATE SEQUENCE statement and its options.

- The CREATE SEQUENCE Statement
- The CACHE Option
- The ORDER Option

The CREATE SEQUENCE Statement

The SQL statement CREATE SEQUENCE establishes a database object from which multiple users can generate unique integers without waiting for other users to commit transactions that access the same sequence number generator.

A parallel server allows users on multiple instances to generate unique sequence numbers with minimal cooperation or contention among instances. Instances locks coordinate sequences across instances in a parallel server.

Sequence numbers are always unique, unless you use the CYCLE option. However, sequence numbers may be assigned out of order if you use the CACHE option without the ORDER option, as described in the following section.

See Also: *Oracle8 SQL Reference.*

The CACHE Option

The CACHE option of CREATE SEQUENCE pre-allocates sequence numbers so that they may be kept in an instance's SGA for faster access. You can specify the number of sequence numbers cached as an argument to the CACHE option; the default value is 20.

Caching sequence numbers significantly improves performance but can cause the loss of some numbers in the sequence. Losing sequence numbers is unimportant in some applications, such as when sequences are used to generate unique numbers for primary keys.

A cache for a given sequence is populated at the first request for a number from that sequence. After the last number in that cached set of numbers is assigned, the cache is repopulated.

Each instance keeps its own cache of sequence numbers in memory. When an instance shuts down, cached sequence values that have not been used in committed DML statements can be lost. The potential number of lost values can be as great as the value of the CACHE option times the number of instances shutting down. Cached sequence numbers can be lost even when an instance shuts down normally.

The initialization parameter SEQUENCE_CACHE_ENTRIES determines the number of sequences that can be cached in the SGA for a given instance. For highest concurrency, set SEQUENCE_CACHE_ENTRIES to the highest possible number of cached sequences that an instance uses at one time.

The ORDER Option

The ORDER option of CREATE SEQUENCE guarantees that sequence numbers are generated in the order of the requests. You can use the ORDER option for timestamp numbers and other sequences that must indicate the request order across multiple processes and instances.

If you do not require sequence numbers to be issued in order, the NOORDER option of CREATE SEQUENCE can significantly reduce overhead in a parallel server environment.

Attention: Oracle Parallel Server does not support the CACHE option with the ORDER option of CREATE SEQUENCE when the database is mounted in parallel mode. Oracle cannot guarantee an order if each instance has some sequence values cached. Therefore, if you should create sequences with both the CACHE and ORDER options, they will be ordered but not cached.

Rollback Segments

This section describes rollback segments as they relate to Oracle Parallel Server.

- Rollback Segments on a Parallel Server
- Parameters Which Control Rollback Segments
- Public and Private Rollback Segments
- How Instances Acquire Rollback Segments

Rollback Segments on a Parallel Server

Rollback segments contain information required for read consistency and to undo changes made by transactions that roll back or abort. Each instance in a parallel server shares use of the SYSTEM rollback segment and requires at least one dedicated rollback segment.

Both private and public rollback segments are acquired at instance startup and used exclusively by the acquiring instance until taken offline or at the acquiring instance shutdown. Private rollback segments are unique to a particular instance and cannot be used by any other instance. A public rollback segment is offline and not used by any instance until an instance that needs an extra rollback segment starts up, acquires it, and brings it online; once online, a public rollback is used exclusively by the acquiring instance.

Only one instance writes to a given rollback segment (except for the SYSTEM rollback segment), but other instances can read from it to create read-consistent snapshots or to perform instance recovery.

A parallel server needs at least as many rollback segments as the maximum number of concurrent instances plus one (SYSTEM). An instance cannot start unless it has exclusive access to at least one rollback segment, whether it is public or private.

You can create new rollback segments in any tablespace. To reduce contention between rollback data and table data, you can partition your rollback segments in a separate tablespace. This also facilitates taking tablespaces offline, because a tablespace cannot be taken offline if it contains an active rollback segment.

In general, you should make all extents for rollback segments the same size by specifying identical values for the storage parameters INITIAL and NEXT.

The data dictionary view DBA_ROLLBACK_SEGS shows each rollback segment's name, segment ID number, and owner (PUBLIC or other).

See Also: "Creating Additional Rollback Segments" on page 14-5 for information about the rollback segments that are required when you create a database.

Oracle8 Administrator's Guide for information about contention for a rollback segment and the performance implications of adding rollback segments.

Parameters Which Control Rollback Segments

The following initialization parameters control the use of rollback segments:

ROLLBACK_SEGMENTS	specifies the names of rollback segments that the instance acquires at startup
GC_ROLLBACK_LOCKS	reserves additional instance locks to reduce contention for blocks that contain rollback entries. In particular, it reserves instance locks for deferred rollback segments, which contain rollback entries for transactions in tablespaces that were taken offline.

See Also: "Monitoring Rollback Segments" on page 14-6
"Data Blocks, Extents, and Segments" in *Oracle8 Concepts*

Public and Private Rollback Segments

There are no performance differences between public and private rollback segments. However, private rollback segments provide more control over the matching of instances with rollback segments, allowing you to locate the rollback segments for different instances on different disks to improve performance. You can therefore use private rollback segments to reduce disk contention in a high-performance system.

Public rollback segments form a pool of rollback segments that can be acquired by any instance needing an additional rollback segment. Using public rollback segments can be disadvantageous when instances are shutdown and started up at the same time. For example, instance X shuts down and releases public rollback segments. Instance Y starts up and acquires the released rollback segments. Instance X starts up and cannot acquire its original rollback segments.

By default a rollback segment is private and is used by the instance specifying it in the parameter file. Private rollback segments are specified using the parameter ROLLBACK_SEGMENTS.

Once a public rollback segment is acquired by an instance, it is then used exclusively by that instance.

Once created, both public and private rollback segments can be brought online using the ALTER ROLLBACK SEGMENT command.

Note: An instance needs at least one rollback segment or it will not be able to start up.

How Instances Acquire Rollback Segments

When an instance starts up, it uses the TRANSACTIONS and TRANSACTIONS_PER_ROLLBACK initialization parameters to determine how many rollback segments to acquire, as follows:

$$\frac{TRANSACTIONS}{TRANSACTIONS\_PER\_ROLLBACK} = total\_rollback\_segments\_required$$

The *total_rollback_segments_required* number is rounded up.

At startup, an instance attempts to acquire rollback segments as follows.

- An instance first acquires any private rollback segments specified by the ROLLBACK_SEGMENTS initialization parameter. If this *total_private_rollback_segments* number is more than the *total_rollback_segments_required*, then no further action is taken to acquire rollback segments.
- If no private rollback segments are specified in the initialization file, the instance attempts to acquire public rollback segments.
- If the *total_private_rollback_segments* falls short of the *total_rollback_segments_required*, then the instance attempts to make up the difference by acquiring public rollback segments.
- If only one private rollback segment is specified and acquired, or one public rollback segment is acquired, the instance will start up, even if one rollback segment is below the *total_rollback_segments_ required*, in which case Oracle generates a message.
- If a private rollback segment cannot be brought online at instance startup, the startup fails and Oracle generates a message.

See Also: "Monitoring Rollback Segments" on page 14-6
Oracle8 SQL Reference

7

Overview of Locking Mechanisms

This chapter provides an overview of the locking mechanisms that are *internal* to the parallel server. The chapter is organized as follows:

- Differentiating Oracle Locking Mechanisms
- Cost of Locks
- Oracle Lock Names
- Coordination of Locking Mechanisms by the Integrated DLM

Differentiating Oracle Locking Mechanisms

This section covers the following topics:

- Overview
- Local Locks
- Instance Locks
- The LCKn Processes
- The LMON and LMD0 Processes

Overview

You must understand locking mechanisms if you are to effectively harness parallel processing and parallel database capabilities. You can influence each kind of locking through the way you set initialization parameters, administer the system, and design applications. If you do not use locks effectively, your system may spend so much time synchronizing shared resources that no speedup and no scaleup is achieved; your parallel system could even suffer performance degradation compared to a single instance system.

Locks are used for two main purposes in Oracle Parallel Server:

- to provide transaction isolation
- to provide cache coherency

Transaction locks are used to implement row level locking for transaction consistency. Row level locking is supported in both single instance Oracle and Oracle Parallel Server.

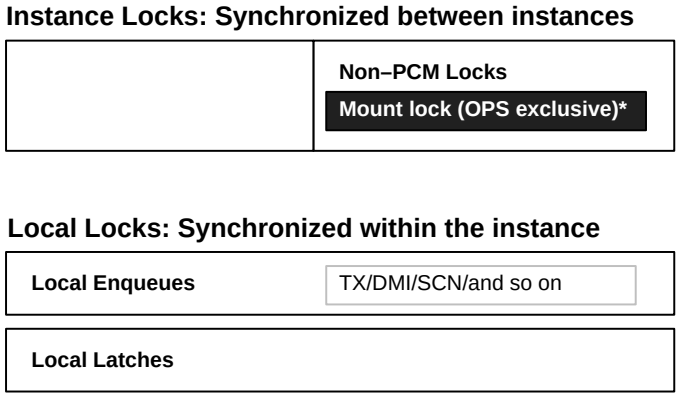
Instance locks (also commonly known as *distributed locks*) guarantee cache coherency. They ensure that data and other resources distributed among multiple instances belonging to the same database remain consistent. Instance locks include PCM and non-PCM locks.

See Also: *Oracle8 Concepts* for a detailed treatment of Oracle locks.
Chapter 8, “Integrated Distributed Lock Manager: Access to Resources” for more information.

Local Locks

Figure 7–1 shows latches and enqueues: locking mechanisms which are synchronized within a single instance. These are used in Oracle with or without the Parallel Server Option, and whether parallel server is enabled or disabled.

Figure 7–1 Locking Mechanisms: Oracle and OPS Disabled



* The mount lock is obtained if the Parallel Server Option has been linked in to your Oracle executable.

Latches

Latches are simple, low level serialization mechanisms to protect in-memory data structures in the SGA. Latches do not protect datafiles. They are entirely automatic, are held for a very short time, and can only be held in exclusive mode. Being local to the node, internal locks and latches do not provide internode synchronization.

Enqueues

Enqueues are shared memory structures which serialize access to resources in the database. These locks can be local to one instance or global to a database. They are associated with a session or transaction, and can be in any mode: shared, exclusive, protected read, protected write, concurrent read, concurrent write, or null.

Enqueues are held longer than latches, have more granularity and more modes, and protect more resources in the database. For example, if you request a table lock (a DML lock) you will receive an enqueue.

Certain enqueues are local to a single instance, when parallel server is disabled. But when parallel server is enabled, those enqueues can no longer be managed on the instance level: they need to be maintained on a system-wide level, managed by the Integrated Distributed Lock Manager (IDLM).

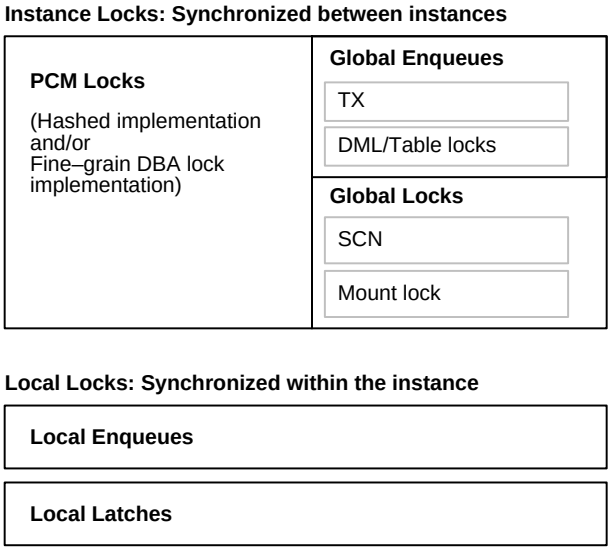
When parallel server is enabled, most of the local enqueues become global enqueues. This is reflected in Figure 7–1 and Figure 7–2. They will all appear as enqueues in the fixed tables—no distinction is made there between local and global enqueues. Global enqueues are handled in a distributed fashion.

Note: Transaction locks are simply a subset of enqueues.

Instance Locks

Figure 7–2 illustrates the instance locks that come into play when Oracle Parallel Server is enabled. In OPS implementations, the status of all Oracle locking mechanisms is tracked and coordinated by the Integrated DLM component.

Figure 7–2 Locking Mechanisms: Parallel Server Enabled



Instance locks (other than the mount lock) only come into existence if you start an Oracle instance with parallel server enabled. They synchronize between instances, communicating the current status of a resource among the instances of an Oracle Parallel Server.

Instance locks are held by background processes of instances, rather than by transactions. An instance *owns* an instance lock that *protects* a resource (such as a data block or data dictionary entry) when the resource enters its SGA.

The Integrated DLM component of Oracle handles locking only for resources accessed by more than one instance of a Parallel Server, to ensure cache coherency. The IDLM communicates requests for instance locks and the status of the locks between the lock processes of each instance. The V$DLM_LOCKS view lists information on all locks currently known to the IDLM.

Instance locks are of two types: parallel cache management (PCM) locks and non-PCM locks.

PCM Locks

Parallel cache management locks are instance locks that cover one or more data blocks (table or index blocks) in the buffer cache. PCM locks do not lock any rows on behalf of transactions. PCM locks are implemented in two ways:

hashed locking	This is the default implementation, in which PCM locks are statically assigned to blocks in the datafiles.
fine grain locking	In this implementation, PCM locks are assigned to blocks on a dynamic basis.

With hashed locking, an instance never *disowns* a PCM lock unless another instance asks for it. This minimizes the overhead of instance lock operations in systems that have relatively low contention for resources. With fine grain locking, once the block is released, the lock is released. (Note that non-PCM locks *are* disowned.)

Non-PCM Locks

Non-PCM locks of many different kinds control access to data and control files, control library and dictionary caches, and perform various types of communication between instances. These locks do not protect datafile blocks. Examples are DML enqueues (table locks), transaction enqueues, and DDL or dictionary locks. The System Change Number (SCN), and the mount lock are global locks, not enqueues.

Note: The context of Oracle Parallel Server causes most local enqueues to become global; they can still be seen in the fixed tables and views which show enqueues (such as V$LOCK). The V$LOCK table does not, however, show instance locks, such as SCN locks, mount locks, and PCM locks.

Many More PCM Locks Than Non-PCM Locks

Although PCM locks are typically far more numerous than non-PCM locks, there is still a substantial enough number of non-PCM locks that you must carefully plan adequate IDLM capacity for them. Typically 5% to 10% of locks are non-PCM. Non-PCM locks do not grow in volume the same way that PCM locks do.

The user controls PCM locks in detail by setting initialization parameters to allocate the number desired. However, the user has almost no control over non-PCM locks. You can try to eliminate the need for table locks by setting DML_LOCKS = 0 or by using the ALTER TABLE ENABLE / DISABLE TABLE LOCK command, but other non-PCM locks will still persist.

See Also: Chapter 16, "Ensuring IDLM Capacity for All Resources & Locks"

The LCK*n* Processes

With the Oracle Parallel Server, up to ten Lock processes (LCK0 through LCK9) provide inter-instance locking.

LCK processes manage most of the locks used by an instance and coordinate requests for those locks by other instances. LCK processes maintain all of the PCM locks (hashed or fine grain) and some of the non-PCM locks (such as row cache or library cache locks). LCK0 will handle PCM as well as non-PCM locks. Additional lock processes, LCK1 through LCK9, are available for systems that require exceptionally high throughput of instance lock requests; they will only handle PCM locks. Multiple LCK processes can improve recovery time and startup time.

Although instance locks are mainly handled by the LCK processes, some instance locks are directly acquired by other background or shadow foreground processes. In general, if a background process such as LCK owns an instance lock, it is for the whole instance. If a foreground process owns an instance lock, it is just for that particular process. For example, the log writer (LGWR) will get the SCN instance lock, the database writer (DBWR) will get the media recovery lock. The bulk of all these locks, however, are handled by the LCK processes.

Attention: Foreground processes obtain transaction locks—LCK processes do not. Transaction locks are associated with the session / transaction unit, not with the process.

See Also: *Oracle8 Concepts* for more information about the LCK*n* processes.

The LMON and LMD0 Processes

The LMON and LMD0 processes implement the global lock management subsystem of Oracle Parallel Server. LMON performs lock cleanup and lock invalidation after the death of an Oracle shadow process or another Oracle instance. It also reconfigures and redistributes the global locks as Oracle Parallel Server instances are started and stopped.

The LMD0 process handles remote lock requests for global locks (that is, lock requests originating from another instance for a lock owned by the current instance). All messages pertaining to global locks that are directed to an Oracle Parallel Server instance are handled by the LMD0 process of that instance.

Cost of Locks

To effectively implement locks, you need to carefully evaluate their relative expense. As a rule of thumb, latches are cheap; local enqueues are more expensive; instance locks and global enqueues are quite expensive. In general, instance locks and global enqueues have equivalent performance impact. (When parallel server is disabled, all enqueues are local; when parallel server is enabled, most are global.)

Table 7–1 dramatizes the *relative expense* of latches, enqueues, and instance locks. The elapsed time required per lock will vary by system--the values used in the "Actual Time Required" column are only examples.

Table 7–1 Comparing the Relative Cost of Locks

Class of Lock	Actual Time Required	Relative Time Required
Latches	1 microsecond	1 minute
Local Enqueues	1 millisecond	1000 minutes (16 hours)
Instance Locks (or Global Enqueues)	1/10 second	100,000 minutes (69 days)

Microseconds, milliseconds, and tenths of a second all sound like negligible units of time. However, if you *imagine* the cost of locks using *grossly exaggerated values* such as those listed in the "Relative Time Required" column, you can grasp the need to carefully calibrate the use of locks in your system and applications. In a big OLTP situation, for example, unregulated use of instance locks would be impermissible. Imagine waiting hours or days to complete a transaction in real life!

Stored procedures are available for analyzing the number of PCM locks an application will use if it performs particular functions. You can set values for your initial-

ization parameters and then call the stored procedure to see the projected expenditure in terms of locks.

See Also: Chapter 15, "Allocating PCM Instance Locks".
Chapter 16, "Ensuring IDLM Capacity for All Resources & Locks".

Oracle Lock Names

This section covers the following topics:

- Lock Name Format
- PCM Lock Names
- Non-PCM Lock Names

Lock Name Format

All Oracle enqueues and instance locks are named using one of the following formats:

type ID1 ID2

or *type*, *ID1*, *ID2*

or *type* (*ID1*, *ID2*)

where

type	A two-character type name for the lock type, as described in the V$LOCK table, and listed in Table 7–2 and Table 7–3.
ID1	The first lock identifier, used by the IDLM. The convention for this identifier differs from one lock type to another.
ID2	The second lock identifier, used by the IDLM. The convention for this identifier differs from one lock type to another.

For example, a space management lock might be named ST 1 0. A PCM lock might be named BL 1 900.

The V$LOCK table contains a list of local and global Oracle enqueues currently held or requested by the local instance. The "lock name" is actually the name of the resource; locks are taken out against the resource.

PCM Lock Names

All PCM locks are Buffer Cache Management locks.

Table 7–2 PCM Lock Type and Name

Type	Lock Name
BL	Buffer Cache Management

The syntax of PCM lock names is *type ID1 ID2*, where

type	is always BL (because PCM locks are buffer locks)
ID1	is the block class
ID2	For fixed locks, *ID2* is the lock element (LE) index number obtained by hashing the block address (see the V$LOCK_ELEMENT fixed view). For releasable locks, *ID2* is the database address of the block.

Sample PCM lock names are:

BL (1, 100)	This is a data block with lock element 100.
BL (4, 1000)	This is a segment header block with lock element 1000.
BL (27, 1)	This is an undo segment header with rollback segment #10. The formula for the rollback segment is 7 + (10 * 2).

Non-PCM Lock Names

Non-PCM locks have many different names.

Table 7–3 Non-PCM Lock Types and Names

Type	Lock Name
CF	Controlfile Transaction
CI	Cross-Instance Call Invocation
DF	Datafile
DL	Direct Loader Index Creation
DM	Database Mount
DX	Distributed Recovery
FS	File Set
KK	Redo Log “Kick”
IN	Instance Number
IR	Instance Recovery
IS	Instance State
MM	Mount Definition
MR	Media Recovery
IV	Library Cache Invalidation
L[A-P]	Library Cache Lock
N[A-Z]	Library Cache Pin
Q[A-Z]	Row Cache
PF	Password File
PR	Process Startup
PS	Parallel Slave Synchronization
RT	Redo Thread
SC	System Commit Number
SM	SMON
SN	Sequence Number
SQ	Sequence Number Enqueue

Table 7–3 Non-PCM Lock Types and Names

Type	Lock Name
SV	Sequence Number Value
ST	Space Management Transaction
TA	Transaction Recovery
TM	DML Enqueue
TS	Temporary Segment (also Table-Space)
TT	Temporary Table
TX	Transaction
UL	User-Defined Locks
UN	User Name
WL	Begin written Redo Log
XA	Instance Registration Attribute Lock
XI	Instance Registration Lock

See Also: *Oracle8 Reference* for descriptions of all these non-PCM locks.

Coordination of Locking Mechanisms by the Integrated DLM

The Integrated DLM component is a *distributed resource manager* that is *internal* to the Oracle Parallel Server. This section explains how the IDLM coordinates locking mechanisms that are *internal* to Oracle. Chapter 8, "Integrated Distributed Lock Manager: Access to Resources" presents a detailed description of IDLM features and functions.

This section covers the following topics:

- The Integrated DLM Tracks Lock Modes
- The Instance Maps Database Resources to Integrated DLM Resources
- How IDLM Locks and Instance Locks Relate
- The Integrated DLM Provides One Lock Per Instance on a Resource

The Integrated DLM Tracks Lock Modes

In Oracle Parallel Server implementations, the Integrated DLM facility keeps an inventory of all the Oracle instance locks and global enqueues held against the resources of your system. It acts as a referee when conflicting lock requests arise.

In Figure 7–3 the IDLM is represented as an inventory sheet listing resources and the current status of locks on each resource across the parallel server. Locks are represented as follows: S for shared mode, N for null mode, X for exclusive mode.

Figure 7–3 The Integrated DLM Inventory of Oracle Resources and Locks

Integrated DLM

Resource	Locks
BL 1, 100	S
BL 1, 101	SSSNNN
BL 4, 3000	X
BL 4, 3001	SSS
BL 6, 100	NNN
BL 6, 101	X
BL 8, 3000	X
BL 8, 3001	N
BL 9, 4000	N

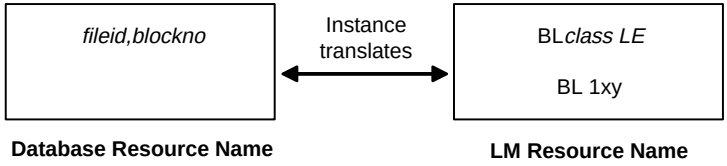
This inventory includes all instances. For example, resource BL 1, 101 is held by three instances with shared locks and three instances with null locks. Since the table reflects up to 6 locks on one resource, at least 6 instances are evidently running on this system.

The Instance Maps Database Resources to Integrated DLM Resources

Oracle database resources are mapped to IDLM resources, with the necessary mapping performed by the instance. For example, a hashed lock on an Oracle database block with a given data block address (such as file 2 block 10) becomes translated as a BL resource with the class of the block and the lock element number (such as BL 9 1). The data block address (DBA) is translated from the Oracle resource level to the IDLM resource level; the hashing function used is dependent on GC_* parameter settings. The IDLM resource name identifies the physical resource in views such as V$LOCK.

Note: For DBA fine grain locking, the database address is used as the second identifier, rather than the lock element number.

Figure 7–4 Database Resource Names Corresponding to IDLM Resource Names

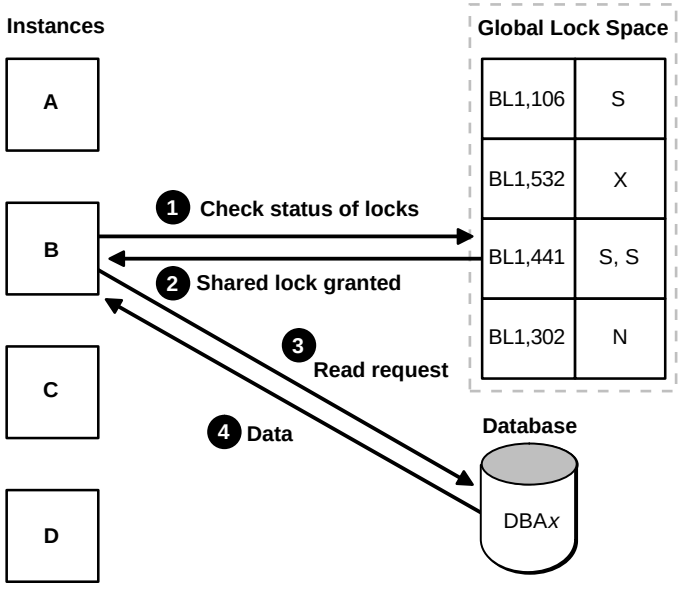


How IDLM Locks and Instance Locks Relate

Figure 7–5 illustrates the way in which IDLM locks and PCM locks relate. For instance B to read the value of data at data block address *x*, it must first check for locks on that data. The instance translates the block's database resource name to the IDLM resource name, and asks the IDLM for a shared lock in order to read the data.

As illustrated in the following conceptual diagram, the IDLM checks all the outstanding locks on the granted queue and determines that there are already two shared locks on the resource BL1,441. Since shared locks are compatible with read-only requests, the IDLM grants a shared lock to Instance B. The instance then proceeds to query the database to read the data at data block address *x*. The database returns the data.

Figure 7–5 The IDLM Checks Status of Locks



Note: The global lock space is managed in distributed fashion by the LMDs of all the instances cooperatively.

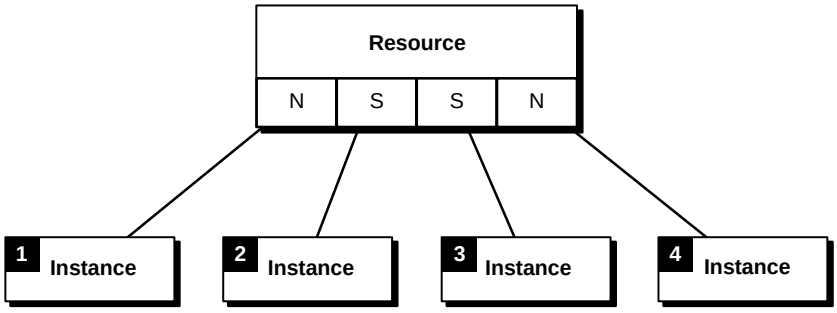
If the required block already had an exclusive lock on it by another instance, then Instance B would have to wait for this to be released. The IDLM would place on the convert queue the shared lock request from Instance B. The IDLM would notify the instance when the exclusive lock was removed, and then grant its request for a shared lock.

The term *IDLM lock* refers simply to the IDLM's notations for tracking and coordinating the outstanding locks on a resource.

The Integrated DLM Provides One Lock Per Instance on a Resource

The IDLM provides one lock per instance on a PCM resource. As illustrated in Figure 7–6, if you have a four-instance system and require a buffer lock on a single resource, you will actually end up with four locks—one per instance.

Figure 7–6 Resources Have One Lock Per Instance



The number of non-PCM locks may depend on the type of lock.

See Also: Chapter 10, “Non-PCM Instance Locks”

8

Integrated Distributed Lock Manager: Access to Resources

This chapter explains the role of the Integrated Distributed Lock Manager (Integrated DLM, or IDLM) in controlling access to resources in a parallel server. The chapter is organized as follows:

- What Is the Integrated Distributed Lock Manager?
- The Integrated DLM Grants and Converts Resource Lock Requests
- Integrated DLM Lock Modes: Resource Access Rights
- Integrated DLM Features

What Is the Integrated Distributed Lock Manager?

The Integrated Distributed Lock Manager component of Oracle8 maintains a list of system resources and provides locking mechanisms to control allocation and modification of Oracle resources. IDLM resources are logical concepts, structures of data. The IDLM does *not* control access to tables or anything in the database itself. Every process interested in the database resource protected by the IDLM must open a lock on the resource.

Oracle Parallel Server uses the IDLM facility to coordinate concurrent access to resources, such as data blocks and rollback segments, across multiple instances. The Integrated Distributed Lock Manager facility has replaced the external Distributed Lock Manager which was used in earlier releases of Oracle.

The Integrated DLM Grants and Converts Resource Lock Requests

- Lock Requests Are Queued
- Asynchronous Traps (ASTs) Communicate Lock Request Status
- Persistent Resources Ensure Efficient Recovery
- Lock Requests Are Converted and Granted

The IDLM coordinates lock requests, ensuring compatibility of access rights to the resources. In this process the IDLM tracks all lock requests. Requests for available resources are granted and the access rights granted are tracked. Requests for resources not currently available are tracked, and access rights are granted when the resource does become available. The IDLM keeps an inventory of all these lock requests, and communicates their status to the users and processes involved.

Lock Requests Are Queued

The IDLM maintains two queues for lock requests:

convert queue	If a lock request cannot be granted immediately, it is placed in the convert queue, where waiting lock requests are tracked.
granted queue	Lock requests that have been granted are tracked in the granted queue.

Asynchronous Traps (ASTs) Communicate Lock Request Status

To communicate the status of lock requests, the IDLM uses two types of asynchronous traps (ASTs) or interrupts:

acquisition AST	When the lock is obtained in the requested mode, an acquisition AST (a "wakeup call") is sent to tell the requestor that the requestor owns the lock.
blocking AST	When a process requests a certain mode of lock on a resource, the IDLM sends a blocking AST to notify processes which currently own locks on that resource in incompatible modes. (Shared and exclusive modes, for example, are incompatible.) Upon notification, owners of locks can relinquish them to permit access by the requestor.

Persistent Resources Ensure Efficient Recovery

The term "persistent resource" refers to the ability of a resource to maintain a particular state if all processes or groups holding a lock on it have died abnormally. This contrasts with normal resources, which cease to exist when there are no longer any owners of locks on that resource, regardless of how they exited.

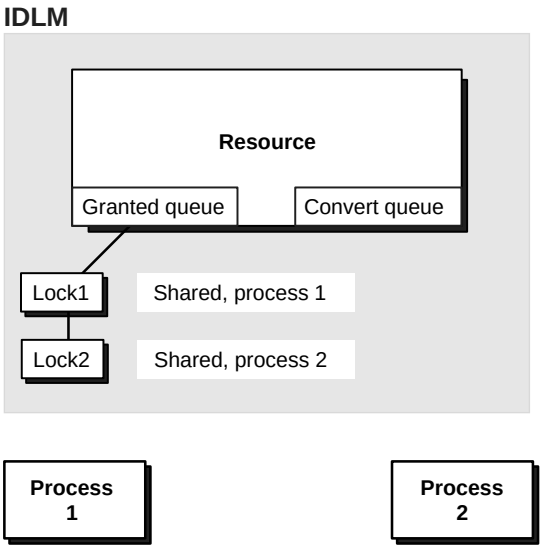
Resource persistence is necessary for fine grain locking. It ensures that the database resources protected by these locks can be recovered correctly and efficiently.

Note: Not all resource information is kept after failures, only adequate information to protect resources during recovery.

Lock Requests Are Converted and Granted

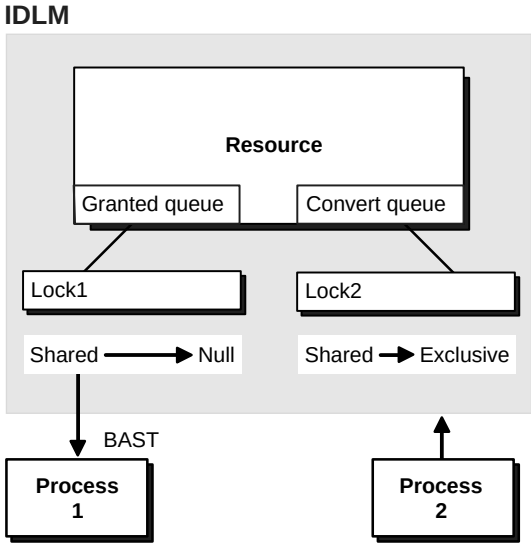
The following figures show how the IDLM handles lock requests. In Figure 8–1, shared lock request #1 has been granted on the resource to process 1, and shared lock request#2 has been granted to process 2. The locks are tracked in the granted queue. When a request for an exclusive lock is made by process 2, it must wait in the convert queue.

Figure 8–1 The IDLM Granted and Convert Queues



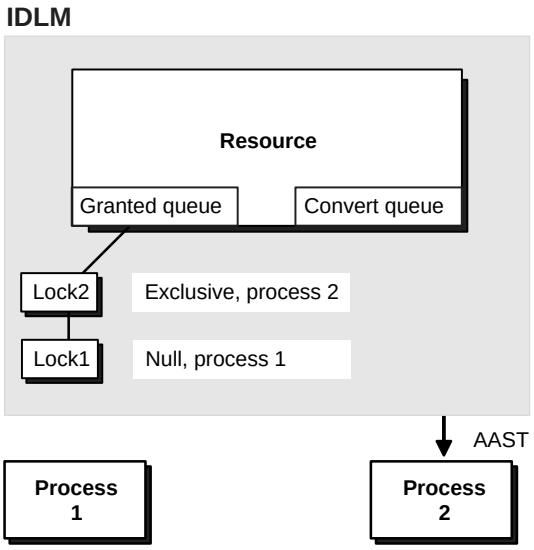
As illustrated in Figure 8–2, the IDLM sends a blocking AST to Process 1, the owner of the shared lock, notifying it that a request for an exclusive lock is waiting. When the shared lock is relinquished by Process 1, it is converted to a null mode lock, or released.

Figure 8–2 Blocking AST



An acquisition AST is then sent to wake up Process 2, the requestor of the exclusive lock. The exclusive lock is granted and it is converted to the granted queue. This is illustrated in Figure 8–3.

Figure 8–3 Acquisition AST



Integrated DLM Lock Modes: Resource Access Rights

Locks are used to obtain various rights to a resource. A lock may be initially created on a resource with no access rights granted. Later, a process will convert a lock to obtain new access rights.

Figure 8–4 illustrates the levels of access rights or “lock modes” which are available through the IDLM.

Figure 8–4 IDLM Lock Modes: Levels of Access

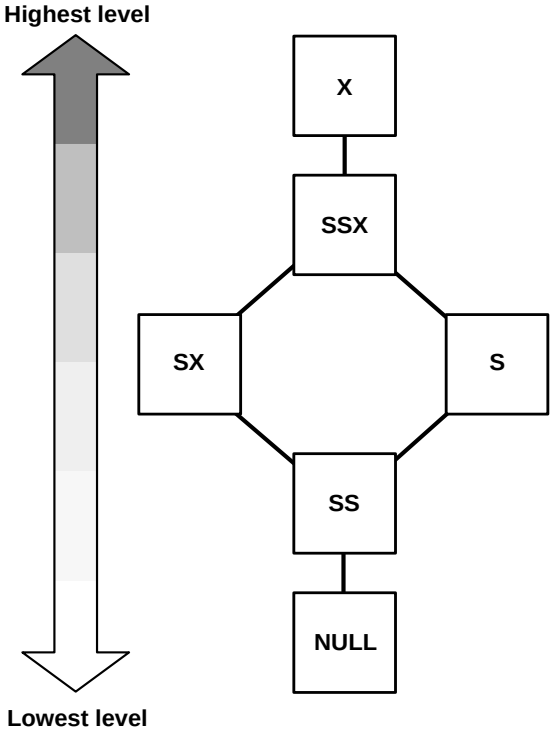


Table 8–1 Lock Mode Names

Oracle Mode	Summary	Description
NULL	Null mode. No lock is on the resource	Holding a lock at this level conveys no access rights. Typically, a lock is held at this level to indicate that a process is interested in a resource, or it is used as a place holder. Once created, null locks ensure that the requestor always has a lock on the resource; there is no need for the IDLM to be constantly creating and destroying locks when ongoing access is needed.
SS	Sub-shared mode (concurrent read). Read; there may be writers and other readers	When a lock is held at this level, the associated resource can be read in an unprotected fashion: other processes can read and write the associated resource.
SX	Shared exclusive mode (concurrent write). Write; there may be other readers and writers	When a lock is held at this level, the associated resource can be read or written in an unprotected fashion: other processes can both read and write the resource.
S	Shared mode (protected read). Read; no writers are allowed	When a lock is held at this level, no process can write the associated resource. Multiple processes can read the resource. This is the traditional shared lock. In shared mode, any number of users can have simultaneous read access to the resource. Shared access is appropriate for read operations.
SSX	Sub-shared exclusive mode (protected write). One writer only; there may be readers	Only one process can hold a lock at this level. This allows a process to modify a resource without allowing any other process to modify the resource at the same time. Other processes can perform unprotected reads. The traditional update lock.
X	Exclusive mode. Write; no other access is allowed	When a lock is held at this level, it grants the holding process exclusive access to the resource. No other process may read or write the resource. This is the traditional exclusive lock.

Integrated DLM Features

This section describes the following features of the Integrated DLM:

- Distributed Architecture
- Fault Tolerance
- Lock Mastering
- Deadlock Detection
- Lamport SCN Generation
- Group-owned Locks
- Persistent Resources
- Memory Requirements
- Support for MTS and XA
- Views to Monitor Integrated DLM Statistics

Distributed Architecture

The IDLM maintains a database of resources and locks held on these resources in different modes. This lock database resides in volatile memory, and is distributed. The Integrated DLM has a distributed architecture. In the distributed architecture each node in the cluster (or each Oracle Parallel Server instance of an Oracle database) participates in global lock management and manages a piece of the global lock database. The lock database is distributed among all the participants. This distributed lock management scheme provides fault tolerance and enhanced runtime performance.

Fault Tolerance

The Integrated DLM is fault tolerant: it provides continual service and maintains the integrity of the lock database in the event of multiple node and Oracle Parallel Server instance failures. Instance reconfiguration may cause a brief delay. A database is accessible as long as there is at least one OPS instance that is active on that database after recovery completes. Fault tolerance also enables OPS instances to be started and stopped at any time, in any order.

Lock Mastering

In a distributed system the IDLM must maintain information about the locks on all nodes that are interested in a given resource. In this situation, the IDLM usually nominates one node to manage *all* relevant information about the resource and its locks. This is called the master node.

Two methods of lock mastering are currently available:

static hashing	The resource name is hashed to one of the Oracle Parallel Server instances, which acts as the master for this resource. This scheme results in an even arbitrary distribution of resources across all available nodes. In this scheme, the directory node is also the master node for a resource.
dynamic hashing	The node that opens the first lock on a resource is nominated to be the master node for this resource. This information is stored at the directory node for the resource. In this scheme, the directory node may not be the master node for a resource.

The Integrated DLM optimizes the method of lock mastering to use in each situation. The method of lock mastering has an impact on system performance, during normal runtime activity as well as during instance startup. Performance is optimized when a resource is mastered locally. When a resource is mastered remotely, all conflicting accesses to this resource result in a message to the master node for this resource, which results in internode message traffic and impacts system performance.

Associated with every resource is a directory node and a master node. The directory node is derived from the resource name (the resource name is hashed to one of the active nodes in the cluster). The directory node maintains information about the node on which a resource is mastered. The node that masters a resource is the master node for the resource. Each node acts as the directory node for a subset of resources. In this sense, the directory service is distributed across all nodes in the cluster.

Deadlock Detection

IDLM performs distributed deadlock detection, in which all deadlock sensitive locks and resources can be distributed.

Lamport SCN Generation

Oracle Parallel Server uses the fast and scalable Lamport SCN generation scheme, which can generate SCNs in parallel on all instances.

See Also: "Lamport SCN Generation" on page 4-7.

Group-owned Locks

Group-based locking provides dynamic ownership: a single lock can be shared by two or more processes belonging to the same group. Processes in the same group can share and / or touch the lock without going to the IDLM grant and convert queues.

See Also: "Support for MTS and XA" on page 8-12.

Persistent Resources

The Integrated DLM provides for persistent resources. Resources maintain their state even if all processes or groups holding a lock on it have died abnormally.

See Also: "Persistent Resources Ensure Efficient Recovery" on page 8-3.

Memory Requirements

The user-level Integrated DLM can normally allocate as many resources as you request; your process size, however, will increase accordingly. This is because you are mapping the shared memory where locks and resources reside into your address space. The process address space can become very large.

Make sure that the IDLM is configured to support all the resources which your application will require.

Support for MTS and XA

Oracle Parallel Server uses two forms of lock ownership:

per-process ownership	Locks are commonly process-owned: that is, if one process owns a lock exclusively, then no other process can touch the lock.
group-based ownership	With group-based locking, ownership becomes dynamic: a single lock can be exchanged by two or more processes belonging to the same group. Processes in the same group can exchange and/or touch the lock without going to the IDLM grant and convert queues.

Group-based locking is an important IDLM feature for Oracle multi-threaded server (MTS) and XA library functionality.

MTS	Group-based locking is used for Oracle MTS configurations. Without it, sessions could not migrate between shared server processes. In addition, load balancing may be affected, especially with long running transactions.
XA libraries	With Oracle XA libraries, multiple sessions or processes can work on the transaction; they therefore need to exchange the same locks, even in exclusive mode. With group-based lock ownership, processes can exchange access to an exclusive resource.

Views to Monitor Integrated DLM Statistics

Four dynamic performance views are available to monitor Integrated DLM statistics. These are:

Table 8–2 Views to Monitor Integrated DLM Statistics

View	Description
V$DLM_CONVERT_LOCAL	Shows the convert time for local lock convert operations
V$DLM_CONVERT_REMOTE	Shows the convert time for remote lock convert operations
V$DLM_LATCH	Contains statistics about Integrated DLM latch performance. For each type of latch this table shows total gets and immediate gets. Ideally, IMM_GETS/TTL_GETS should be as close to 1 as possible.
V$DLM_LOCKS	Contains statistics about all locks currently known to the IDLM which are being blocked or are blocking others.
V$DLM_MISC	Contains various Integrated DLM statistics.

See Also: *Oracle8 Reference* for a complete description of these dynamic performance views.

9

Parallel Cache Management Instance Locks

The planning and allocation of PCM locks is one of the most complex tasks facing the Oracle Parallel Server database administrator. This chapter provides a conceptual overview of PCM locks.

This chapter covers the following topics:

- PCM Locks and How They Work
- How Initialization Parameters Control Blocks and PCM Locks
- Two Methods of PCM Locking: Fixed and Releasable
- How Locks Are Assigned to Blocks
- Examples: Mapping Blocks to PCM Locks

See Also: Chapter 15, “Allocating PCM Instance Locks”, for details on how to plan and assign these locks.
Chapter 8, “Integrated Distributed Lock Manager: Access to Resources” for more information about the IDLM facility.

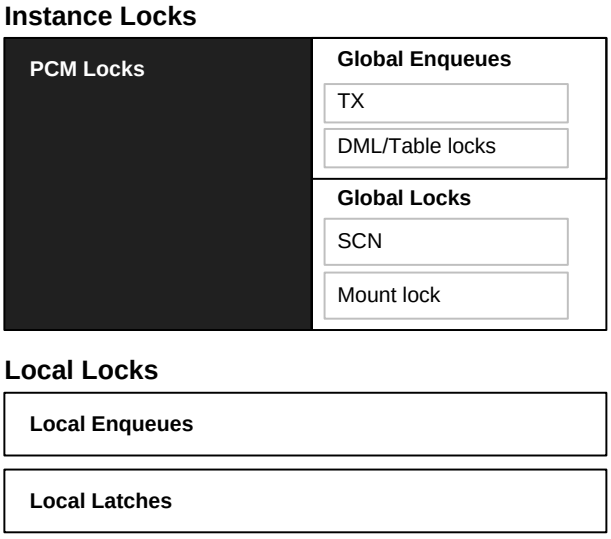
PCM Locks and How They Work

This section covers the following topics:

- What PCM Locks Are
- Allocation and Release of PCM Locks
- How PCM Locks Work
- Number of Blocks per PCM Lock
- Pinging: Signaling the Need to Update
- Lock Mode and Buffer State

Figure 9–1 highlights PCM locks in relation to other locks used in Oracle.

Figure 9–1 Oracle Locking Mechanisms: PCM Locks



What PCM Locks Are

Parallel cache management locks, or *PCM locks*, are the instance locks which manage the locking of blocks in datafiles. They can cover one or more blocks of any class: data blocks, index blocks, undo blocks, segment headers, and so on. Oracle Parallel Server uses these instance locks to coordinate access to shared resources. The Integrated DLM maintains the status of the instance locks.

PCM locks ensure cache coherency by forcing instances to acquire a lock before modifying or reading any database block. PCM locks allow only one instance at a time to modify a block. If a block is modified by an instance, the block must first be written to disk before another instance can acquire the PCM lock, read the block, and modify it.

PCM locks use the minimum amount of communication to ensure cache coherency. The amount of cross-instance activity—and the corresponding performance of a parallel server—is evaluated in terms of *pings*. A ping occurs each time a block must be written to disk by one instance so that another instance can read it.

Note that busy systems can have a great deal of locking activity, but do not necessarily have pinging. If data is well partitioned, then the locking will be local to each node—therefore pinging will not occur.

Allocation and Release of PCM Locks

For optimal performance, the Oracle Parallel Server administrator must allocate PCM locks to datafiles. You do this by specifying values for initialization parameters which are read at startup of the database. Chapter 15, “Allocating PCM Instance Locks” describes this procedure in detail.

You use the initialization parameter GC_FILES_TO_LOCKS to specify the number of PCM locks which cover the data blocks in a data file or set of data files. The smallest granularity is one PCM lock per datablock; this is the default. PCM locks usually account for the greatest proportion of instance locks in a parallel server.

Four types of PCM locks can be allocated. They differ in the method by which they are allocated, and in whether or not they are released.

Allocation of Releasable Fine Grain Locks

Fine grain PCM locks are acquired and released as needed. Since they are allocated only as required, the instance can start up much faster than with hashed locks. An IDLM resource is created and an IDLM lock is obtained only when a user actually requests a block. Once a fine grain lock has been created, it can be converted to various modes as required by various instances.

Fine grain locks are releasable: an instance can give up all references to the resource name during normal operation. The IDLM resource is released when it is required for reuse for a different block. This means that sometimes no instance holds a lock on a given resource.

Allocation of Fixed Hashed Locks

Hashed locks are preallocated and statically hashed to blocks at startup time. The first instance which starts up creates an IDLM resource and an IDLM lock (in null mode) on the IDLM resource for each hashed PCM lock. The first instance initializes each lock. The instance then proceeds to convert IDLM locks to other modes as required. When a second instance requires a particular IDLM lock, it waits until the lock is available and then converts the lock to the mode required.

By default, hashed PCM locks are never released; each will stay in the mode in which it was last requested. If the lock is required by another instance, it is converted to null mode. These locks are deallocated only at instance shutdown.

Allocation of Releasable Hashed Locks

You can specify releasable hashed PCM locks by using the R option with the GC_FILES_TO_LOCKS parameter. Releasable hashed PCM locks are taken from the pool of GC_RELEASABLE_LOCKS.

Allocation of Fixed Fine Grain Locks

You can also allocate fixed locks in a fine grained manner. For example, you could set 50,000 PCM locks for a particular file and thus provide 1 fixed lock for each block.

See Also: "GC_FILES_TO_LOCKS Syntax" on page 15-8 for a detailed explanation of how to set the GC_FILES_TO_LOCKS parameter.

How PCM Locks Work

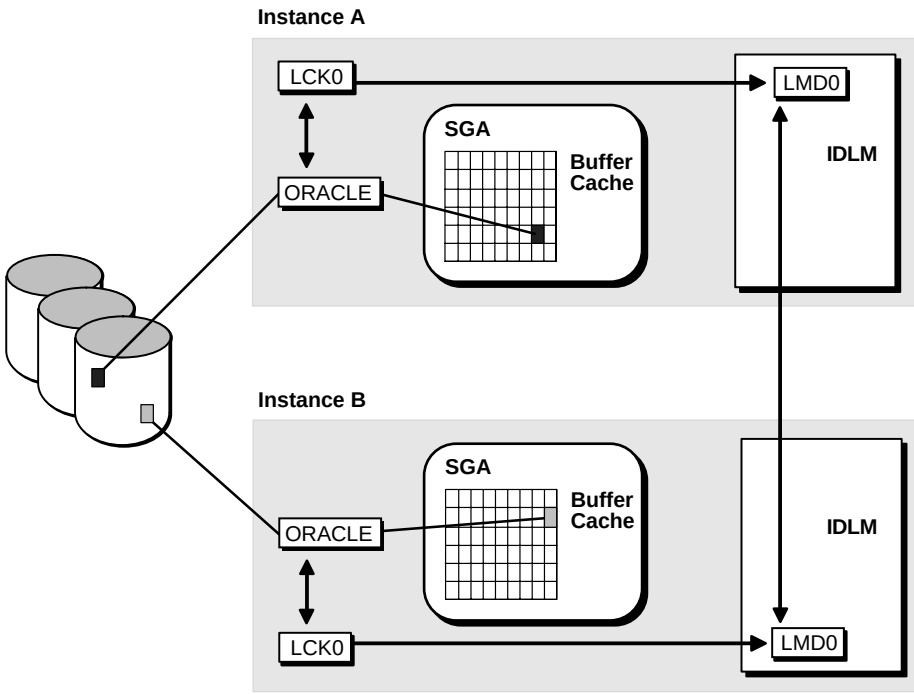
Fixed PCM locks are initially acquired in null mode. All specified hashed locks are allocated at instance startup, and deallocated at instance shutdown. Because of this, hashed locks entail more overhead and longer startup time than fine grain locks. The advantage of fixed hashed PCM locks, however, is that they do not need to be continually acquired and released.

Releasable PCM locking is more dynamic than fixed hashed locking. For example, if you set GC_RELEASABLE_LOCKS to 10000 you can obtain up to ten thousand fine grain PCM locks. However, locks are allocated only as needed by the IDLM. At

startup Oracle allocates lock elements, which are obtained directly in the requested mode (normally shared or exclusive mode).

Figure 9–2 illustrates the way PCM locks work. When instance A reads in the black block for modifications, it obtains the PCM lock for the black block. The same scenario occurs with the shaded block and Instance B. If instance B requires the black block, the block must be written to disk because instance A has modified it. The ORACLE process communicates with the LMD processes in order to obtain the instance lock from the IDLM.

Figure 9–2 How PCM Locks Work



PCM Locks Are Owned by Instance LCK Processes

Each instance has at least one LCK background process. If multiple LCK processes exist within the same instance, the PCM locks are divided among the LCK processes. This means that each LCK process is only responsible for a subset of the PCM locks.

Locks Are Converted from One Mode to Another

A PCM lock is "owned" or controlled by an instance when a block covered by that lock (in shared or exclusive mode) enters the buffer cache belonging to the instance.

LCK processes maintain PCM locks on behalf of the instance. The LCK processes obtain and convert hashed PCM locks; they obtain, convert, and release fine grained PCM locks.

Locks Can Be Owned by Multiple Instances

A PCM lock owned in shared mode is not disowned by an instance if another instance also requests the PCM lock in shared mode. Thus, two instances may have the same data block in their buffer caches because the copies are shared (no writes occur). Different data blocks covered by the same PCM lock can be contained in the buffer caches of separate instances. This can occur if all the different instances request the PCM lock in shared mode.

Number of Blocks per PCM Lock

Typically, a PCM lock covers a number of data blocks. The number of PCM locks assigned to datafiles and the number of data blocks in those datafiles determines the number of data blocks covered by a single PCM lock.

- If GC_FILES_TO_LOCKS is *not* set for a file, then releasable locks are used with one lock for each block.
- If GC_FILES_TO_LOCKS *is* set for a file, then the number of blocks per PCM lock can be expressed as follows, on a per file level. (This example assumes values of GC_FILES_TO_LOCKS = 1:300,2:200,3-5:100.)

$$\text{File 1: } \frac{\textit{file1 blocks}}{\text{300 locks}}$$

$$\text{File 2: } \frac{\textit{file2 blocks}}{\text{200 locks}}$$

$$\text{File 3: } \frac{\text{sum } \textit{(file3, file4, file5 blocks)}}{\text{100 locks}}$$

If the size of each file, in blocks, is a multiple of the number of PCM locks assigned to it, then each hashed PCM lock covers exactly the number of data blocks given by the equation.

If the file size is *not* a multiple of the number of PCM locks, then the number of data blocks per hashed PCM lock can vary by one for that datafile. For example, if you assign 400 PCM locks to a datafile which contains 2,500 data blocks, then 100 PCM locks cover 7 data blocks each and 300 PCM locks cover 6 blocks. Any datafiles not specified in the GC_FILES_TO_LOCKS initialization parameter use the remaining PCM locks.

If *n* files share the same hashed PCM locks, then the number of blocks per lock can vary by as much as *n*. If you assign locks to individual files, either with separate clauses of GC_FILES_TO_LOCKS or by using the keyword EACH, then the number of blocks per lock does not vary by more than one.

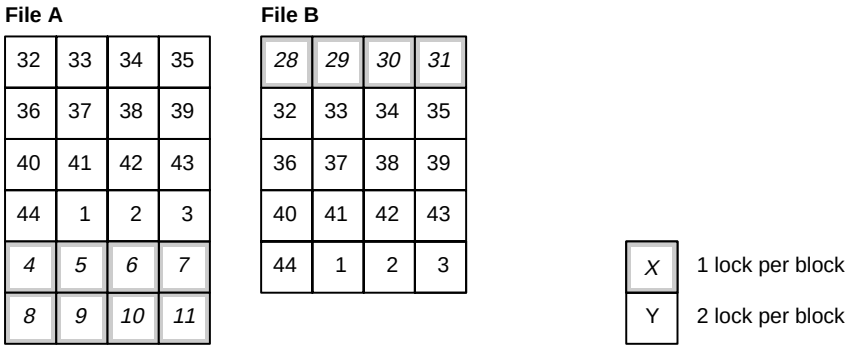
If you assign hashed PCM locks to a set of datafiles collectively, then each lock usually covers one or more blocks in each file. Exceptions can occur when you specify contiguous blocks (using the "!*blocks*" option) or when a file contains fewer blocks than the number of locks assigned to the set of files.

Example

The following example illustrates how hashed PCM locks can cover multiple blocks in different files. Figure 9–3 assumes 44 PCM locks assigned to 2 files which have a total of 44 blocks. GC_FILES_TO_LOCKS is set to A,B:44

Block 1 of a file does not necessarily begin with lock 1; a hashing function determines which lock a file begins with. In file A, which has 24 blocks, block 1 hashes to lock 32. In file B, which has 20 blocks, block 1 hashes to lock 28.

Figure 9–3 Hashed PCM Locks Covering Blocks in Multiple Files



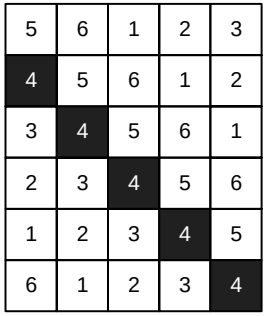
In Figure 9–3, locks 32 through 44 and 1 through 3 are used to cover 2 blocks each. Locks 4 through 11 and 28 through 31 cover 1 block each; and locks 12 through 27 cover no blocks at all!

In a worst case scenario, if two files hash to the same lock as a starting point, then all the common locks will cover two blocks each. If your files are large and have multiple blocks per lock (on the order of 100 blocks per lock), then this is not an important issue.

Periodicity of Hashed PCM Locks

Note also the *periodicity* of PCM locks. Figure 9–4 shows a file of 30 blocks which is covered by 6 PCM locks. This file has hashed to begin with lock number 5. As suggested by the shaded blocks covered by PCM lock number 4, use of each lock forms a pattern over the blocks of the file.

Figure 9–4 Periodicity of Hashed PCM Locks



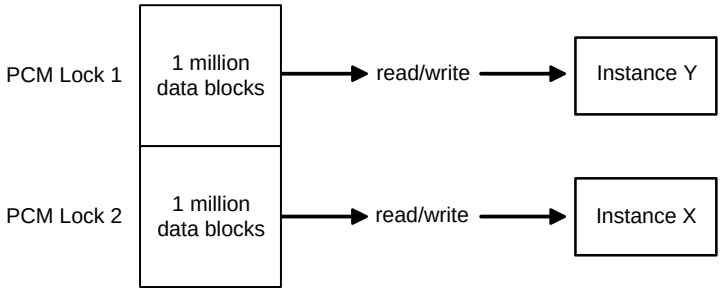
Pinging: Signaling the Need to Update

On a parallel server, a particular data block can only be modified by a single instance at a time. If one instance modifies a data block which another instance needs, then each instance's locks on the data block must be converted accordingly. The first instance must write the block to disk before the other instance can read it. This is known as *pinging* a block. The LCK process uses the Integrated DLM facility to signal this need between the two instances.

Data blocks are pinged when a block that is held in the exclusive current (XCUR) state in the buffer cache of one instance, is needed by a different instance. If an instance has a block in SHARE mode, it will be pinged if another instance needs it XCUR. In some cases, therefore, the number of PCM locks covering data blocks may have little impact on whether a block gets pinged. You can have lost a PCM lock on a block and still have a row lock on it: pinging is independent of whether a commit has occurred. You can modify a block, but whether or not it is pinged is independent of whether you have made the commit.

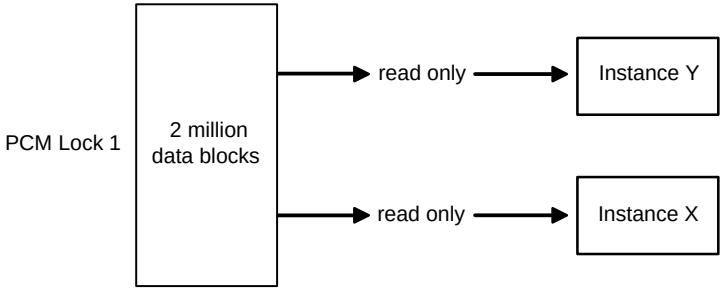
If you have partitioned data across instances and are doing updates, you can have a million blocks each on the different instances, each covered by one PCM lock, and still not have any forced reads or forced writes. As shown in Figure 9–5, if a single PCM lock covers one million data blocks in a table which are read / write into the SGA of instance X, and another single PCM lock covers another million data blocks in the table which are read / write into the SGA of instance Y, then regardless of the number of updates, there will be no forced reads or writes on data blocks between instance X and instance Y.

Figure 9–5 Partitioning Data to Avoid Pinging



With read-only data, both instance X and instance Y can hold the PCM lock in shared mode, and no pinging will take place. This scenario is illustrated in Figure 9–6.

Figure 9–6 No Pinging of Read-only Data



Lock Mode and Buffer State

The state of a block in the buffer cache relates directly to the mode of the lock held upon it. For example, if a buffer is in exclusive current (XCUR) state, you know that an instance owns the PCM lock in exclusive mode. There can be only one XCUR version of a block in the database, but there can be multiple SCUR versions. To perform a modification, a process must get the block in XCUR mode.

Finding the State of a Buffer

To learn the state of a buffer, check the STATUS column of the V$BH dynamic performance table. This table provides information about each buffer header.

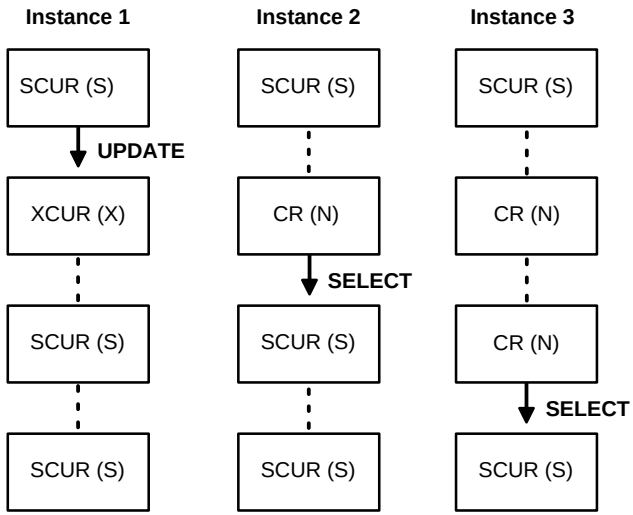
Table 9–1 PCM Lock Mode and Buffer State

PCM Lock Mode	Buffer State Name	Description
X	XCUR	Instance has an EXCLUSIVE lock for this buffer
S	SCUR	Instance has a SHARED lock for this buffer
N	CR	Instance has a NULL lock for this buffer

How Buffer State and Lock Mode Change

Figure 9–7 shows how buffer state and lock mode change as instances perform various operations on a given buffer. Lock mode is shown in parentheses.

Figure 9–7 How State of Buffer and Lock Mode Change



In Figure 9–7, the three instances start out with blocks in shared current mode, and shared locks. When Instance 1 performs an update on the block, its lock mode on

the block changes to exclusive mode (X). The shared locks owned by Instance 2 and Instance 3 convert to null mode (N). Meanwhile, the block state in Instance 1 becomes XCUR, and in Instance 2 and Instance 3 becomes CR. These lock modes are compatible. Similar conversions of lock mode and block state occur when Instance 2 performs a SELECT operation on the block, and when Instance 3 performs a SELECT operation on it.

Lock Modes May Be Compatible or Incompatible

When one process owns a lock in a given mode, another process requesting a lock in any particular mode succeeds or fails as shown in the following table.

Table 9–2 Lock Mode Compatibility

Lock Requested: / Lock Owned	Null	SS	SX	S	SSX	X
NULL	SUCCEED	SUCCEED	SUCCEED	SUCCEED	SUCCEED	SUCCEED
SS	SUCCEED	SUCCEED	SUCCEED	SUCCEED	SUCCEED	FAIL
SX	SUCCEED	SUCCEED	SUCCEED	FAIL	FAIL	FAIL
S	SUCCEED	SUCCEED	FAIL	SUCCEED	FAIL	FAIL
SSX	SUCCEED	SUCCEED	FAIL	FAIL	FAIL	FAIL
X	SUCCEED	FAIL	FAIL	FAIL	FAIL	FAIL

How Initialization Parameters Control Blocks and PCM Locks

This section explains how certain initialization parameters control blocks and PCM locks.

- GC_* Initialization Parameters
- Handling Data Blocks

GC_* Initialization Parameters

PCM locks are controlled by the initialization parameters listed below. Be sure to set *all* of these parameters for your application.

Table 9–3 Parameters Which Control PCM Locks

Parameter	Description	Value
GC_FILES_TO_LOCKS	Gives the mapping of hashed and releasable locks to blocks within each datafile. The meaning of this parameter has changed. Previously, files not mentioned in this parameter (or files added later) were assigned the remaining hashed locks. Files not mentioned in this parameter use releasable locks. You can now have multiple entries of GC_FILES_TO_LOCKS.	The configuration string for GC_FILES_TO_LOCKS now includes a value of zero for the number of locks. This indicates that the blocks are protected by fine grain locks. Instances must have identical values.
GC_LCK_PROCS	Sets the number of background lock processes (LCK0 through LCK9) for an instance in a parallel server.	In shared mode, the value of this parameter must be greater than 0; the default value is 1. In exclusive mode, this parameter is ignored. Instances must have identical values.

Table 9–3 Parameters Which Control PCM Locks

Parameter	Description	Value
GC_LATCHES	Specifies the number of lock element latches each LCK process has. This parameter should only be set if there is lock element latch contention.	Defaults to (#CPUs * 2). On a uniprocessor, therefore, each LCK process would have 2 latches.
GC_RELEASABLE_LOCKS	Sets the number of locks which will be used for DBA locks.	Defaults to the value of DB_BLOCK_BUFFERS. Normally this value is optimal, and you should not change it. Note: In versions prior to Oracle8, setting this parameter to a value less than DB_BLOCK_BUFFERS was ineffective: the value was automatically returned to this default. In Oracle8, lower settings are valid. If you have migrated from an earlier version, check the setting of this parameter to avoid performance impact.
GC_ROLLBACK_LOCKS	For each rollback segment, specifies the number of instance locks available for simultaneously modified rollback segment blocks.	The default value is to use releasable locks for each rollback segment.

See Also: *Oracle8 Reference* for complete specifications for these parameters. Chapter 15, "Allocating PCM Instance Locks", provides information on how to set these parameters.

Handling Data Blocks

Do *not* allocate PCM locks for files which *only* contain the following, because class 1 blocks are not used for these files:

- temporary tables for internal sorts. (These are class 2 blocks.)
- rollback segments. These are protected by GC_ROLLBACK_LOCKS.

Two Methods of PCM Locking: Fixed and Releasable

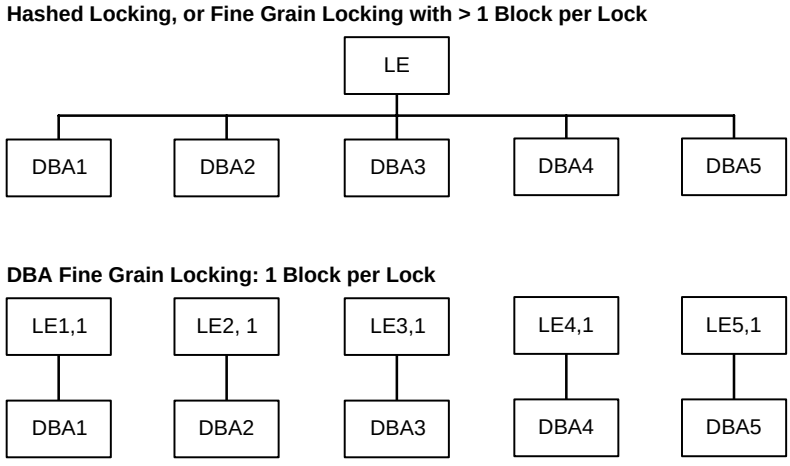
This section compares the two methods for PCM locking: fixed and releasable locking. You can use either or both kinds of PCM locks to protect the blocks in datafiles.

- Integrated DLM Lock Elements and PCM Locks
- Number of Blocks per PCM Lock
- Fine Grain Locking: Locks for One or More Blocks
- How Fine Grain Locking Works
- Performance Effects of Releasable Locking
- Applying Fine Grain and Hashed Locking to Different Files

Integrated DLM Lock Elements and PCM Locks

Figure 9–8 illustrates the correspondence of lock elements to blocks in hashed and fine grain locking. A lock element (LE) is an Oracle-specific data structure that represents an IDLM lock. There is a one-to-one correspondence between a lock element and a PCM lock in the IDLM.

Figure 9–8 Hashed Locking and DBA Fine Grain Locking



Lock Elements for Hashed PCM Locks

For both hashed PCM locks and fine grain locks, you can specify more than 1 block per lock element. The difference is that by default hashed PCM locks are not releasable; the lock element name is fixed.

When the lock element is pinged, any other modified blocks owned by that lock element will be written along with the needed one. For example, in Figure 9–8, if LE is pinged when block DBA2 is needed, blocks DBA1, DBA3, DBA4, and DBA5 will all be written to disk as well—if they have been modified.

Lock Elements for Fine Grain PCM Locks

In fine grain locking, the name of the lock element is the name of the resource inside the IDLM.

Although a fixed number of lock elements cover potentially millions of blocks, the lock element names change over and over again as they are associated with specific blocks that are requested. The lock element name (for example, LE7,1) contains the database block address (7) and class (1) of the block it covers. Before a lock element can be reused, the IDLM lock must be released. You can then rename and reuse the lock element, creating a new resource in the IDLM if necessary.

When using fine grain locking, you can set your system with many more potential lock names, since they do not need to be held concurrently. However, the number of blocks mapped to each lock is configurable in the same way as hashed locking.

Lock Elements for DBA Fine Grain PCM Locks

In fine grain locking you can set a one-to-one relationship between lock elements and blocks. Such an arrangement, illustrated in Figure 9–8, is called *DBA locking*. Thus if LE2,1 is pinged, only block DBA2 is written to disk.

Number of Blocks per PCM Lock

This section explains the ways in which hashed locks and fine grain locks can differ in lock granularity.

Hashed Locks for Multiple Blocks

Hashed PCM locks can protect more than one Oracle database block. The mapping of PCM locks to blocks in the database is determined on a file-by-file basis using initialization parameters specified when the first Oracle Parallel Server instance is started. The parameters can specify that the PCM lock protects a range of contiguous blocks within the file.

Hashed locks are useful in the following situations:

Table 9–4 When to Use Hashed PCM Locks

Situation	Reason
When the data is mostly read-only	A few hashed locks can cover many blocks without requiring frequent lock operations. These locks are released only when another instance needs to modify the data. Hashed locking can perform up to 100% faster than fine grain locking on read-only data with the Parallel Query Option. **Note:** If the data is strictly read-only, consider designating the tablespace itself as read-only. The tablespace will not then require any PCM locks.
When the data can be partitioned according to the instance which is likely to modify it	Hashed locks which are defined to match this partitioning allow instances to hold disjoint IDLM lock sets, reducing the need for IDLM operations.
When a large set of data is modified by a relatively small set of instances	Hashed locks permit access to a new database block to proceed without IDLM activity, if the lock is already held by the requesting instance.

Hashed locks may cause extra cross-instance lock activity, since conflicts may occur between instances which are modifying different database blocks. The resolution of this false conflict (“false pinging”) may require writing several blocks from the cache of the instance which currently owns the lock.

Fine Grain Locking: Locks for One or More Blocks

A fine grain lock can protect one or more Oracle database blocks. If you create a one-to-one correspondence between PCM locks and datablocks, then contention will occur only when instances need data from the same block. This level of fine grain locking is known as DBA locking. (A *DBA* is the data block address of a single block of data.) If you assign more than one block per lock, then contention will occur as in hashed locking.

On most systems an instance could not possibly hold a lock for each block of a database since SGA memory or IDLM locking capabilities would be exceeded. Therefore, instances acquire and release fine grain locks as required. Since fine grain locks, lock elements, and resources are renamed in the IDLM and reused, a system can employ fewer of them. The value of DB_BLOCK_BUFFERS is the recommended minimum number of releasable locks you should allocate.

DBA fine grain locks are useful when a database object is updated frequently by several instances. This advantage is gained as follows:

- Conflicts occur only when the same block is needed by the two instances.
- Only the required block is written to disk by the instance currently owning the PCM lock *in exclusive mode*.

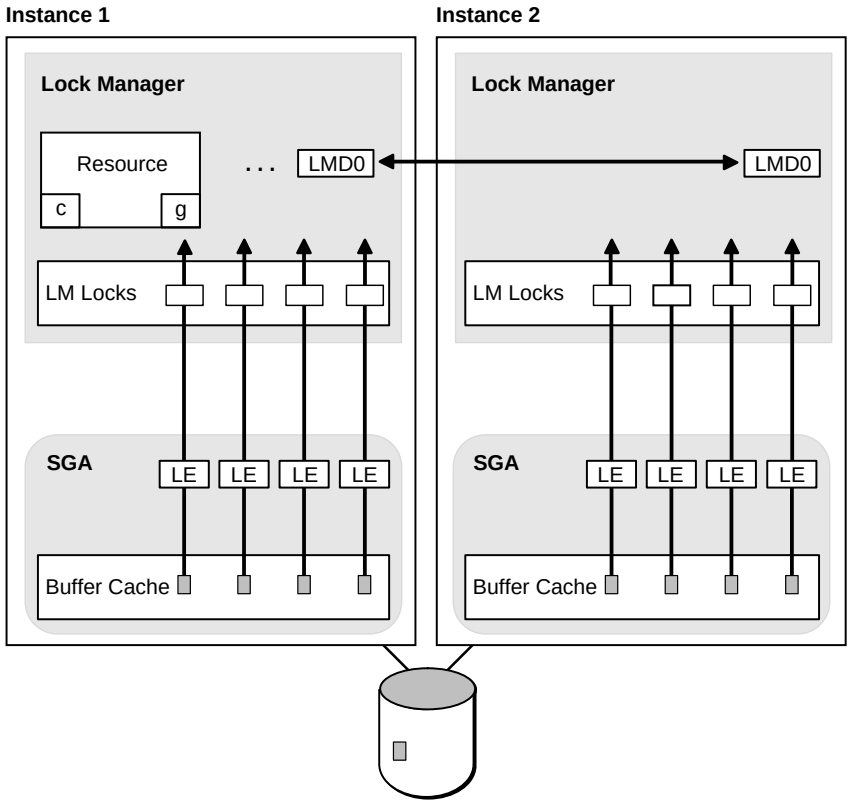
A disadvantage of fine grain locking is that overhead is incurred for each block read, and performance will be affected accordingly. (Acquiring a new lock and releasing it each time causes more overhead through the IDLM than converting the mode of an existing lock from null mode to exclusive mode and back, as is done in hashed locking.)

See Also: "Releasable Lock Example" on page 15-10.

How Fine Grain Locking Works

Figure 9–9 shows how fine grain locking operates.

Figure 9–9 Lock Elements Coordinate Blocks (by Fine Grain Locking)



The foreground process checks in the SGA to see if the instance owns a lock on the block.

- If it does not, then LCK will obtain the lock. To do this, LCK needs a free lock element. If no free lock element exists, LCK will create one by releasing some other lock. When this is done, LCK will obtain the new lock.
- If the instance does own the lock, but in the wrong mode, then LCK will convert the lock (for example, from shared to exclusive mode).

A lock element is created in either of two modes: fixed or releasable

- Fixed locks (whether hashed or fine grain) create a lock element in fixed mode, which is always valid. This mode is static; lock elements stay the same, once allocated.
- Releasable locks (whether fine grain or hashed) create a lock element in non-fixed mode; these lock element names can change, and the block or blocks covered can change. Lock elements in non-fixed mode can be valid, old, or free. If the valid bit is set then a lock is owned on the resource in the IDLM. If not set, there is no lock. If it is free, then there is a lock but we have unlinked the buffer from the lock element, so it is on the least recently used list of free lock elements.

  Note: Valid lock elements have a lock in the IDLM; invalid lock elements do not. A free lock element indicates that a lock exists in the IDLM which is not currently linked to this buffer; it is waiting on the LRU list. If a lock element is old, then there is a valid lock handle for the old name. It must be given a new name before Oracle can use it.

The V$LOCK_ELEMENT view shows the status of the lock elements.

Performance Effects of Releasable Locking

Releasable locking may affect performance of Oracle Parallel Server. Since releasable locks are more expensive (since they may cause a release lock and get lock on a buffer get), some operations may show a decreased level of performance when run in this mode. However, other types of access to the database will improve with releasable fine grain locks. Fine grain locking may have the following results:

- Read-only scans of tables may require more lock operations. If this happens, you can use hashed locking.
- There may be a reduction of false conflicts on high-contention blocks or objects. If this happens, fine grain locking is a good choice.
- The system has more expensive lock operations and a lower false conflict rate for low concurrency data blocks. If this happens you must examine your priorities and decide whether this is a reasonable trade-off for your application.
- Grouping with fine grain locks might be good for table scans.

Applying Fine Grain and Hashed Locking to Different Files

Each datafile can use one or the other method of locking. For best results, you may need to use hashed locks on some datafiles, and fine grain locking on other datafiles.

You can selectively apply hashed and fine grain locking on different files. For example, you could apply locks as follows on a set of files:

GC_FILES_TO_LOCKS = ”1=100:2=0:3=1000:4-5=0EACH”
GC_RELEASABLE_LOCKS=10000

Table 9–5 Selective Application of Hashed and Fine Grain Locking

File Number	Locking Mode	Value in GC_FILES_TO_LOCKS
1	Hashed	100
2	Fine grain	0
3	Hashed	1000
4	Fine grain	0
5	Fine grain	0

How Locks Are Assigned to Blocks

This section explains how hashed locks and fine grain locks are assigned to blocks. (DBA locks, of course, have a one-to-one correspondence to blocks.)

- File to Lock Mapping
- Number of Locks per Block Class
- Lock Element Number

File to Lock Mapping

Two data structures in the SGA control file to lock mapping. The first structure maps each file (DB_FILES) to a bucket (index) in the second structure. This structure contains information on the number of locks allocated to this bucket, base lock number and grouping factor. To find the number of locks for a tablespace, you must count up the number of actual fixed locks which protect the different files. If files share locks, you count the shared locks only once.

1. To find the number of locks for a tablespace, begin by performing a select from the FILE_LOCK data dictionary table:

   ```
   SELECT * FROM FILE_LOCK:
   ```

   For example, you would get results like the following if you had set GC_FILES_TO_LOCKS="1=500:5=200":

   ```
   FILE_ID  FILE_NAME  TABLESPACE_NAME  START_LK  NLOCKS BLOCKING
   1        file1      system             1         500     1
   1        file2      system             0
   1        file3      system             0
   1        file4      system             0
   1        file5      system           501         200     1
   ```

2. Count the number of locks in the tablespace by summing the number of locks (value of the NLOCKS column) *only for rows with different values in the START_LCK column*.

   In this example, both file1 and file5 have different values for START_LCK. You therefore sum up their NLOCKS values for a total of 700 locks.

If, however, you had set GC_FILES_TO_LOCKS="1-2=500:5=200", your results would look like the following:

```
FILE_ID  FILE_NAME  TABLESPACE_NAME  START_LK  NLOCKS BLOCKING
1        file1      system             1         500    1
1        file2      system             1         500    1
1        file3      system             0
1        file4      system             0
1        file5      system           501         200    1
```

This time, file1 and file 2 have the same value for START_LCK; this indicates that they share the locks in question. File5 has a different value for START_LCK. You therefore count once the 500 locks shared by files 1 and 2, and add an additional 200 locks for file 5, for a total of 700.

Number of Locks per Block Class

You need only concern yourself with the number of blocks in the data and undo block classes. Data blocks (class 1) contain data from indexes or tables. System undo header blocks (class 10) are also known as the rollback segment headers or transaction tables. System undo blocks (class 11) are part of the rollback segment and provide storage for undo records.

User undo segment *n* header blocks are identified as class 10 + (*n**2), where *n* represents the rollback segment number. A value of *n* = 0 indicates a system rollback segment; a value of *n* > 0 indicates a non-system rollback segment. Similarly, user undo segment *n* header blocks are identified as class 10 + ((n*2) + 1).

The following query shows the number of locks allocated per class:

```
SELECT CLASS, COUNT(*)
  FROM V$LOCK_ELEMENT
 GROUP BY CLASS
 ORDER BY CLASS;
```

The following query shows the number of fixed (non-releasable) PCM locks:

```
SELECT COUNT(*)
  FROM V$LOCK_ELEMENT
 WHERE bitand(flag, 4)!=0;
```

The following query shows the number of fine grain PCM locks:

```
SELECT COUNT(*)
  FROM V$LOCK_ELEMENT
 WHERE bitand(flag, 4)=0;
```

Lock Element Number

For a data class block the file number is determined from the data block address (DBA). The bucket is found through the X$KCLFI dynamic performance table. Data class blocks are hashed to lock element numbers as follows:

$$\left(\frac{DBA}{grouping\_factor} \right) \text{ modulo } (locks) + (start)$$

Other block classes are hashed to lock element numbers as follows

(*DBA*) modulo (*locks_in_class*)

Examples: Mapping Blocks to PCM Locks

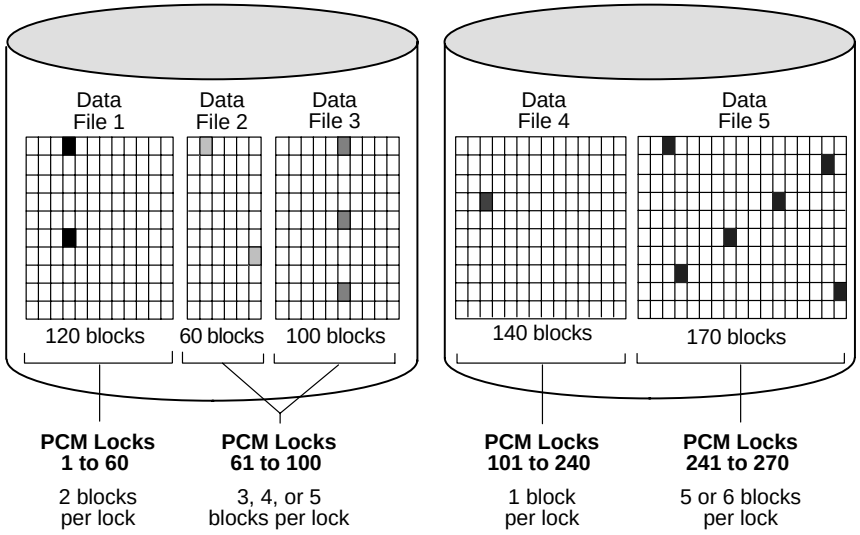
- Setting GC_FILES_ TO_LOCKS
- More Sample Hashed Settings of GC_FILES_TO_LOCKS
- Sample Fine Grain Setting of GC_FILES_TO_LOCKS

Setting GC_FILES_ TO_LOCKS

The following examples show different ways of mapping blocks to PCM locks, and how the same locks are used on multiple datafiles.

Note: These examples discuss very small sample files to illustrate important concepts. The actual files you manage will be significantly larger.

Figure 9–10 Mapping PCM Locks to Data Blocks



Example 1 Figure 9–10 shows an example of mapping blocks to PCM locks for the parameter value GC_FILES_TO_LOCKS = ”1=60:2-3=40:4=140:5=30”.

In datafile 1 of the figure, 60 PCM locks map to 120 blocks, which is a multiple of 60. Each PCM lock therefore covers two data blocks.

In datafiles 2 and 3, 40 PCM locks map to a total of 160 blocks. A PCM lock can cover either one or two data blocks in datafile 2, and two or three data blocks in

datafile 3. Thus, one PCM lock may cover three, four, or five data blocks across both datafiles.

In datafile 4, each PCM lock maps exactly to a single data block, since there is the same number of PCM locks as data blocks.

In datafile 5, 30 PCM locks map to 170 blocks, which is not a multiple of 30. Each PCM lock therefore covers five or six data blocks.

Each of the PCM locks illustrated in Figure 9–10 can be held in either read-lock mode or read-exclusive mode.

Example 2 The following parameter value allocates 500 PCM locks to datafile 1; 400 PCM locks each to files 2, 3, 4, 10, 11, and 12; 150 PCM locks to file 5; 250 PCM locks to file 6; and 300 PCM locks collectively to files 7 through 9:

```
GC_FILES_TO_LOCKS = "1=500:2-4,10-12=400EACH:5=150:6=250:7-9=300"
```

This example assigns a total of (500 + (6*400) + 150 + 250 + 300) = 3600 PCM locks. You may specify more than this number of PCM locks if you intend to add more datafiles later.

Example 3 In Example 2, 300 PCM locks are allocated to datafiles 7, 8, and 9 collectively with the clause "7-9=300". The keyword EACH is omitted. If each of these datafiles contains 900 data blocks, for a total of 2700 data blocks, then each PCM lock covers 9 data blocks. Because the datafiles are multiples of 300, the 9 data blocks covered by the PCM lock are spread across the 3 datafiles; that is, one PCM lock covers 3 data blocks in each datafile.

Example 4 The following parameter value allocates 200 PCM locks each to files 1 through 3; 50 PCM locks to datafile 4; 100 PCM locks collectively to datafiles 5, 6, 7, and 9; and 20 data locks in contiguous 50-block groups to datafiles 8 and 10 combined:

```
GC_FILES_TO_LOCKS = "1-3=200EACH 4=50:5-7,9=100:8,10=20!50"
```

In this example, a PCM lock assigned to the combined datafiles 5, 6, 7, and 9 covers one or more data blocks in each datafile, unless a datafile contains fewer than 100 data blocks. If datafiles 5 to 7 contain 500 data blocks each and datafile 9 contains 100 data blocks, then each PCM lock covers 16 data blocks: one in datafile 9 and five each in the other datafiles. Alternatively, if datafile 9 contained 50 data blocks, half of the PCM locks would cover 16 data blocks (one in datafile 9); the other half of the PCM locks would only cover 15 data blocks (none in datafile 9).

The 20 PCM locks assigned collectively to datafiles 8 and 10 cover contiguous groups of 50 data blocks. If the datafiles contain multiples of 50 data blocks and the total number of data blocks is not greater than 20 times 50 (that is, 1000), then each PCM lock covers data blocks in either datafile 8 or datafile 10, but not in both. This is because each of these PCM locks covers 50 contiguous data blocks. If the size of datafile 8 is not a multiple of 50 data blocks, then one PCM lock must cover data blocks in both files. If the sizes of datafiles 8 and 10 exceed 1000 data blocks, then some PCM locks must cover more than one group of 50 data blocks, and the groups might be in different files.

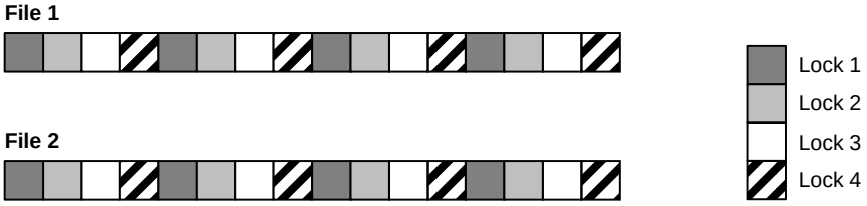
More Sample Hashed Settings of GC_FILES_TO_LOCKS

Examples 5, 6, and 7 show the results of specifying various values of GC_FILES_TO_LOCKS. In the examples, files 1 and 2 each have 16 blocks of data.

Example 5 GC_FILES_TO_LOCKS="1-2=4"

In this example four locks are specified for files 1 and 2. Therefore, the number of blocks covered by each lock is 8 ((16+16)/4). The blocks are not contiguous.

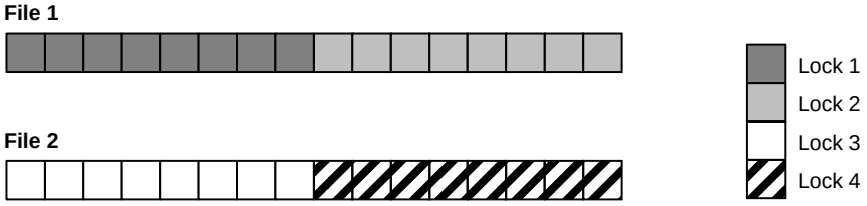
Figure 9–11 GC_FILES_TO_LOCKS Example 5



Example 6 GC_FILES_TO_LOCKS=”1-2=4!8”

In this example four locks are specified for files 1 and 2. However, the locks must cover 8 contiguous blocks.

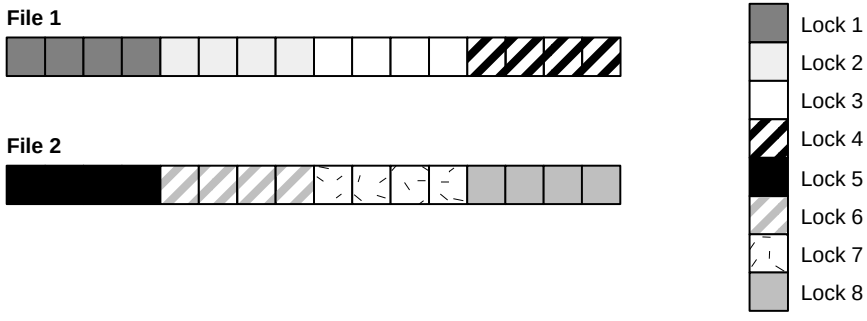
Figure 9–12 GC_FILES_TO_LOCKS Example 6



Example 7 GC_FILES_TO_LOCKS=”1-2=4!4EACH”

In this example four locks are specified for file 1 and four for file 2. The locks must cover 4 contiguous blocks.

Figure 9–13 GC_FILES_TO_LOCKS Example 7



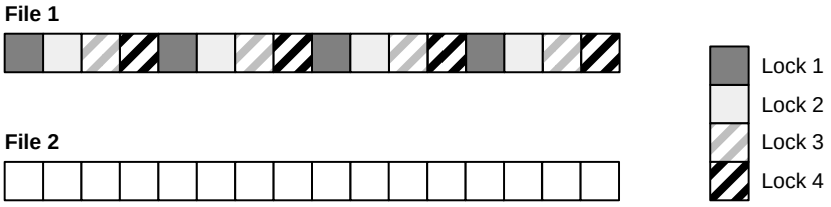
Sample Fine Grain Setting of GC_FILES_TO_LOCKS

The following example shows fine grain locking mixed with hashed locking.

Example 8 GC_FILES_TO_LOCKS="1=4:2=0"

File 1 has hashed PCM locking with 4 locks. On file 2, fine grain locks are allocated on demand—none are initially allocated.

Figure 9–14 GC_FILES_TO_LOCKS Example 8



10

Non-PCM Instance Locks

This chapter describes some of the most common non-PCM instance locks. It covers the following information:

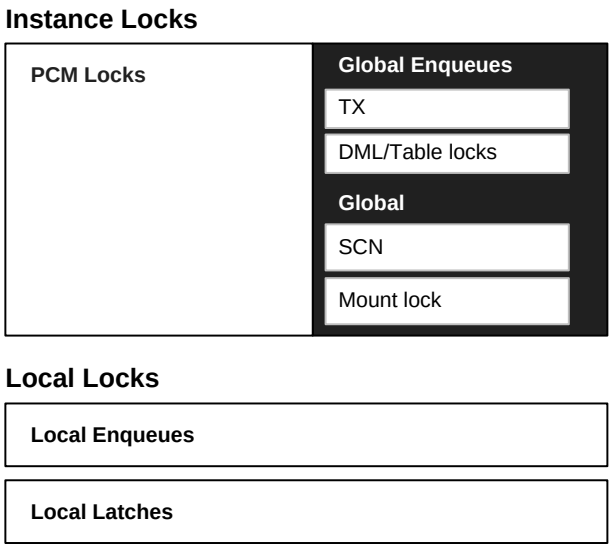
- Overview
- Transaction Locks (TX)
- Table Locks (TM)
- System Change Number (SC)
- Library Cache Locks (N[A-Z])
- Dictionary Cache Locks (Q[A-Z])
- Database Mount Lock (DM)

See Also: Chapter 16, “Ensuring IDLM Capacity for All Resources & Locks”, for details on how to calculate the number of non-PCM resources and locks to configure in the Integrated DLM.

Overview

Figure 10–1 highlights non-PCM locks in relation to other locks used in Oracle.

Figure 10–1 Oracle Locking Mechanisms: Non-PCM Locks



Whereas PCM locks are static (you allocate them when you design your application), non-PCM locks are very dynamic. Their number and corresponding space requirements will change as your system's initialization parameter values change.

See Also: *Oracle8 Reference* for descriptions of all non-PCM locks.

Transaction Locks (TX)

Row locks are locks that protect selected rows. A transaction acquires a global enqueue and an exclusive lock for each individual row modified by one of the following statements:

- INSERT
- UPDATE
- DELETE
- SELECT with the FOR UPDATE clause

These locks are stored in the block, and each lock refers back to the global transaction enqueue.

A transaction lock is acquired in exclusive mode when a transaction initiates its first change. It is held until the transaction does a COMMIT or ROLLBACK. It is also acquired exclusive by SMON when doing recovery (undo) of a transaction. Transaction locks are used as a queuing mechanism for processes which are awaiting the release of an object that is locked by a transaction in progress.

Table Locks (TM)

Table locks are DML locks that protect entire tables. A transaction acquires a table lock when a table is modified by one of the following statements: INSERT, UPDATE, DELETE, SELECT with the FOR UPDATE clause, and LOCK TABLE. A table lock can be held in any of several modes: null (N), row share (RS), row exclusive (RX), share lock (S), share row exclusive (SRX), and exclusive (X).

When an instance attempts to mount the database, a table lock is used to ensure that all participating instances either have DML_LOCKS = 0 or DML_LOCKS != 0. If they do not, than error ORA-61 is returned and the mount attempt fails. Table locks are acquired during the execution of a transaction when referencing a table with a DML statement so that the object is not dropped or altered during the execution of the transaction. This occurs if and only if the DML_LOCKS parameter is non-zero.

You can also selectively turn table locks on or off for a particular table, using the statement

```
ALTER TABLE tablename DISABLE|ENABLE TABLE LOCK
```

Note that if DML_LOCKS is set to zero, then no DDL operations are allowed. The same is true for tables which have disabled table locks.

See Also: "Minimizing Table Locks to Optimize Performance" on page 16-8 to consider disabling table locks for improved performance.

System Change Number (SC)

The System Change Number (SCN) is a logical time stamp Oracle uses to order events within a single instance, and across all instances. One of the schemes Oracle uses to generate SCNs is the lock scheme.

The lock SCN scheme keeps the global SCN in the value block of the SCN lock. This value is incremented in response to many database events, most notably COMMIT WORK. A process incrementing the global SCN will get the SCN lock in exclusive mode, increment the SCN, write the lock value block, and downgrade the lock. Access to the SCN lock value is batched. Oracle keeps a cache copy of the global SCN in memory. A process may get an SCN without any communication overhead by reading the SCN fetched by other processes.

The SCN implementation can differ from platform to platform. On most platforms, Oracle uses the lock SCN scheme when the MAX_COMMIT_PROPAGATION_DELAY initialization parameter is smaller than a platform-specific threshold (typically 7). Oracle uses the Lamport SCN scheme when MAX_COMMIT_PROPAGATION_DELAY is larger than the threshold.You can examine the alert log after an instance is started to see which SCN generation scheme has been picked.

See Also: Your Oracle system-specific documentation for information about the SCN implementation.

Library Cache Locks (N[A-Z])

When a database object (table, view, procedure, function, package, package body, trigger, index, cluster, synonym) is referenced during parsing or compiling of a SQL (DML/DDL) or PL/SQL statement, the process parsing or compiling the statement acquires the library cache lock in the correct mode. In Oracle8 the lock is held only until the parse or compilation completes (for the duration of the parse call).

Dictionary Cache Locks (Q[A-Z])

The data dictionary cache contains information from the data dictionary, the meta-data store. This cache provides efficient access to the data dictionary.

Creating a new table, for example, causes the meta-data of that table to be cached in the data dictionary. If a table is dropped, the meta-data needs to be removed from the data dictionary cache. To synchronize access to the data dictionary cache, latches are used in exclusive mode and in single shared mode. Instance locks are used in multiple shared (parallel) mode.

In the case of parallel server, the data dictionary cache on all nodes may contain the meta-data of a table that gets dropped on one instance. The meta-data for this table needs to be flushed from the data dictionary cache of every instance. This is performed and synchronized by instance locks.

Database Mount Lock (DM)

The mount lock shows whether or not any instance has mounted a particular database. This lock is only used with Oracle Parallel Server. It is the only multi-instance lock used by OPS in exclusive mode, where it prevents another instance from mounting the database in shared mode.

In Oracle Parallel Server single shared mode, this lock is held in shared mode. Another instance can successfully mount the same database in shared mode. In OPS exclusive mode, however, another instance will not able to get the lock.

11

Space Management and Free List Groups

Thus would I double my life's fading space;
For he that runs it well, runs twice his race.

Abraham Cowley, *Discourse xi, Of Myself*

This chapter explains space management concepts:

- How Oracle Handles Free Space
- SQL Options for Managing Free Space
- Managing Free Space on Multiple Instances
- Free Lists Associated with Instances, Users, and Locks
- Controlling the Allocation of Extents

See Also: Chapter 17, "Using Free List Groups to Partition Data", for a description of space management procedures.

How Oracle Handles Free Space

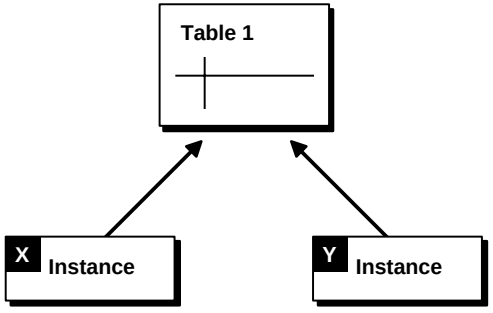
This section provides an overview of how Oracle handles free space. It contains the following sections:

- Overview
- Database Storage Structures
- Structures for Managing Free Space
- Example: Free List Groups

Overview

Oracle Parallel Server enables transactions running on separate instances to insert and update data in the same table concurrently, without contention to locate free space for new records.

Figure 11–1 Instances Concurrently Inserting to a Table



To take advantage of this capability, you must actively manage free space in your database using several structures which are defined in this chapter.

For each database object (a table, cluster, or index), Oracle keeps track of blocks with space available for inserts (or for updates which may cause rows to exceed space available in their original block). A user process that needs free space can look in the master free list of blocks that contain free space. If the master free list does not contain a block with enough space to accommodate the user process, Oracle allocates a new extent.

New extents that are automatically allocated to a table add their blocks to the master free list. This can eventually result in contention for free space among multiple instances on a parallel server because the free space contained in automatically allo-

cated extents cannot be reallocated to any group of free lists. You can have more control over free space if you specifically allocate extents to instances; in this way you can minimize contention for free space.

Database Storage Structures

This section describes basic structures of database storage:

- Segments and Extents
- High Water Mark

Segments and Extents

A *segment* is a unit of logical database storage. Oracle allocates space for segments in smaller units called *extents*. An extent is a specific number of contiguous data blocks allocated for storing a specific type of information.

A segment thus comprises a set of extents allocated for a specific type of data structure. For example, each table's data is stored in its own data segment, while each index's data is stored in its own index segment.

The extents of a segment are all stored in the same tablespace; they may or may not be contiguous on disk. The segments can span files, but individual extents cannot.

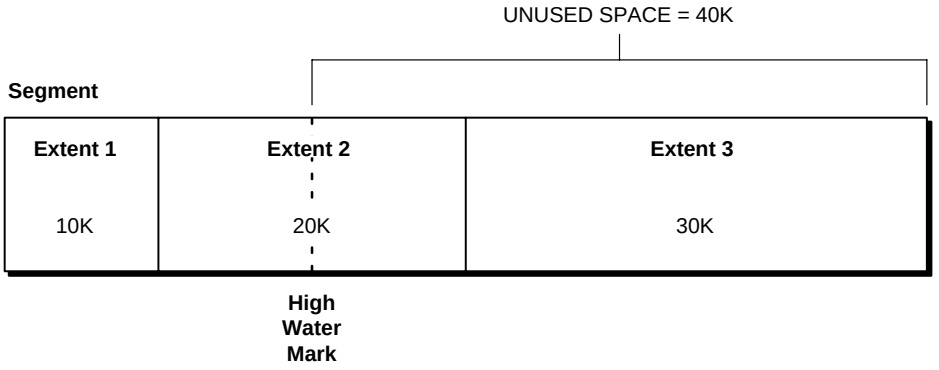
Note that although you can allocate additional extents, the blocks themselves are allocated separately. If you allocate an extent to a specific instance, the blocks are immediately allocated to the free list. However, if the extent is not allocated to a specific instance, then the blocks themselves are allocated only when the high water mark moves.

High Water Mark

The *high water mark* is the boundary between used and unused space in a segment. As requests are received for new free blocks (which cannot be satisfied by existing free lists), the block to which the high water mark points becomes a used block, and the high water mark is advanced to the next block. In other words, the segment space to the left of the high water mark is used, and the space to the right of it is unused.

Figure 11–2 shows a segment which consists of three extents containing 10K, 20K, and 30K of space, respectively. The high water mark is in the middle of the second extent, thus the segment contains 20K of used space (to the left of the high water mark), and 40K of unused space (to the right of the high water mark.

Figure 11–2 High Water Mark



See Also: *Oracle8 Concepts* for further information about segments and extents.

Structures for Managing Free Space

Oracle uses the following structures to manage free space:

- Transaction Free Lists
- Process Free Lists
- Free List Groups
- The Master Free List

Process free lists are used to relieve contention for free space among processes inside the instance, even if multiple instances can hash to a single free list group. Free list groups are used to relieve forced reads / writes between instances. Process free lists and free list groups are supported on *all* database objects alike: tables, indexes, and clusters.

Transaction Free Lists

A *transaction free list* is a list of blocks made free by uncommitted transactions. They exist by default.

When transactions are committed, the freed blocks eventually go to the master free list (described below).

Process Free Lists

A *process free list* (also termed simply a "free list" in this documentation) is a list of free data blocks that can be drawn from a number of different extents within the segment.

Blocks in free lists contain free space greater than PCTFREE (the percentage of a block to be reserved for updates to existing rows). In general, blocks included in process free lists for a database object must satisfy the PCTFREE and PCTUSED constraints described in the chapter "Data Blocks, Extents, and Segments" in *Oracle8 Concepts*.

Process free lists must be specifically enabled by the user. You can specify the number of process free lists desired by setting the FREELISTS parameter when you create a table, index or cluster. The maximum value of the FREELISTS parameter depends on the Oracle block size on your system. In addition, for each free list, you need to store a certain number of bytes in a block to handle overhead.

Note: The reserved area and the number of bytes required per free list depend upon your platform. For more information, see your Oracle system-specific documentation.

Free List Groups

A *free list group* is a set of free lists you can specify for use by one or more particular instances. Each free list group provides free data blocks to accommodate inserts or updates on tables and clusters, and is associated with instance(s) at startup.

A parallel server has multiple instances, and process free lists alone cannot solve the problem of contention. Free list groups, however, effectively reduce pinging between instances.

When enabled, free list groups divide the set of free lists into subsets. Descriptions of process free lists are stored in separate blocks for the different free list groups. Each free list group block points to the same free lists, except that every instance gets its own. (Or, in the case of more instances than free list groups, multiple instances hash into the same free list group.) This ensures that the instances do not compete for the same blocks

Attention: In Oracle Parallel Server, you should always use free list groups, along with process free lists.

The Master Free List

The master free list is a repository of blocks which contain available space, drawn from any extent in the table. It exists by default, and includes:

- blocks which were made free by a committed transaction. These go on the master free list when there is a need for free blocks.
- subsequent space allocations not specifically associated with any free list group. When the high water mark moves, then blocks go on the master free list.

If free list groups exist, each group has its own master free list. There is, in addition, a central master free list which is mostly used for parallel operations.

Avoiding Contention for the Segment Header and Master Free LIst

A highly concurrent environment has potential contention for the segment header, which contains the master free list.

- *If free list groups exist,* then the segment header only points to the central master free list. In addition, every free list group block contains pointers to its own master free list, transaction free lists, and process free lists.
- I*f free list groups do not exist,* then the segment header contains pointers to the master free list, transaction free lists, and process free lists.

In a single instance environment, multiple process free lists help to solve the problem of many users seeking free data blocks by easing contention on segment header blocks.

In a multi-instance environment, as illustrated in Figure 11–3, process free lists provide free data blocks from available extents to different instances. You can partition multiple free lists so that extents are allocated to specific database instances. Each instance hashes to one or more free list groups, and each group's header block points to process free lists.

If no free list groups are allocated, however, the segment header block of a file points to the process free lists. Without free list groups, every instance must read the segment header block in order to access the free lists.

Figure 11–3 Contention for the Segment Header

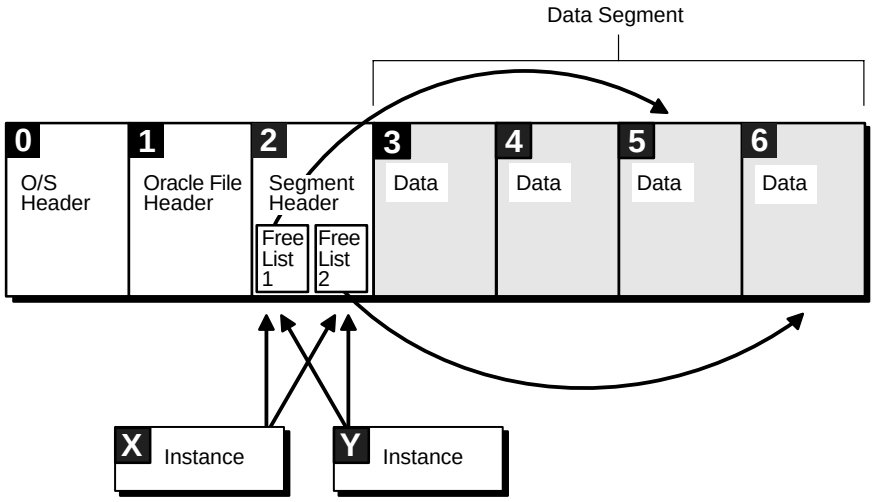
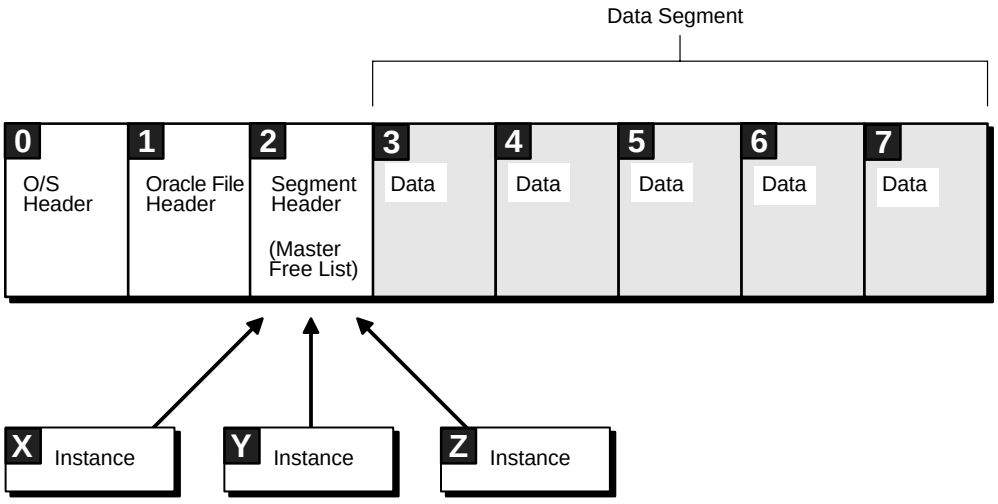


Figure 11–4 shows the blocks of a file in which the master free list is stored in the segment header block. Three instances are forced to read this block in their effort to obtain free space. Because there is only one free list, there is only one insertion point. Process free lists can help to reduce this contention by spreading this insertion point over multiple blocks, each of which will be accessed less often.

Figure 11–4 Contention for Master Free List



Example: Free List Groups

A Simple Case

Figure 11–5 illustrates the division of free space for a table into a master free list and two free list groups, each of which contains three free lists. This example concerns a well-partitioned application in which deletes occur. The master free list pictured is the master free list for this particular free list group.

The table was created with one initial extent, after which extents 2 and 5 were allocated to instance X, extents 3 and 4 were allocated to instance Y, and extent 6 was allocated automatically (not to a particular instance). Notice the following:

- The dark shaded blocks in the initial allocation and extent 6 represent the master free list of free blocks.
- The light gray blocks represent available free space in free list group X.
- The medium gray blocks represent the available free space in free list group Y.
- Extent 5 is newly allocated, thus all of its blocks are in free list group X.

- Solid black blocks represent space freed by deletions, which returns to free list groups X and Y.
- Unshaded blocks do not contain enough free space for inserts.

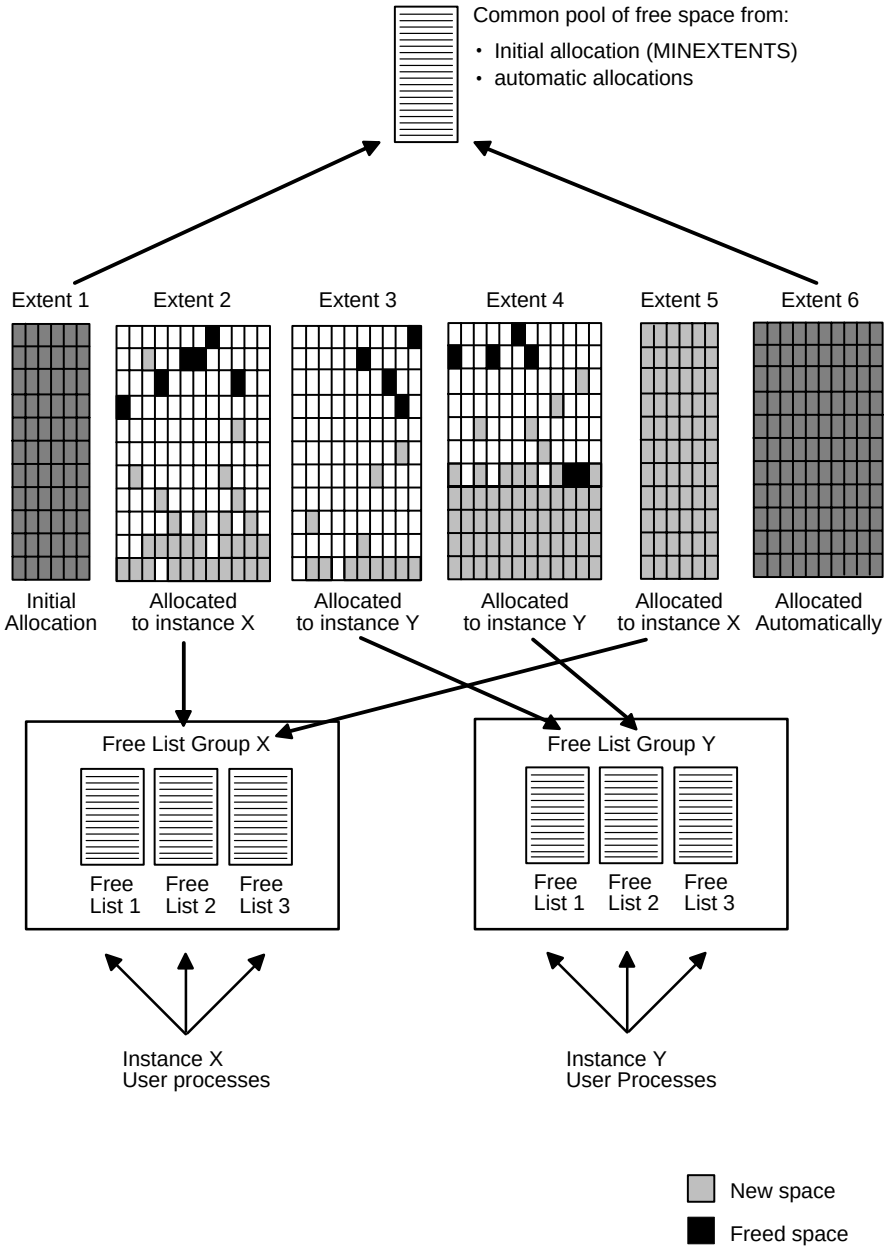
Each user process running on instance X uses one of the free lists in group X, and each user process on instance Y uses one of the free lists in group Y. If more instances start up, their user processes share free lists with instance X or Y.

A More Complicated Case

The simple case in Figure 11–5 becomes more complicated when you consider that extents are not allocated to instances permanently, and that space allocated to one instance cannot be used by another instance. Each free list group has its own master free list. After allocation, some blocks go onto the master free list for the group, some to a process free list, and some do not belong to a free list. If the application is totally partitioned, then once blocks are allocated to a given instance, they stay with that instance. However, blocks can move from one instance to another if the application is not totally partitioned.

Consider a situation in which instance Y fills a block, takes it off the free list, and then instance X frees the block. The block then goes to the free list of instance X, the instance which freed it. If instance Y needs space, it cannot reclaim this block. Instance Y can only obtain free space from its own free list group.

Figure 11–5 Groups of Free Lists for a Table



SQL Options for Managing Free Space

Several SQL options enable you to allocate process free lists and free list groups for tables, clusters, and indexes. You can explicitly specify that new space for an object be taken from a specific datafile. You can also associate free space with particular free list groups, which can then be associated with particular instances.

The SQL statements include:

CREATE TABLE [CLUSTER, INDEX]

STORAGE

FREELISTS

FREELIST GROUPS

ALTER TABLE [CLUSTER, INDEX]

ALLOCATE EXTENT

SIZE

DATAFILE

INSTANCE

You can use these SQL options with the initialization parameter INSTANCE_NUMBER to associate data blocks with instances.

See Also: *Oracle8 SQL Reference* for complete syntax of these statements.

Managing Free Space on Multiple Instances

This section describes:

- Partitioning Free Space into Multiple Free Lists
- Partitioning Data with Free List Groups
- How Free Lists and Free List Groups Are Assigned to Instances

Partitioning Free Space into Multiple Free Lists

You can partition free space for individual tables, clusters (other than hash clusters), and indexes into multiple process free lists. Multiple free lists allow a process to search a specific pool of blocks when space is needed, thus reducing contention among users for free space. Within an instance, using free lists can reduce contention if multiple processes are inserting into the same table.

Each table has a master free list of blocks with available space, and can also contain multiple free lists. Before looking in the master free list, a user process scans the appropriate free list to locate a block that contains enough space.

Partitioning Data with Free List Groups

The separation of free space into groups can improve performance by reducing contention for free data blocks during concurrent inserting by multiple instances on a parallel server. You can thus create groups of process free lists for a parallel server, each of which can contain multiple free lists for a table, index, or cluster. You can use free list groups to partition data by allocating extents to particular instances.

In general, all tables should have the same number of free list groups, but the number of free lists within a group may vary, depending on the type and amount of activity of each table.

Partitioning free space can particularly improve the performance of applications that have a high volume of concurrent inserts, or updates requiring new space, from multiple instances. Performance improvements also depend, of course, on your operating system, hardware, data block size, and so on.

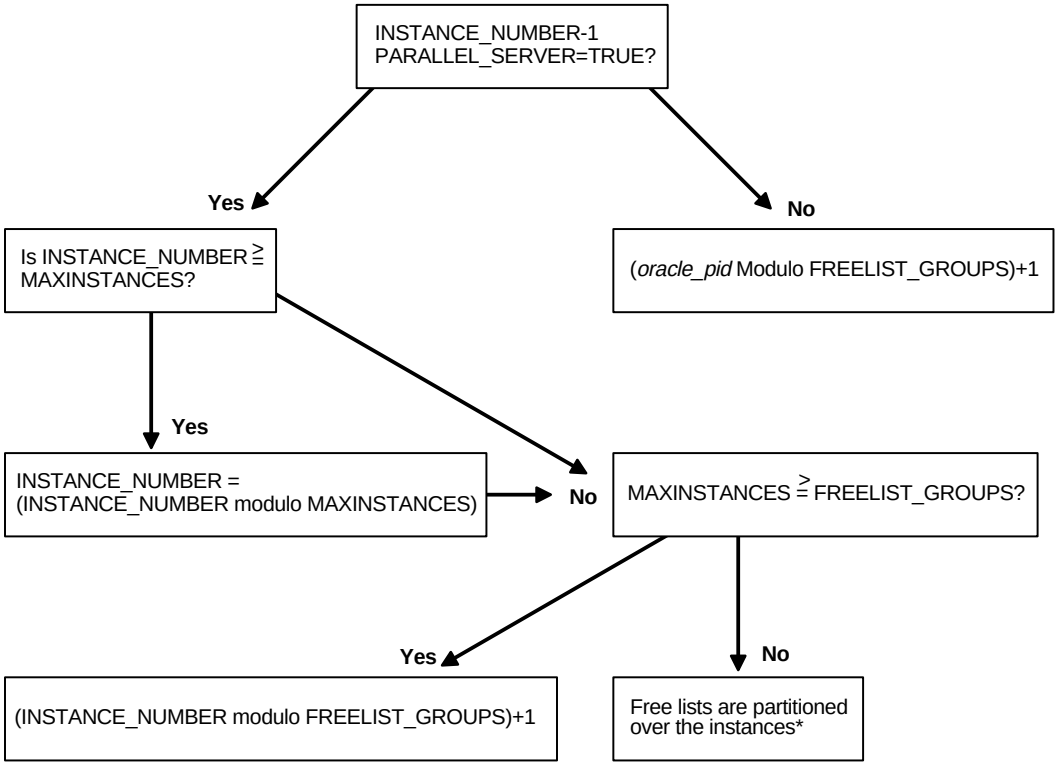
Note: In a multi-instance environment, information about multiple free lists and free list groups is not preserved upon import. If you use Export and Import to back up and restore your data, it will be difficult to import the data so that it is partitioned again.

See Also: "Free Lists with Import and Export Utilities" on page B-4
Chapter 12, "Application Analysis", for more information on partitioning data.

How Free Lists and Free List Groups Are Assigned to Instances

Figure 11–6 illustrates the way in which free lists and free list groups are assigned to instances.

Figure 11–6 How Free Lists and Free List Groups Are Assigned



Note: Using the statement ALTER SESSION INSTANCE_NUMBER you can alter the instance number to be larger than the value of MAXINSTANCES. The figure shows how this possibility is taken into account: for the purposes of the internal calculation whereby free list groups are assigned, the instance number is brought back within the boundaries of MAXINSTANCES.

* Free lists are partitioned as follows: If there are 3 instances and 35 free list groups, then instance 1 will handle the first twelve free list groups, instance 2 the next twelve, and instance 3 the remaining eleven. The actual free list group block is determined by hashing *oracle_pid* by the number of free list groups.

Free Lists Associated with Instances, Users, and Locks

This section describes:

- Associating Instances with Free Lists
- Associating User Processes with Free Lists
- Associating PCM Locks with Free Lists

Associating Instances with Free Lists

A table can have separate groups of process free lists that are assigned to particular instances. Each group of free lists can be associated with a single instance, or several instances can share one group of free lists. All instances also have access to the master free list of available space.

Groups of free lists allow you to associate instances with different sets of data blocks for concurrent inserts and updates requiring new space. This reduces contention for the segment header block, which contains information about the master free list of free blocks. For tables that do not have multiple free list groups, the segment header also contains information about free lists for user processes. You can use free list groups to locate the data that an instance inserts and accesses frequently in extents allocated to that instance.

Data partitioning can reduce contention for data blocks. Often the PCM locks that cover blocks in one free list group tend to be held primarily by the instance using that free list group, because an instance that modifies data is usually more likely to reuse that data than other instances. However, if multiple instances take free space from the same extent, they are more likely to contend for blocks in that extent if they subsequently modify the data that they inserted.

Assignment of New Instances to Existing Free List Groups

If MAXINSTANCES is greater than the number of free list groups in the table or cluster, then an instance number maps to the free list group associated with:

instance_number modulo *number_of_free_list_groups*

Note: "Modulo" (or "rem" for "remainder") is a formula for determining which free list group should be used by calculating a remainder value. In the following example there are 2 free list groups and 10 instances. To determine which free list group instance 6 will use, the formula would read 6 modulo 2 = 0. Six divided by 2 is 3 with zero remainder, so instance 6 will use free list group 0. Similarly, instance 5 would use free list group 1 because 5 modulo 2 = 1. Five is divisible by 2 with a remainder of 1.

If there are more free list groups than MAXINSTANCES, then a different hashing mechanism is used.

If multiple instances share one free list group, they share access to every extent specifically allocated to any instance sharing that free list group.

FREELIST GROUPS and MAXINSTANCES

In a system with relatively few nodes, such as a clustered system, the FREELIST GROUPS option for a table should generally have the same value as the MAXINSTANCES option of CREATE DATABASE, which limits the number of instances that can access a database concurrently.

In a massively parallel system, however, MAXINSTANCES could be many times larger than FREELIST GROUPS so that many instances share one group of free lists.

See Also: "Associating Instances, Users, and Locks with Free List Groups" on page 17-9.

Associating User Processes with Free Lists

User processes are associated with process free lists based on their Oracle process IDs. Each user process has access to only one free list in the free list group for the instance on which it is running. Every user process also has access to the master free list of free blocks.

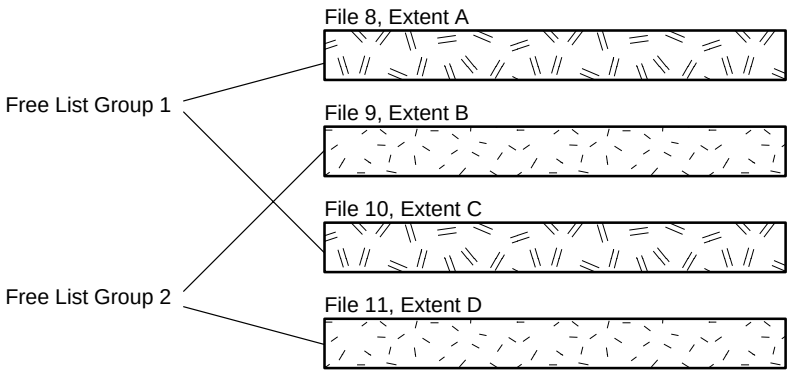
If a table has multiple free lists but does not have multiple free list groups, or has fewer free list groups than the number of instances, then each free list is shared by user processes from different instances.

Associating PCM Locks with Free Lists

If each extent in the table is in a separate datafile, you can use the GC_FILES_TO_LOCKS parameter to allocate specific ranges of PCM locks to each extent, so that each set of PCM locks is associated with only one group of free lists.

Figure 11–7 shows multiple extents in separate files. The GC_FILES_TO_LOCKS parameter allocates 10 locks to files 8 and 10, and 10 locks to files 9 and 11. Extents A and C are in the same free list group, and extents B and D are in another free list group. One set of PCM locks is associated with files 8 and 10, and a different set of PCM locks is associated with files 9 and 11. You do not need separate locks for files which are in the same free list group (such as files 8 and 10, or files 9 and 11).

Figure 11–7 Extents and Free List Groups



This example assumes total partitioning for reads as well as for writes. If more than one instance is to update blocks, then it would still be desirable to have more than one lock per file in order to minimize forced reads and writes. This is because even with a shared lock, *all* blocks held by a lock are subject to forced reads when another instance tries to read even *one* of the locked blocks.

See Also: "Setting GC_FILES_TO_LOCKS: PCM Locks for Each Datafile" on page 15-7.

Controlling the Allocation of Extents

This section covers the following topics:

- Automatic Allocation of New Extents
- Pre-allocation of New Extents
- Dynamic Allocation of Blocks on Lock Boundaries

When a row is inserted into a table and new extents need to be allocated, a certain number of contiguous blocks (specified with !*blocks* in the GC_FILES_TO_LOCKS parameter) are allocated to the free list group associated with an instance. Extents allocated when the table or cluster is first created and new extents that are automatically allocated add their blocks to the master free list (space above the high water mark).

Automatic Allocation of New Extents

When a user explicitly allocates an extent without specifying an instance, or when an extent is automatically allocated to a segment because the system is running out of space (the high water mark cannot be advanced any more), the new extent becomes part of the unused space. It is placed at the end of the extent map, which means that the current high water mark is now in an extent "to the left" of the new one. The new extent is thus added "above" the high water mark.

Pre-allocation of New Extents

You have two options for controlling the allocation of new extents.

- pre-allocating extents to free list groups
- dynamically allocating blocks to free list groups

Pre-allocating extents is a static approach to the problem of preventing automatic allocation of extents by Oracle. You can pre-allocate extents to tables that have free list groups. This means that all the free blocks will be formatted into free lists, which will reside in the free list group of the instance to which you are pre-allocating the extent. This approach is useful if you need to partition data so as to greatly reduce all pinging on insert, or if you need to accommodate objects which you expect will grow.

Note: False pinging will not be eliminated.

See Also: "Pre-allocating Extents (Optional)" on page 17-10.

Dynamic Allocation of Blocks on Lock Boundaries

If you primarily need to accommodate growth, the strategy of dynamically allocating blocks to free list groups would be more effective than pre-allocation of extents. You can use the !*blocks* option of GC_FILES_TO_LOCKS to dynamically allocate blocks to a free list from the high water mark within a lock boundary. This method does not eliminate *all* pinging on the segment header-- rather, it allocates blocks on the fly so that you do not have to pre-allocate extents.

Remember that locks are owned by instances. Blocks are allocated on a per-instance basis--and that is why they are allocated to free list groups. Within an instance, blocks can be allocated to different free lists.

Using this method, you can either explicitly allocate the !*blocks* value, or else leave the balance of new blocks still covered by the existing PCM lock. If you choose the latter, remember that there still may be contention for the existing PCM lock by allocation to other instances. If the PCM lock covers multiple groups of blocks, there

may still be unnecessary forced reads and writes of all the blocks covered by the lock.

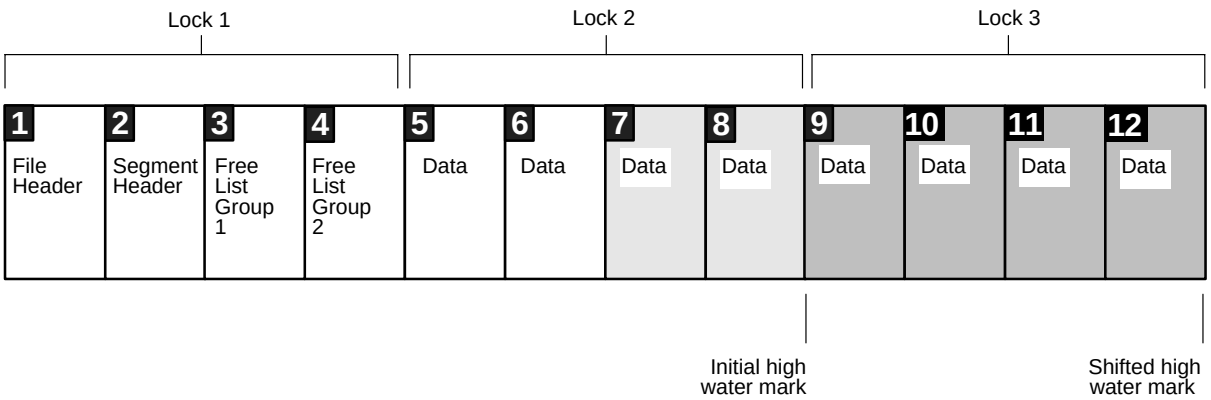
See Also: "Dynamically Allocating Extents" on page 17-14.

Moving the High Water Mark of a Segment

A segment's *high water mark* is the current limit to the number of blocks that have been allocated within the segment. If you are allocating extents dynamically, the high water mark is also the lock boundary. The lock boundary and the number of blocks which will be allocated at one time within an extent must coincide. This value must be the same for all instances.

Consider the following example, in which there are 4 blocks per lock (!4). Locks have been allocated before the block content has been entered. If we have filled datablock D2, held by Lock 2, and then allocate another range of 4 blocks, only the number of blocks which fits within the lock boundary will actually be allocated: in this case, blocks 7 and 8. Both of these are protected by your current lock. With the high water mark at 8, when instance 2 allocates a range of blocks, all four blocks 9 to 12 are allocated, covered by lock 3. The next time instance 1 allocates blocks it will get blocks 13 to 16, covered by lock 4.

Figure 11–8 A File with High Water Mark Moving as Blocks Are Allocated



Example The example in this section assumes that GC_FILES_TO_LOCKS has the following setting for both instances:

```
GC_FILES_TO_LOCKS = "1000!5"
```

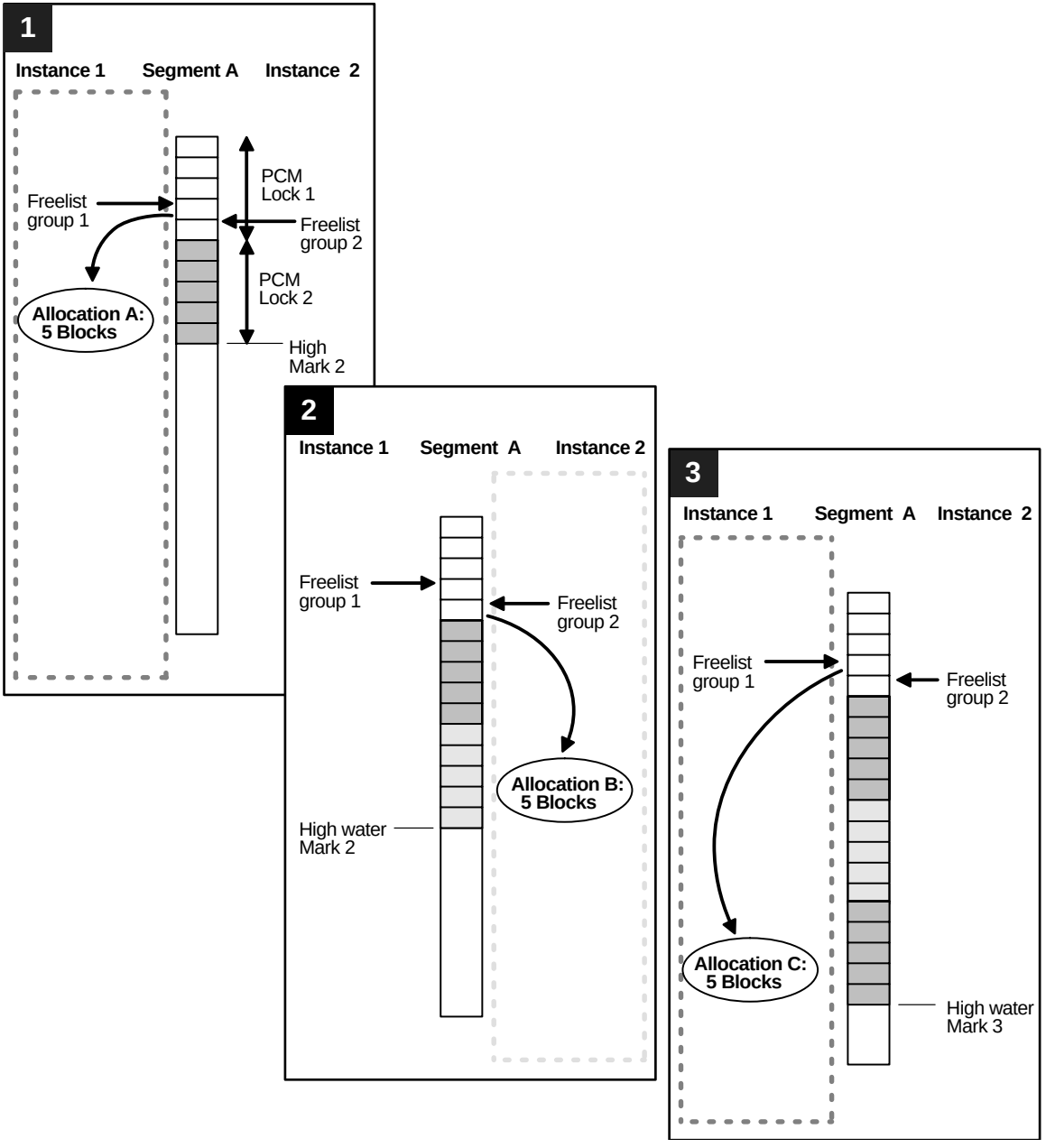
With the EACH option specified, each file in *file_list* is allocated *#locks* number of PCM locks. Within each file, !*blocks* specifies the number of contiguous data blocks to be covered by each lock.

Figure 11–9 shows the incremental process by which the segment grows:

- Stage 1 shows an extent in which instance 1 allocates 5 data blocks, which are protected by Lock 2.
- Stage 2 shows instance 2 allocating 5 more data blocks, protected by Lock 3.
- Stage 3 shows instance 1 once more allocating 5 data blocks, protected by Lock 4.

In this way, if user A on Instance 1 is working on block 10, no one else from either instance can work on any block in the range of blocks that are covered by Lock 2 (that is, blocks 6 through 10).

Figure 11–9 Allocating Blocks Within an Extent



12

Application Analysis

This chapter provides a conceptual framework for optimizing Oracle Parallel Server application design. It includes the following sections:

- How Detailed Must Your Analysis Be?
- Understanding Your Application Profile
- Partitioning Guidelines

See Also: *Oracle8 Tuning* for a discussion of performance tuning principles and method.

How Detailed Must Your Analysis Be?

The level of detail to which you must analyze an application depends upon your goals for the use of Oracle Parallel Server. If you need OPS to boost overall database throughput, then a detailed analysis of the database design and application workload profile will be necessary. This is to ensure that the additional CPU power provided by each node of the parallel server is fully used for application processing. Even if you are using Oracle Parallel Server primarily to provide high availability, careful analysis will enable you to predict the resources that will be needed by your workload.

Experience gained over many benchmark and real applications shows that, for optimal performance, Oracle Parallel Server systems must minimize the computing resources used for parallel cache management. This means minimizing the number of instance lock operations. A successful OPS implementation ensures that each node performs very few instance lock operations and subsequently the machine-to-machine high speed interconnect traffic is within the design limitations of the cluster.

You cannot successfully minimize the number of PCM lock operations during the final fine tuning phase of the database lifetime. Rather, you must plan this early in the physical database design process.

See Also: Chapter 13, “Designing a Database for Parallel Server”, for a case study which shows how to design applications to take advantage of the Oracle Parallel Server.

Understanding Your Application Profile

To understand your application profile you must classify tables according to application functions and access patterns. This section describes:

- Analyzing Application Functions and Table Access Patterns
- Read-only Tables
- Random SELECT and UPDATE Tables
- INSERT, UPDATE, or DELETE Tables
- Planning the Implementation

The following comments apply equally to clustered tables or non-clustered tables.

Analyzing Application Functions and Table Access Patterns

Beyond performing the usual application and data analysis phases, the database designer for parallel server must anticipate the types of transactions or business functions that may cause excessive lock conversion rates. You must cross reference the core application tables and their access patterns with the application functions.

See Also: Chapter 13, “Designing a Database for Parallel Server”, for worksheets you can use to analyze table access patterns.

Read-only Tables

With tables that are predominantly read-only, all Oracle Parallel Server nodes quickly initialize the PCM locks to shared mode and very little lock activity takes place. Read-only tables and their associated index structures require the allocation of very few PCM locks. With this table type you can expect good performance and scalability with Oracle Parallel Server.

Also consider putting tables in read-only tablespaces, using the SQL statement ALTER TABLESPACE READ ONLY. This has two advantages: it speeds up recovery, and no PCM instance locks are required.

Scalability of the parallel query on an Oracle Parallel Server environment is subject to the interconnect speed between the nodes. You may need to run high levels of parallelism just to keep the processors busy. It is not unusual to run a degree of parallelism three times the number of nodes (or processors).

These files should have their own PCM lock as specified in the GC_FILES_TO_LOCKS parameter, even if the application is read-only Large sorts,

such as queries utilizing SORT MERGE JOINs, or with GROUP-BYs and ORDER-BYs, can update the data dictionary in the SYSTEM tablespace.

See Also: "The Four Levels of Scalability You Need" on page 2-2.

"Setting the Degree of Parallelism" in *Oracle8 Tuning*.

Random SELECT and UPDATE Tables

Random SELECT and UPDATE tables (that is, tables that are not partitioned) have transactions that may read and then subsequently update any of the rows in a table. This kind of access requires many lock conversions. First, the instance executing the transaction must obtain a shared PCM lock on the data block. This lock request may cause a lock downgrade operation on another node. The instance executing the transaction must finally obtain an exclusive mode PCM lock when the UPDATE is actually performed.

If user transactions on different Oracle Parallel Server nodes modify data blocks locked by the same PCM lock concurrently, there will be a noticeable performance penalty. In some cases you can reduce this contention by creating additional hashed PCM locks. In large tables, however, hardware and practical limitations may mean that the number of hashed PCM locks you can effectively use may be limited. For example, to reduce false contention you would need millions of hashed PCM locks--but memory limitations and startup time would make this impossible. On supported platforms, fine grain locks offer a viable and economical solution.

For this type of table, if none of the table's index keys are actually updated, then the index's PCM locks are only converted to shared mode and thus require few PCM locks.

INSERT, UPDATE, or DELETE Tables

Transactions on random INSERT, UPDATE and DELETE tables require reading a number of data blocks and then modifying some or all of the data blocks read. This process for each of the data blocks specified again requires converting the PCM lock to shared mode and then converting it to exclusive mode upon block modification. This process has the same performance issues as random SELECT and UPDATE tables.

For this table type more performance issues exist for two main reasons: index data blocks are changed, and contention occurs for data blocks on the table's free list.

In INSERT, DELETE and UPDATE transactions that modify indexed keys, you need to maintain the table's indexes. This process requires the modification of additional index blocks--and so the number of potential lock converts increases. In addi-

tion, index blocks will probably require additional lock converts since users on other nodes will be using the index to access other data. This applies particularly to the initial root components of the index where block splitting may be taking place. This causes more lock converts from null to exclusive and vice versa on all nodes within the cluster.

If the INSERT and DELETE operations are subject to long running transactions, then there is a high chance that another Oracle Parallel Server instance will require read consistency information to complete its transactions. This process will force yet more lock conversions as rollback segment data blocks are flushed to disk and are made available to other instances.

Index block contention involving high lock convert rates must be avoided at all costs, if performance is a critical issue in the Oracle Parallel Server implementation.

Index block contention can be made more extreme when using a sequence number generator to generate unique keys for a table from multiple OPS nodes. When generating unique keys, make the instance number part of the primary key so that each instance performs INSERTs into a different part of the index. Spreading the INSERT load over the full width of the index can improve both single and multiple instance performance.

In INSERT operations the allocation of free space within an extent may also cause high lock convert rates. This is because multiple instances may wish to insert new rows into the same data blocks, or into data blocks which are close together. If these data blocks are managed by the same PCM lock, there will be contention. To avoid this, create tables so as to allow the use of multiple free lists and multiple free list groups.

See Also: Chapter 17, “Using Free List Groups to Partition Data”.

Planning the Implementation

Having analyzed the application workload, you can now plan the application’s OPS implementation. Using the access profile you can see which transactions will run well over multiple Oracle Parallel Server nodes, and which transactions should be executed within a single Oracle Parallel Server node. In many cases compromises and trade-offs are required to ensure that the application performs as needed.

Note: Load balancing between nodes should not be the main objective. Whereas load balancing is useful in a benchmarking situation, it may not be useful in a real-world application. Partitioning is the key to performance in an Oracle Parallel Server system.

Partitioning Guidelines

This section covers the following topics:

- Overview
- Application Partitioning
- Data Partitioning

Overview

The database designer must clearly understand the system performance implications and design trade-offs made by application partitioning. Always bear in mind that your goal is to minimize synchronization: this will result in optimized performance.

As noted earlier, if the number of lock conversions is minimized the performance of the Oracle Parallel Server system will be predictable and scalable. By partitioning the application and/or data you can create and maintain cache affinities of database data with respect to specific nodes of a cluster. A partitioned application ensures that a minimum number of lock conversions are performed, thus data block pinging and Integrated DLM activity should be very modest. If excessive IDLM lock activity occurs in a partitioned application, your partitioning strategy may be inappropriate, or the database creation and tuning process was incorrect.

Application Partitioning

Many partitioning techniques exist to achieve high system performance. One of the simplest ways to break up or partition the load upon the database is to run different applications that access the same database on different nodes of the cluster. For example, one application may only reference a fixed set of tables that reside in one set of datafiles, and another application may reference a different set of tables that reside in a different set of datafiles. These applications can be run on different nodes of a cluster and should yield good performance if the datafiles are assigned different PCM locks. There will be no conflict for the same database objects (since they are in different files) and hence no conflict for the same database blocks.

This scenario is particularly applicable to applications that during the day need to support many users and a high OLTP workload, and during the night need to run a high batch and decision support workload. In this case applications can be partitioned amongst the cluster nodes to sustain good OLTP performance during the day.

This model is very similar to a distributed database model, where tables that are accessed together are stored together. At night, when it is necessary to access tables that may be partitioned for OLTP purposes, you still can exploit the advantages of a single database: all the data is stored effectively within a single database. Advantages include improved batch and decision support, query performance, reduced network traffic, and data replication issues.

With this approach you must ensure that each application's tables and indexes are stored such that one PCM lock does not cover any data blocks that are used by both applications. Should this happen the purpose of partitioning would be lost. To rectify the situation you would store each application's table and index data in separate datafiles.

Applications which share a set of SQL statements perform best when they run on the same instance. Because shared SQL areas are not shared across instances, similar sets of SQL statements should run on one instance to improve memory usage and reduce parsing.

Data Partitioning

Sometimes the partitioning of applications between nodes may not be possible. As an alternative approach, you can partition the database objects themselves. To do this effectively you must analyze the application profile in depth. You may or may not need to split a table into multiple tables. In Oracle Parallel Server situations the partitioning process can involve horizontal partitioning of the table between predefined key ranges.

In addition to partitioning and splitting database objects, you must ensure that each transaction from a user is executed upon the correct OPS instance. The correct node for execution of the transaction is a function of the actual data values being used in the transaction. This process is more commonly known as *data-dependent routing*.

The process of partitioning a table for purposes of increasing parallel server performance brings with it various development and administration implications.

From a development perspective, as soon as the table is partitioned the quantity and complexity of application code increases. In addition, partitioning a table may compromise the performance of other application functions such as batch and decision support queries.

The administration of data-dependent routing may be complex and involve additional application code. The process may be simplified if a transaction processing monitor (TPM) or RPC mechanism is used by the application. It is possible to code

into the configuration of the TPM a data-dependent routing strategy based upon the input RPC arguments. Similarly, this process could be coded into piece of procedural code using a case statement to determine which instance should execute the transaction.

See Also: "Client-Server Systems" on page 1-22

Part III

OPS System Development Procedures

13

Designing a Database for Parallel Server

This chapter prescribes a general methodology for designing systems optimized for the Oracle Parallel Server.

- Overview
- Case Study: From First-Cut Database Design to OPS
- Analyze Access to Tables
- Analyze Transaction Volume by Users
- Partition Users and Data
- Partition Indexes
- Implement Hashed or Fine Grain Locking
- Implement and Tune Your Design

Overview

This chapter provides techniques for designing a new application for use with Oracle Parallel Server. You can also use these analytical techniques to evaluate existing applications and see how well suited they are for migration to a parallel server.

Attention: Always bear in mind that your goal is to minimize synchronization: this will result in optimized performance.

The chapter assumes that you have made at least a first cut of your database design. To optimize your design for a parallel server, follow the methodology suggested here.

1. Make a first cut of your database design.
2. Analyze access to tables.
3. Analyze transaction volume.
4. Decide how to partition users and data.
5. Decide how to partition indexes, if necessary.
6. Choose hashed or fine grain locking.
7. Implement and tune your design.

Case Study: From First-Cut Database Design to OPS

A simple case study is used throughout this chapter to demonstrate analytical techniques in practice. Although your application will differ, this example will help you to understand the process.

- "Eddie Bean" Catalog Sales
- Tables
- Users
- Application Profile

"Eddie Bean" Catalog Sales

The case study concerns the Eddie Bean catalog sales company, which has many order entry clerks who take telephone orders for various products. Shipping clerks fulfill orders, accounts receivable clerks handle billing. Accounts payable clerks handle orders for supplies and services which the company requires internally. Sales managers and financial analysts run reports on the data. This company's financial application has three areas which operate on a single database:

- order entry
- accounts payable
- accounts receivable

Tables

Tables from the Eddie Bean database include:

Table 13–1 "Eddie Bean" Sample Tables

Table	Contents
ORDER_HEADER	Order number, customer name and address.
ORDER_ITEMS	Products ordered, quantity, and price.
ORGANIZATIONS	Names, addresses, phone numbers of customers and suppliers.
ACCOUNTS_PAYABLE	Tracks the company's internal purchase orders and payments for supplies and services.
BUDGET	Balance sheet of the company's expenses and income.
FORECASTS	Projects future sales and records current performance.

Users

Various application users access the database to perform different functions:

- order entry clerks
- accounts payable clerks
- accounts receivable clerks
- shipping clerks
- sales manager
- financial analyst

Application Profile

Operation of the Eddie Bean application is fairly consistent throughout the day: order entry, order processing, and shipping are performed all day and not, for example, segregated into one-hour slots.

About 500 orders are entered per day. Each order header is updated about 4 times through its lifetime (so we expect about 4 times as many updates as inserts). There are many selects, because lots of people are querying order headers--people doing sales work, financial work, shipping, tracing the status of orders, and so on.

There are about 4 items per order. Order items are never updated: an item may be deleted and another item entered.

The ORDER_HEADER table has four indexes, and each of the other tables has a primary key index only.

Budget and Forecast activity has a much lower volume than the order tables. They are read frequently, but modified infrequently. Forecasts are updated more often than Budget, and are deleted once they go into actuals.

The vast bulk of the deletes are performed as a batch job at night: this maintenance activity does not therefore need to be included in the analysis of normal functioning of the application.

Analyze Access to Tables

Begin by analyzing the existing (or expected) access patterns for the tables in your database. You will then decide how to partition the tables, and group them according to access pattern.

- Table Access Analysis Worksheet
- Case Study: Table Access Analysis

Table Access Analysis Worksheet

List all your high-activity database tables in a worksheet like this:

Table 13–2 Table Access Analysis Worksheet

Table Name	Daily Access Volume							
	Read Access		Write Access					
	Select		Insert		Update		Delete	
	Opera-tions	I/Os	Opera-tions	I/Os	Opera-tions	I/Os	Opera-tions	I/Os

To fill out this worksheet, you estimate the volume of operations of each type, and then calculate the number of reads and writes (I/Os) the operations will entail.

Estimating Volume of Operations

For each type of operation that will be performed on a table, enter a figure that reflects *the normal volume you would expect in the course of a day.*

Attention: The emphasis throughout this analysis is on *relative values*—gross figures that describe the normal use of an application. Even if an application does not yet exist, you can nonetheless project types of users and estimate relative levels of activity. Maintenance activity on the tables is not generally relevant to this analysis.

Calculating I/Os per Operation

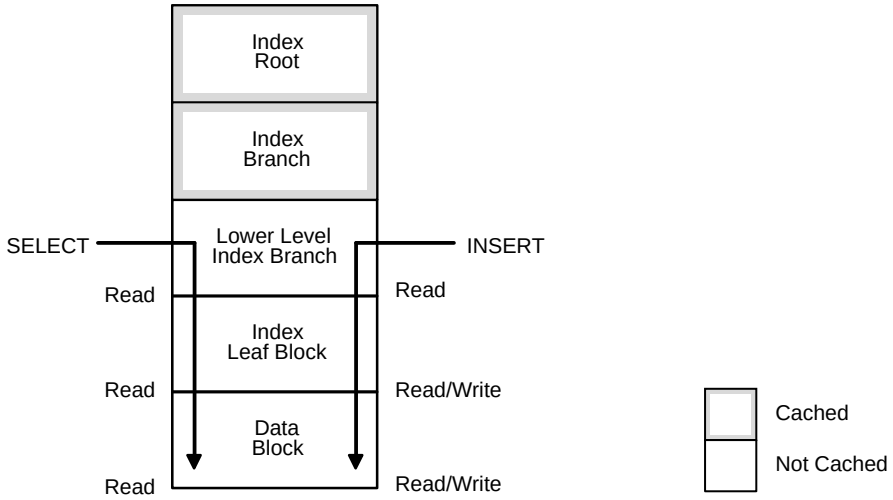
For each value in the Operations column, calculate the number of I/Os that will be generated (using a worst-case scenario).

Note that the SELECT operation involves read access, and the INSERT, UPDATE and DELETE operations involve both read and write access. These operations access not only data blocks, but also any related index blocks.

Attention: The number of I/Os generated per operation changes *by table* depending on the access path of the table, and the table's size. It also changes depending on the number of indexes a table has. A small index, for example, may have only a single index branch block.

For example, Figure 13–1 illustrates read and write access to data in a large table in which two levels of the index are not in the buffer cache and only a high level index is cached in the SGA.

Figure 13–1 Number of I/So per SELECT or INSERT Operation



In this example, assuming that you are accessing data via the primary key, a SELECT entails three I/Os:

1. one I/O to read the first lower level index block
2. one I/O to read the second lower level index block
3. one I/O to read the data block

Note: If all of the root and branch blocks are in the SGA, a SELECT may entail only two I/Os: read leaf index block, read data block.

An INSERT or DELETE statement entails at least five I/Os:

1. one I/O to read the data block
2. one I/O to write the data block
3. three I/Os *per index*: 2 to read the index entries and 1 to write the index

One UPDATE in this example entails seven I/Os:

1. one I/O to read the first lower level index block
2. one I/O to read the second lower level index block
3. one I/O to read the data block
4. one I/O to write the data block
5. one I/O to read the first lower level index block again
6. one I/O to read the second lower level index block again
7. one I/O to write the index block

Note: An INSERT or DELETE affects *all* indexes, but an UPDATE sometimes may affect only *one* index. Check to see how many index keys are changed.

I/Os per Operation for Sample Tables

In the case study, number of I/Os per operation differs from table to table—because the number of indexes differs from table to table.

Table 13–3 shows how many I/Os are generated by each type of operation on the ORDER_HEADER table. It assumes that the ORDER_HEADER table has four indexes.

Table 13–3 Number of I/Os per Operation: Sample ORDER_HEADER Table

Operation	SELECT	INSERT	UPDATE	DELETE
Type of Access	read	read / write	read / write	read / write
Number of I/Os	3	14	7	14

Attention: Remember that you must adjust these figures depending upon the actual number of indexes and access path for each table in your database.

Table 13–4 shows how many I/Os are generated per operation for each of the other tables in the case study, assuming that each of them has a primary key index only.

Table 13–4 Number of I/Os per Operation: Other Sample Tables

Operation	SELECT	INSERT	UPDATE	DELETE
Type of Access	read	read / write	read / write	read / write
Number of I/Os	3	5	7	5

For purposes of this analysis you can disregard the fact that any changes made to the data will also generate rollback segments, entailing additional I/Os. These I/Os are instance-based, and so should not cause problems with your parallel server application.

See Also: *Oracle8 Concepts* for more information about indexes.

Case Study: Table Access Analysis

Table 13–5 shows rough figures reflecting normal use of the application in the case study.

Table 13–5 Case Study: Table Access Analysis Worksheet

	Daily Access Volume							
	Read Access		Write Access					
	Select		Insert		Update		Delete	
Table Name	Operations	I/Os	Operations	I/Os	Operations	I/Os	Operations	I/Os
ORDER_HEADER	20,000	60,000	500	7,000	2,000	14,000	1,000	14,000
ORDER_ITEM	60,000	180,000	2,000	10,000	0	0	4,030	20,150
ORGANIZATIONS	40,000	120,000	10	50	100	700	0	0
BUDGET	300	900	1	5	2	14	0	0
FORECASTS	500	1,500	1	5	10	70	2	10
ACCOUNTS_PAYABLE	230	690	50	250	20	140	0	0

The following conclusions can be drawn from this table:

- Only the ORDER_HEADER and ORDER_ITEM tables have significant levels of write access.
- ORGANIZATIONS, by contrast, is predominantly a lookup table; while a certain number of INSERT, UPDATE, and DELETE operations will be performed to maintain it, its normal use is SELECT-only.

Analyze Transaction Volume by Users

Begin by analyzing the existing (or expected) access patterns for the tables in your database. You will then decide how to partition the tables, and group them according to access pattern.

- Transaction Volume Analysis Worksheet
- Case Study: Transaction Volume Analysis

Transaction Volume Analysis Worksheet

For each table which has a high volume of write access, analyze the transaction volume per day for each type of user.

Attention: For read-only tables, you do *not* need to analyze transaction volume by user type.

Use worksheets like this:

Table 13–6 Transaction Volume Analysis Worksheet

Table Name:									
		Daily Transaction Volume							
		Read Access		**Write Access**					
		Select		**Insert**		**Update**		**Delete**	
Type of User	**No.Users**	**Operations**	**I/Os**	**Operations**	**I/Os**	**Operations**	**I/Os**	**Operations**	**I/Os**

Begin by estimating the volume of transactions by each type of user, and then calculate the number of I/Os entailed.

Case Study: Transaction Volume Analysis

The following tables show transaction volume analysis of the three tables in the case study which have a high level of write access: ORDER_HEADER, ORDER_ITEMS, and ACCOUNTS_PAYABLE.

ORDER_HEADER Table

Table 13–7 shows rough figures for the ORDER_HEADER table in the case study.

Table 13–7 Case Study: Transaction Volume Analysis: ORDER_HEADER Table

Table Name: ORDER_HEADER									
		Daily Transaction Volume							
		Read Access		**Write Access**					
		Select		**Insert**		**Update**		**Delete**	
Type of User	**No.Users**	**Oper-ations**	**I/Os**	**Oper-ations**	**I/Os**	**Oper-ations**	**I/Os**	**Oper-ations**	**I/Os**
OE clerk	25	5,000	15,000	500	7,000	0	0	0	0
AP clerk	5	0	0	0	0	0	0	0	0
AR clerk	5	6,000	18,000	0	0	1,000	7,000	0	0
Shipping clerk	4	4,000	12,000	0	0	1,000	7,000	0	0
Sales manager	2	3,000	9,000	0	0	0	0	0	0
Financial analyst	2	2,000	6,000	0	0	0	0	0	0

The following conclusions can be drawn from this table:

- OE clerks perform all inserts on this table.
- AR and shipping clerks perform all updates.
- Sales managers and financial analysts only perform select operations on it.
- AP clerks never touch the table.

Deletes are performed as a maintenance operation, so they need not be considered in this analysis.

Furthermore, the application developers realize that sales managers normally access data for the current month, whereas financial analysts access historical data.

ORDER_ITEMS Table

Table 13–8 shows rough figures for the ORDER_ITEMS table in the case study.

Table 13–8 Case Study: Transaction Volume Analysis: ORDER_ITEMS Table

Table Name: ORDER_ITEMS									
		Daily Transaction Volume							
		Read Access		**Write Access**					
		Select		**Insert**		**Update**		**Delete**	
Type of User	**No.Users**	**Oper-ations**	**I/Os**	**Oper-ations**	**I/Os**	**Oper-ations**	**I/Os**	**Oper-ations**	**I/Os**
OE clerk	25	15,000	45,000	2,000	10,000	0	0	20	100
AP clerk	5	0	0	0	0	0	0	0	0
AR clerk	5	18,000	54,000	0	0	0	0	10	50
Shipping clerk	4	12,000	36,000	0	0	0	0	0	0
Sales manager	2	9,000	27,000	0	0	0	0	0	0
Financial analyst	2	6,000	18,000	0	0	0	0	0	0

The following conclusions can be drawn from this table:

- OE clerks perform all inserts on this table.
- Updates are rarely performed
- AR clerks, shipping clerks, sales managers and financial analysts perform a heavy volume of select operations on the table.
- AP clerks never touch the table.

Note that the ORDER_HEADER table has more writes than ORDER_ITEMS because the order header tends to require more changes of status (such as address changes) than the list of available products. The ORDER_ITEM table is seldom updated because new items are listed as journal entries, instead.

ACCOUNTS_PAYABLE Table

Table 13–9 shows rough figures for the ACCOUNTS_PAYABLE table in the case study.

Although this table does not have a particularly high level of write access, we have analyzed it because it contains the main operation that the AP clerks perform.

Table 13–9 Case Study: Transaction Volume Analysis: ACCOUNTS_PAYABLE Table

Table Name: ACCOUNTS_PAYABLE									
		Daily Transaction Volume							
		Read Access		**Write Access**					
		Select		**Insert**		**Update**		**Delete**	
Type of User	**No.Users**	**Oper-ations**	**I/Os**	**Oper-ations**	**I/Os**	**Oper-ations**	**I/Os**	**Oper-ations**	**I/Os**
OE clerk	25	0	0	0	0	0	0	0	0
AP clerk	5	200	600	50	250	20	140	0	0
AR clerk	5	0	0	0	0	0	0	0	0
Shipping clerk	4	0	0	0	0	0	0	0	0
Sales manager	2	0	0	0	0	0	0	0	0
Financial analyst	2	30	90	0	0	0	0	0	0

The following conclusions can be drawn from this table:

- Accounts payable clerks send about 50 purchase orders per day to suppliers. These clerks are the only users who change the data in this table.
- Financial analysts occasionally study the information.

Deletes are performed as a maintenance operation, so they need not be considered in this analysis.

Partition Users and Data

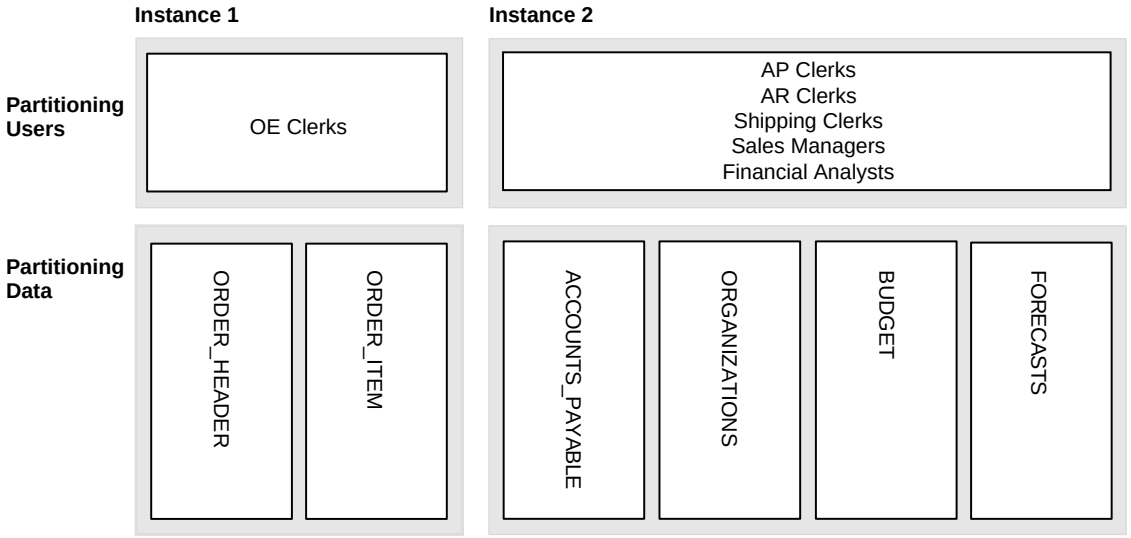
Your goal is now to partition applications across instances. This can involve splitting types of users across instances, and partitioning data that needs to be written only by certain types of user. This will minimize the amount of contention on your system. This section covers:

- Case Study: Initial Partitioning Plan
- Case Study: Further Partitioning Plans

Case Study: Initial Partitioning Plan

In the case study, for example, the large number of Order Entry clerks who do heavy insert activity on the ORDER_HEADER and ORDER_ITEM tables should not be split across machines. They should be concentrated on one node along with the two tables they use so intensively. A good starting point, then, would be to set aside one node for the OE clerks, and one node for all the other users, as illustrated in Figure 13–2.

Figure 13–2 Case Study: Partitioning Users and Data



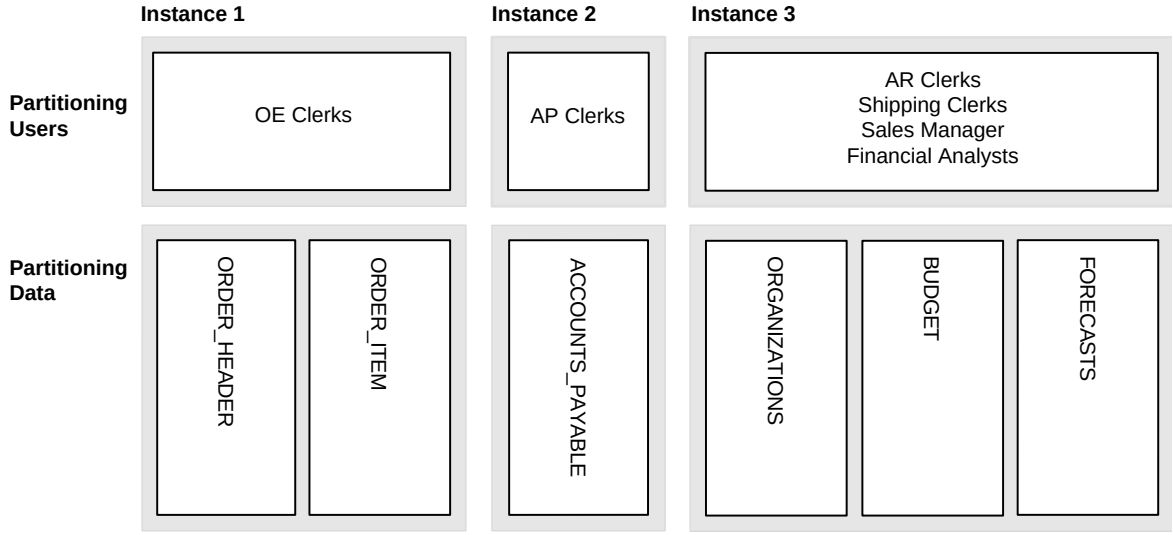
This system would probably be well balanced across nodes. The database intensive reporting done by financial analysts takes a good deal of system resources, whereas the transactions run by the OE clerks are relatively lightweight.

This kind of load balancing of the number of users across the system is typically useful, but not always critical. Load balancing has a lower priority for tuning than reducing contention. not vitally important that financial analysts have the current day's data--if they are primarily interested in looking at historical data. (This would not be appropriate if they needed up-to-the minute data.)

Case Study: Further Partitioning Plans

In the case study it is also clear that the Accounts Payable data is written exclusively by AP clerks. This data and set of users can also be very effectively partitioned onto a separate instance, as shown in Figure 13–3.

Figure 13–3 Case Study: Partitioning Users and Data: Design Option 1



When all users who need write access to a certain part of the data are concentrated on one node, the PCM locks will all reside on that node. In this way lock ownership will not have to go back and forth between instances.

Two design options suggest themselves, based on this analysis.

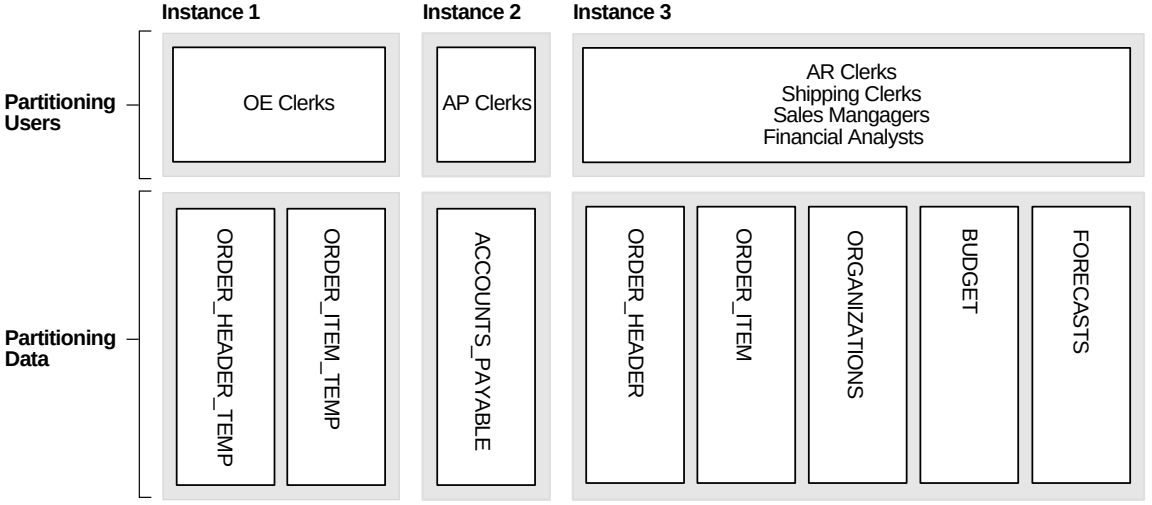
Design Option 1

You can set up the system just as shown above, with all of the order entry clerks together on one instance so as to minimize contention for exclusive PCM locks on the table. In this way sales managers and financial analysts could get up to the minute information. Since they do want data that is predominantly historical, there should still not be too much contention for current records.

Design Option 2

Alternatively, you could implement a separate ORDER_ITEM / ORDER_HEADER temporary table purely for the taking of new orders. Overnight, you could incorporate changes into the main table against which all queries are performed. This solution would work well if it is not vitally important that financial analysis have the current day's data—if they are primarily interested in looking at historical data. (This would not be appropriate if they needed up-to-the-minute data.)

Figure 13–4 Case Study: Partitioning Users and Data: Design Option 2



Partition Indexes

You need to consider index partitioning if multiple nodes in your system are inserting into the same index. In this situation you must make sure that the different instances insert into different points within the index.

(The problem is avoided in the Eddie Bean case study because application and data usage are partitioned.)

See Also: "Creating Free Lists for Indexes" on page 17-7 for tips on using free lists, free list groups, and sequence numbers to avoid contention on indexes.

"Pinpointing Lock Contention within an Application" on page 19-5 regarding indexes as a point of contention.

Implement Hashed or Fine Grain Locking

For many applications, the DBA needs to decide whether to use hashed or fine grain locking for particular database files.

On very large tables the locking mode you use will have a strong impact on performance. If one node in exclusive mode gives 100% performance with hashed locking, one node in shared mode might give 70% of that performance with fine grain locking. The second node in shared mode would also give 70% performance. With hashed locking, the more nodes are added, the more the performance degrades. Fine grain locking is thus a more scalable solution.

You should design for worst case (hashed locking). Then, in the design or monitoring phase if you come to a situation where you have too many locks, or if you suspect false pings, you should try fine grain locking.

Begin with an analysis on the database level. You can use a worksheet like this:

Table 13–10 Worksheet: Database Analysis for Hashed or Fine Grain Locking

Block Class	Relevant Parameter(s)	Use Fine Grain or Hashed Locking?

Next, list files and database objects in a worksheet like the following. Decide which locking mode to use for each file.

Table 13–11 Worksheet: When to Use Hashed or Fine Grain Locking

Filename	Objects Contained	Use Fine Grain or Hashed Locking?

See Also: "Applying Fine Grain and Hashed Locking to Different Files" on page 9-21.

Implement and Tune Your Design

Thus far you have conducted an analysis using gross figures. To finalize your design you must now either prototype the application or implement it in practice—and get it running. By observing the system in action you can tune it further. Try the following techniques:

- Identify blocks that are being pinged: find out where contention exists.
- Consider moving users from one instance to another in order to reduce the amount of pinging and false pinging.
- If a high level of false pinging appears, consider increasing the granularity of the locks (placing more locks on each file).
- If there is pinging on inserts, adjust the free lists or use multiple sequence number generators so that inserts occur in different parts of the index.

See Also: Chapter 19, “Tuning the System to Optimize Performance”
Oracle8 Tuning

14

Creating a Database & Objects for Multiple Instances

This chapter describes:

- Creating a Database for a Multi-instance Environment
- Creating Database Objects to Support Multiple Instances
- Changing the Value of CREATE DATABASE Options

Creating a Database for a Multi-instance Environment

This section covers aspects of database creation that are specific to a parallel server:

- Summary of Tasks
- Setting Initialization Parameters for Database Creation
- Creating a Database and Starting Up
- Setting CREATE DATABASE Options

Summary of Tasks

Database creation tasks specific to the parallel server can be summarized as follows:

1. Set initialization parameters, including log archiving.
2. With parallel server disabled, enter the CREATE DATABASE statement, setting MAXINSTANCES and other important options that are specific to a multi-instance environment.
3. Create rollback segments for each node.
4. Dismount the database, then remount it with parallel server enabled. Then start up the parallel server.

See Also: "Creating a Database" in *Oracle8 Administrator's Guide*.

Setting Initialization Parameters for Database Creation

Certain initialization parameters that are critical at database creation or that affect certain database operations must have the same value for every instance in a parallel server. Be sure that these are set appropriately before you create a database for a multi-instance environment.

Using ARCHIVELOG Mode

To enable the ARCH process while creating a database, you must set the initialization parameter LOG_ARCHIVE_START to TRUE. Then you can change the mode to ARCHIVELOG with the ALTER DATABASE statement before you start up the instance that creates the database.

Alternatively, you can reduce overhead by creating the database in NOARCHIVELOG mode (the default). Then change to ARHIVELOG mode.

You cannot use the STARTUP command to change the database archiving mode. After creating a database, you can use the following Server Manager commands to change archiving mode and reopen the database with parallel server enabled:

```
ALTER DATABASE CLOSE;
ALTER DATABASE ARCHIVELOG;
SHUTDOWN;
STARTUP PARALLEL;
```

See Also: "Archiving the Redo Log Files" on page 21-2
"Parameters Which Must Be Identical on Multiple Instances" on page 18-10

Creating a Database and Starting Up

Use the standard procedure to create a database.

Attention: The CREATE DATABASE statement mounts and opens the newly created database, leaving the parallel server disabled. You must close and dismount the database, then remount it with parallel server enabled.

1. Start Server Manager.
2. Connect with SYSDBA privileges.
3. Start up an instance with the NOMOUNT option.
4. Issue the CREATE DATABASE statement.
5. Create additional rollback segments and threads, as needed.
6. Close and dismount the database.

   ```
   SVRMGR> SHUTDOWN
   ```
7. Update the initialization files to be sure they point to the proper rollback segments and threads, and that parallel server is enabled.
8. Remount the database.

   ```
   SVRMGR> STARTUP [ OPEN databasename ]
   ```

See Also: "Starting Up Instances" on page 18-12

Setting CREATE DATABASE Options

This section describes CREATE DATABASE options specific to the parallel server.

Setting MAXINSTANCES

The MAXINSTANCES option of CREATE DATABASE limits the number of instances that can access a database concurrently. MAXINSTANCES defaults to the maximum value specific to your operating system; on most systems the default is 2.

For a parallel server, you should set MAXINSTANCES to a value greater than the maximum number of instances you expect to run concurrently. In this way, if instance A fails and is being recovered by instance B, you will be able to start instance C before instance A is fully recovered.

Setting MAXLOGFILES and MAXLOGMEMBERS

The MAXLOGFILES option of CREATE DATABASE specifies the maximum number of redo log groups that can be created for the database, and the MAXLOGMEMBERS option specifies the maximum number of members (copies) per group.

For a parallel server, you should set MAXLOGFILES to the maximum number of threads possible, times the maximum anticipated number of groups per thread.

Setting MAXLOGHISTORY

The MAXLOGHISTORY option of CREATE DATABASE specifies the maximum number of redo log files that can be recorded in the log history of the control file. The log history is used for automatic media recovery of a parallel server.

For a parallel server, you should set MAXLOGHISTORY to a large value, such as 1000. The control files can then only store information about this number of redo log files. When the log history exceeds this limit, the old history entries are overwritten in a circular fashion. The default for MAXLOGHISTORY is zero, which disables the log history.

Setting MAXDATAFILES

The MAXDATAFILES option is generic, but a parallel server tends to have more data files and log files than standard systems. On your platform the default value of this option may be too low.

See Also: *Oracle8 SQL Reference* for complete descriptions of the SQL statements CREATE DATABASE and ALTER DATABASE.
See your Oracle operating system-specific documentation for information on default values of CREATE DATABASE options.

"Redo Log Files" on page 6-3 for more information about redo log groups and members.
"Redo Log History in the Control File" on page 21-6 for more information on MAXLOGHISTORY.

Creating Database Objects to Support Multiple Instances

In order to prepare a new database for the parallel server environment, you must also create and configure the following additional database objects.

- Creating Additional Rollback Segments
- Configuring the Online Redo Log for a Parallel Server
- Providing Locks for Added Datafiles

Creating Additional Rollback Segments

You must create at least one rollback segment for each instance of a parallel server. To avoid contention, create these rollback segments in a tablespace other than the SYSTEM tablespace.

You must create and bring online one additional rollback segment in the SYSTEM tablespace before you can create rollback segments in any other tablespace. The instance that creates the database can create this additional rollback segment and new tablespaces, but it cannot create database objects in non-SYSTEM tablespaces until you bring the additional rollback segment online.

Using Private Rollback Segments

To allocate a private rollback segment to one instance, follow these steps:

1. Create the rollback segment with the SQL statement CREATE ROLLBACK SEGMENT, omitting the keyword PUBLIC. Optionally, before creating the rollback segment you can create a tablespace for it.
2. Specify the rollback segment in the instance's parameter file by naming it as a value for the parameter. This reserves the rollback segment for that instance.
3. Use ALTER ROLLBACK SEGMENT to bring the rollback segment online. You can also restart the instance to use the reserved rollback segment.

A private rollback segment should be specified in only one parameter file so that it is associated with only one instance. If an instance attempts to acquire a private rollback segment that another instance has already acquired, Oracle generates a message and prevents the instance from starting up.

Using Public Rollback Segments

Any instance can create a public rollback segment, which can then be *claimed* by any instance when it starts up. Once a rollback segment has been claimed, it is only used by the instance that claimed it until the instance shuts down, releasing the rollback segment for use by another instance.

To create a public rollback segment, use the SQL statement CREATE PUBLIC ROLLBACK SEGMENT.

Typically, the parameter file for any particular instance does not specify public rollback segments because they are assumed to be available to any instance needing them. However, if another instance is not already using it, you can name a public rollback segment as a value of the ROLLBACK_SEGMENTS parameter.

Public rollback segments are identified in the data dictionary view DBA_ROLLBACK_SEGS as having the owner PUBLIC.

If the parameter file omits the ROLLBACK_SEGMENTS initialization parameter, the instance uses public rollback segments by default.

A public rollback segment is brought online when an instance that requires public rollback segments starts up and acquires it. However, starting an instance that uses public rollback segments does not ensure that any particular public rollback segment comes online, unless the instance acquires all of the available public rollback segments. Once acquired, a public rollback segment is used exclusively by the acquiring instance.

Bringing online, taking offline, creating, and dropping rollback segments, whether private or public, is the same whether parallel server is enabled or disabled.

Private rollback segments stay offline until brought online or the owning instance restarts. A public rollback segment stays offline until brought online for a specific instance or until an instance that requires a public rollback segment starts up and acquires it.

If you need to keep a public rollback segment offline and do not want to drop it and re-create it, you must ensure no instance starts up that requires public rollback segments.

Monitoring Rollback Segments

You can use the Server Manager command MONITOR ROLLBACK to display information about the status of the rollback segments that the current instance uses.

Alternatively, you can query the dynamic performance views V$ROLLNAME and V$ROLLSTAT for information about the current instance's rollback segments.

Use the Server Manager command CONNECT *@instance-path* to change the current instance before using the MONITOR command or querying the V$ views. You must have Net8 installed to use the CONNECT command for an instance on a remote node.

You can also query the data dictionary views DBA_ROLLBACK_SEGS and DBA_SEGMENTS for information about the current status of all rollback segments in your database.

For example, to list all the current rollback segments, you can query DBA_ROLLBACK_SEGS with the following statement:

```
SELECT segment_name, segment_id, owner, status
      FROM dba_rollback_segs
```

This query displays the rollback segment's name, ID number, owner, and whether it is in use, as shown in the following example:

```
SEGMENT_NAME                SEGMENT_ID     OWNER      STATUS
--------------------------- ----------     ------     ------------
SYSTEM                               0     SYS        ONLINE
PUBLIC_RS                            1     PUBLIC     ONLINE
USERS1_RS                            2     SYS        ONLINE
USERS2_RS                            3     SYS        OFFLINE
USERS3_RS                            4     SYS        ONLINE
USERS4_RS                            5     SYS        ONLINE
PUBLIC2_RS                           6     PUBLIC     OFFLINE
```

In the above example, rollback segments identified as owned by user SYS are private rollback segments; the rollback segments identified as owned by user PUBLIC are public rollback segments. The view DBA_ROLLBACK_SEGS also includes information (not shown) about the tablespace containing the rollback segment, the datafile containing the segment header, and the extent sizes. The view DBA_SEGMENTS includes additional information about the number of extents in each rollback segment and the segment size.

See Also: *Oracle8 Administrator's Guide* for more information about rollback segments, and about connecting to a database.
Oracle Net8 Administrator's Guide and your Oracle system-specific documentation for the format of the connect string in *instance-path.*
Oracle8 Reference for a description of DBA_ROLLBACK_SEGS and DBA_SEGMENTS, and other dynamic performance views.

Configuring the Online Redo Log for a Parallel Server

Each database instance has its own "thread" of online redo, consisting of its own online redo log groups. When running a parallel server, two or more instances concurrently access a single database, and each instance must have its own thread. This section explains how to configure these online redo threads for multiple instances with a parallel server.

You must create each thread with at least two redo log files (or multiplexed groups), and you must enable the thread before an instance can use it.

The CREATE DATABASE statement creates thread number 1 as a public thread and enables it automatically. You must use the ALTER DATABASE statement to create and enable subsequent threads.

Creating Threads

Threads can be either public or private. The initialization parameter THREAD assigns a unique thread number to the instance. If THREAD is zero, the default, the instance acquires an available public thread by default.

Each thread must be created with at least two redo log files, or multiplexed groups, and the thread must then be enabled before an instance can use it**.**

The CREATE DATABASE statement creates thread number 1 as a public thread and enables it automatically. Subsequent threads must be created and enabled with the ALTER DATABASE statement. For example, the following statements create thread 2 with two groups of three members each, as shown in Figure 6–1 on page 6 - 4:

```
ALTER DATABASE ADD LOGFILE THREAD 2
  GROUP 4 (disk1_file4, disk2_file4, disk3_file4) SIZE 1M REUSE
  GROUP 5 (disk1_file5, disk2_file5, disk3_file5) SIZE 1M REUSE;
ALTER DATABASE ENABLE PUBLIC THREAD 2;
```

If you omit the keyword PUBLIC when you enable the thread, it will be a private thread that cannot be acquired by default. Only one thread number may be specified in the ALTER DATABASE ADD LOGFILE statement, and the THREAD clause must be specified if the thread number of the current instance was chosen by default.

Disabling Threads

You can disable a public or private thread with the statement ALTER DATABASE DISABLE THREAD. You cannot disable a thread if an instance using the thread has the database mounted. To change a thread from public to private, or vice versa, you must disable it and then enable it again. An instance cannot disable its own thread. The database must be open when you disable or enable a thread.

When you disable a thread, Oracle marks its current redo log file as needing to be archived. If you want to drop that file, you might need to first archive it manually.

An error or failure while a thread is being enabled can result in a thread that has a current set of log files but is not enabled. These log files cannot be dropped or archived. In this case, you should disable the thread, even though it is already disabled, then enable it.

Setting the Log's Mode

The mode of using the redo log (ARCHIVELOG or NOARCHIVELOG) is set at database creation. Although rarely necessary, the archive mode can be changed by the SQL statement ALTER DATABASE. When archiving is enabled, online redo log files cannot be reused until they are archived. To switch archiving mode, the database must be mounted with parallel server disabled, but it cannot be open.

The redo log mode is associated with the database rather than with individual instances. For most purposes, all instances should use the same archiving method (automatic or manual) if the redo log is being used in ARCHIVELOG mode.

Changing the Redo Log

You can change the configuration of the redo log (add, drop, or rename a log file or log file member) while the database is mounted with parallel server either enabled or disabled. The only restrictions are that you cannot drop or rename a log file or log file member that is currently in use by any thread, and you cannot drop a log file if that would reduce the number of log groups below two for the thread it is in.

Any instance can add or rename redo log files (or members) of any group for any other instance. As long as there are more than two groups for an instance, a redo log group can be dropped from that instance by any other instance. Changes to redo log files and log members take effect on the next log switch.

See Also: "Archiving the Redo Log Files" on page 21-2

Providing Locks for Added Datafiles

If datafiles are added while a parallel server is running, you must evaluate whether enough locks are available to cover the new files.

Added datafiles use the unassigned locks which were created when the value for GC_FILES_TO_LOCKS was set. If the remaining locks are not adequate to protect the new files and avoid contention, then you must provide more locks by adjusting these two GC parameters. Performance problems are likely if you neglect to make these adjustments.

Note that in a read-only database extra locks would not be necessary even if you added many new datafiles. In a database heavily used for inserts, however, you might very well need to provide for more locks.

1. Analyze whether the remaining locks are adequate. If more are needed, then go on to the next step.
2. Shut down the system.
3. Modify the GC_FILES_TO_LOCKS initialization parameter to provide enough locks for the additional datafiles.
4. Restart the system.

Changing the Value of CREATE DATABASE Options

You can use the CREATE CONTROLFILE statement to change the value of the following database parameters for an existing database:

- MAXINSTANCES
- MAXLOGFILES
- MAXLOGMEMBERS
- MAXLOGHISTORY
- MAXDATAFILES

See Also: *Oracle8 SQL Reference* for a description of the statements CREATE CONTROLFILE and ALTER DATABASE BACKUP CONTROLFILE TO TRACE.

15

Allocating PCM Instance Locks

This chapter explains how to allocate PCM locks to datafiles by specifying values for parameters in the initialization file of an instance.

- Planning Your PCM Locks
- Setting GC_FILES_TO_LOCKS: PCM Locks for Each Datafile
- Tips for Setting GC_FILES_TO_LOCKS
- Setting Other GC_* Parameters
- Tuning Your PCM Locks

See Also: Chapter 9, "Parallel Cache Management Instance Locks", for a conceptual discussion of PCM locks and GC_* parameters.
Oracle8 Reference for descriptions of all the initialization parameters used to allocate locks for the parallel server.

Planning Your PCM Locks

This section describes how to plan and maintain PCM locks. It covers:

- Planning and Maintaining Instance Locks
- Key to Allocating PCM Locks
- Examining Your Datafiles and Data Blocks
- Using Worksheets to Analyze PCM Lock Needs
- Mapping Hashed PCM Locks to Data Blocks
- Partitioning PCM Locks Among Instances

Planning and Maintaining Instance Locks

The IDLM allows you to allocate only a finite number of locks. For this reason you need to analyze and plan for the number of locks your application requires. You also need to know how much memory is required by lock or by resource. Consider these ramifications:

- If you attempt to use more locks than the number configured in the IDLM facility, Oracle will get an error message and shut down.
- If you change Oracle GC_* or LM_* initialization parameters to specify large numbers of locks, this will impact the amount of memory used or available in the SGA.
- The number of instances also affects the memory requirements and number of locks needed by your system.

Key to Allocating PCM Locks

The key to assigning locks is to analyze how often data is *changed* (via INSERT, UPDATE, DELETE). You can then figure out how to group objects into files, based on whether they should be read-only or read / write. Finally, assign locks based on the groupings you have made. In general, follow these guidelines:

- Allocate only a few locks to read-only files.
- Allocate more locks to read / write intensive files.
- If the whole tablespace is read-only, you can simply assign it a single lock. (If you were not to assign the tablespace any locks at all, then the system would attempt to use spare locks, for which it must contend with other tablespaces. This will generate a lot of unnecessary forced reads / writes.)

They key distinction is *not* between types of *object* (index or table) but between *operations* which are being performed on an object. The operation dictates the quantity of locks needed.

See Also: Chapter 12, "Application Analysis".

Examining Your Datafiles and Data Blocks

You must allocate locks at various levels:

- Specify the maximum number of PCM locks to be allocated for all datafiles.
- Specify how many locks to allocate to blocks in each datafile.
- Specify particular locks to cover particular classes of datablocks in a given file.

Begin by getting to know your datafiles and the blocks they contain.

How to Determine File ID, Tablespace Name, and Number of Blocks

Use the following command to determine the file ID, file name, tablespace name, and number of blocks for all databases.

```
SQL> SELECT FILE_NAME, FILE_ID, TABLESPACE_NAME, BLOCKS
2  FROM DBA_DATA_FILES;
```

Results are displayed as in the following example:

```
FILE_NAME               FILE_ID     TABLESPACE_NAME     BLOCKS
-------------------------------------------------------------
/v7/data/data01.dbf     1           SYSTEM               200
/v7/data/data02.dbf     2           ROLLBACK            1600
. . .
```

How Many Locks Do You Need?

Use the following approach to estimate the number of locks you need for particular uses.

- Consider the nature of the data and the application.

  Many locks are needed on heavily used, concurrently updated datafiles. But a query-only application does not need many locks—a single lock on the datafile suffices.

- Assign many locks to files which many instances modify concurrently.

  This reduces lock contention, minimizes I/O activity, and increases accessibility of the data in the files.

- Assign fewer locks to files which do not need to be accessed concurrently by multiple instances.

  This avoids unnecessary lock management overhead.
- Examine the objects in your files, and consider what operations are used on them.
- Group read-only objects together in read-only tablespace(s).

Using Worksheets to Analyze PCM Lock Needs

On large applications, you need to carefully study the business processes involved. Worksheets similar to the ones illustrated in this section may be useful.

Figure out the breakdown of operations on your system on a daily basis. The distinction between operations needing X locks and those needing S locks is the key. Every time you have to go from one mode to the other, you need locks. Take into consideration the interaction of different instances on a table. Also take into consideration the number of rows in a block, the number of rows in a table, and the growth rate. Based on this analysis, you can group your objects into files, and assign free list groups.

Figure 15–1 PCM Lock Worksheet 1

Object	Operations needing X mode: Writes			Ops needing S mode: Reads	TS/Datafile
	INSERTS	UPDATES	DELETES	SELECTS	
A		80%		20% full table scan? single row?	
B				100%	
C					
D					

Figure 15–2 PCM Lock Worksheet 2

Object	Instance 1	Instance 2	Instance 3
D	INSERT UPDATE DELETE	SELECT	
E			
F			

Figure 15–3 PCM Lock Worksheet 3

Table Name	TS to put it in	Row Size	Number of Columns

Mapping Hashed PCM Locks to Data Blocks

In many cases, relatively few PCM locks are needed to cover read-only data compared to data which is updated frequently. This is because read-only data can be shared by all instances of a parallel server. Data which is never updated can be covered by a single PCM lock. Data which is not read-only should be covered by more than a single PCM lock.

If data is read-only, then once an instance owns the PCM locks for the read-only tablespace, the instance never disowns them. No distributed lock management operations are required after the initial lock acquisition. For best results, partition your read-only tablespace so that it is covered by its own set of PCM locks.

You can do this by placing read-only data in a tablespace which does not contain any writable data. Then you can allocate PCM locks to the datafiles in the tablespace, using the GC_FILES_TO_LOCKS parameter.

Do not put read-only data and writable data in the same tablespace.

Partitioning PCM Locks Among Instances

You can map PCM locks to particular data blocks in order to partition the PCM locks among instances based on the data each instance accesses.

This technique minimizes unnecessary distributed lock management. Likewise, it minimizes the disk I/O caused by an instance having to write out data blocks because a requested data block was covered by a PCM lock owned by another instance.

For example, if Instance X primarily updates data in datafiles 1, 2, and 3, while Instance Y primarily updates data in datafiles 4 and 5, you can assign one set of PCM locks to files 1, 2, and 3 and another set to files 4 and 5. Then each instance acquires ownership of the PCM locks which cover the data it updates. One instance disowns the PCM locks only if the other instance needs access to the same data.

By contrast, if you assign one set of PCM locks to datafiles 3 and 4, I/O will increase. This happens because both instances regularly use the same set of PCM locks.

Setting GC_FILES_TO_LOCKS: PCM Locks for Each Datafile

Set the GC_FILES_TO_LOCKS initialization parameter to specify the number of PCM locks which will cover the data blocks in a datafile or set of datafiles. This section covers:

- GC_FILES_TO_LOCKS Syntax
- Fixed Lock Examples
- Releasable Lock Example
- Guidelines

Note: Whenever you add or resize a datafile, create a tablespace, or drop a tablespace and its datafiles, you should adjust the value of GC_FILES_TO_LOCKS before restarting Oracle with the parallel server enabled.

See Also: Chapter 9, “Parallel Cache Management Instance Locks”, to understand how the number of data blocks covered by a single PCM lock is determined.

GC_FILES_TO_LOCKS Syntax

The syntax for setting the GC_FILES_TO_LOCKS parameter specifies the translation between the database address and class of a database block, and the lock name which will protect it. You cannot specify this translation for files which are not mentioned in the GC_FILES_TO_LOCKS parameter.

The syntax for setting this parameter is:

GC_FILES_TO_LOCKS="{*file_list=#locks*[!*blocks*][R][EACH][:]} . . ."

where

file_list	*file_list* specifies a single file, range of files, or list of files and ranges as follows: *fileidA*[-*fileidC*][,*fileidE*[-*fileidG*]] ... Query the data dictionary view DBA_DATA_FILES to find the correspondence between file names and file ID numbers.
#locks	Sets the number of PCM locks to assign to file_list. A value of zero (0) for *#locks* means that fine grain locks will be used instead of hashed locks.
!*blocks*	Specifies the number of contiguous data blocks to be covered by each lock.
EACH	Specifies *#locks* as the number of locks to be allocated to *each* file in *file_list*.
R	Specifies that the hashed locks are releasable: they may be released by the instance when they are no longer needed. Releasable hashed PCM locks are taken from the pool GC_RELEASABLE_LOCKS.

Note: GC_ROLLBACK_LOCKS uses the same syntax.

Spaces are not permitted within the quotation marks of the GC_FILES_TO_LOCKS parameter.

In addition to controlling the mapping of PCM locks to datafiles, GC_FILES_TO_LOCKS now controls the number of locks in the default bucket. The default bucket is used for all files not explicitly mentioned in GC_FILES_TO_LOCKS. A value of zero can be used, and the default is "0=0". For example, "0=100", "0=100R", "0-9=100EACH". By default, the locks in this bucket are releasable; you can however, set these locks to be fixed, if you wish.

You can specify releasable hashed PCM locks by using the R option with the GC_FILES_TO_LOCKS parameter. Releasable hashed PCM locks are taken from the pool of GC_RELEASABLE_LOCKS

REACH is a new keyword (combination of R and EACH). For example, GC_FILES_TO_LOCKS="0-9=100REACH". EACHR is *not* a valid keyword.

Omitting EACH and "!*blocks*" means that *#locks* PCM locks are allocated collectively to *file_list* and individual PCM locks cover data blocks across every file in *file_list*. However, if any datafile contains fewer data blocks than the number of PCM locks, some PCM locks will not cover a data block in that datafile.

The default value for *!blocks* is 1. When specified, *blocks* contiguous data blocks are covered by each one of the *#locks* PCM locks. To specify a value for *blocks*, you must use the "!" separator. You would primarily specify *blocks* (and not specify the EACH keyword) to allocate sets of PCM locks to cover multiple datafiles. You can use *blocks* to allocate a set of PCM locks to cover a single datafile where PCM lock contention on that datafile will be minimal, thus reducing PCM lock management.

Always set the *!blocks* value to avoid breaking data partitioning gained by using free list groups. Normally you do not need to pre-allocate disk space. When a row is inserted into a table and new extents need to be allocated, contiguous blocks specified with *!blocks* in GC_FILES_TO_LOCKS are allocated to the free list group associated with an instance.

Fixed Lock Examples

For example, you can assign 300 locks to file 1 and 100 locks to file 2 by adding the following line to the parameter file of an instance:

```
GC_FILES_TO_LOCKS = "1=300:2=100"
```

The following entry specifies a total of 1500 locks—500 each for files 1, 2, and 3:

```
GC_FILES_TO_LOCKS = "1-3=500EACH"
```

By contrast, the following entry specifies a total of only 500 locks, spread across the three files:

```
GC_FILES_TO_LOCKS = "1-3=500"
```

The following entry indicates that 1000 distinct locks should be used to protect file 1. The data in the files is protected in groups of 25 blocks.

```
GC_FILES_TO_LOCKS = "1=1000!25"
```

The following entry indicates that the 1000 hashed locks (which protect file 1 in groups of 25 blocks) may be released by the instance when they are no longer needed.

```
GC_FILES_TO_LOCKS = "1=1000!25R"
```

Releasable Lock Example

To specify fine grain locks for data blocks with a group factor, you can specify the following in the parameter file of an instance:

```
GC_FILES_TO_LOCKS="1=0!4"
```

This specifies fine grain locks with a group factor of 4 for file 1.

Guidelines

Use the following guidelines to set the GC_FILES_TO_LOCKS parameter:

- Always specify all datafiles in GC_FILES_TO_LOCKS.
- Assign the same value to GC_FILES_TO_LOCKS for each instance accessing the same database.
- The number of PCM locks specified for a datafile should never exceed the number of blocks in the datafile. This ensures that if a datafile increases in size, the new blocks can be covered by the extra PCM locks.

  If a datafile is defined with the AUTOEXTEND clause or you issue the ALTER DATABASE ... DATAFILE ... RESIZE command, then you should regularly monitor the datafile for an increase in size. If this occurs, then you should update the parameter GC_FILES_TO_LOCKS as soon as possible, then shut down and restart your parallel server.

  Note: Restarting the parallel server is not required; but if you do not shut down and restart, the locks will cover more blocks.

  If the number of PCM locks specified for *file_list* is less than the actual number of data blocks in the datafiles, then the IDLM uses some PCM locks to cover more datablocks than specified. This can hurt performance, so you should always ensure that sufficient PCM locks are available.
- When you add new datafiles, always specify their locks in GC_FILES_TO_LOCKS to avoid automatic allocation of the "spare" PCM locks.

  At some point you may need to add a datafile (via ALTER TABLESPACE ... ADD DATAFILE) while the parallel server is running. If you do this, then you

should update GC_FILES_TO_LOCKS as soon as possible, then shut down and restart your parallel server.

- When you want to reduce resource contention by creating disjoint data to be accessed by different instances, you should place datafiles on different disks. Use GC_FILES_TO_LOCKS to allocate PCM locks to cover the data blocks in the separate datafiles.
- Specify relatively fewer PCM locks for blocks containing index data which is modified infrequently. Place indexes in their own tablespace or in their own datafiles within a tablespace so that a separate set of PCM locks can be assigned to them. For a read-only index, only one PCM lock is needed.
- Note that files not mentioned in GC_FILES_TO_LOCKS use DBA fine-grained locking.

Tips for Setting GC_FILES_TO_LOCKS

Setting GC_FILES_TO_LOCKS is an important tuning task in the Oracle Parallel Server environment. This section covers some simple checks you can perform to help ensure that your parameter settings are on the mark.

- Providing Room for Growth
- Checking for Valid Number of Locks
- Checking for Valid Lock Assignments
- Setting Tablespaces to Read-only
- Checking File Validity
- Adding Datafiles Without Changing Parameter Values

Providing Room for Growth

Sites which must run nonstop cannot afford to shut down in order to adjust parameter values. Therefore, when you size these parameters, remember to provide room for growth: room for files to extend.

Additionally, whenever you add or resize a datafile, create a tablespace, or drop a tablespace and its datafiles, you should adjust the value of GC_FILES_TO_LOCKS before restarting Oracle with parallel server enabled.

Checking for Valid Number of Locks

It is wise to check that the number of locks allocated is not larger than the number of data blocks allocated. (Note that blocks currently allocated may be zero if you are about to insert into a table.)

Check the FILE_LOCK data dictionary view to see the number of locks which are allocated per file. Check V$DATAFILE to see the maximum size of the data file.

See Also: *Oracle8 Reference* for more information about FILE_LOCK and V$DATAFILE.

Checking for Valid Lock Assignments

To avoid problems with lock assignments, check the following:

- Do not assign locks to files which only hold rollback segments.
- Do not assign locks to files which only hold temporary data for internal temporary tables.
- Group read-only objects together and assign one lock only to that file.

  Note: This will only work and/or perform if there is absolutely no write to the file (there is no change to the blocks, such as block clean out, and so on).

Setting Tablespaces to Read-only

If a tablespace is actually read-only, consider setting it to read-only in Oracle. This ensures that no write to the database will occur and no PCM locks will be used (except for a single lock you can assign, to ensure that the tablespace will not have to contend for spare locks).

Checking File Validity

Count the number of objects in each file, as follows:

```
  SELECT E.FILE_ID      FILE_ID,
         COUNT(DISTINCT OWNER||NAME ) OBJS
    FROM DBA_EXTENTS     E,
         EXT_TO_OBJ V
   WHERE E.FILE_ID = FILE#
     AND E.BLOCK_ID >= LOWB
     AND E.BLOCK_ID <= HIGHB
     AND KIND != 'FREE EXTENT'
     AND KIND != 'UNDO'
GROUP BY E.FILE_ID;
```

Examine the files which store multiple objects. Run CATPARR.SQL to use the EXT_TO_OBJ view. Make sure that they can coexist in the same file (that is, make sure the GC_FILES_TO_LOCKS settings are compatible).

Adding Datafiles Without Changing Parameter Values

Consider the consequences for PCM lock distribution if you add a datafile to the database. You cannot assign locks to this file without shutting down, changing the GC_FILES_TO_LOCKS parameter, and restarting the database. This may not be possible for a production database.

In this case, the datafile will be assigned to the pool of remaining locks and the file must contend with all the files which were not mentioned in the GC_FILES_TO_LOCKS parameter.

Setting Other GC_* Parameters

This section describes how to set two additional GC_* parameters:

- Setting GC_RELEASABLE_ LOCKS
- Setting GC_ROLLBACK_ LOCKS

Setting GC_RELEASABLE_ LOCKS

For GC_RELEASABLE_LOCKS Oracle recommends the default setting, which is the value of DB_BLOCK_BUFFERS. This recommendation holds although GC_RELEASABLE_LOCKS can be set to less than the default to save memory, or more than the default to get a possible reduction in locking activity. Too low a value for GC_RELEASABLE_LOCKS could affect performance.

The statistic “releasable freelist waits” in the V$SYSSTAT view tracks the number of times the system runs out of releasable locks. If this condition should occur, as indicated by a non-zero value for releasable freelist waits, you must increase the value of GC_RELEASABLE_LOCKS.

Setting GC_ROLLBACK_ LOCKS

If you are using fixed locks, it is wise to check that the number of locks allocated is not larger than the number of data blocks allocated. (Note that blocks currently allocated may be zero if you are about to insert into a table.) Find the number of blocks allocated to a rollback segment by entering:

```
  SELECT S.SEGMENT_NAME NAME,
         SUM(RBLOCKS) BLOCKS
    FROM DBA_SEGMENTS S,
         DBA_EXTENTS R
   WHERE S.SEGMENT_TYPE = 'ROLLBACK'
     AND S.SEGMENT_NAME = R.SEGMENT_NAME
GROUP BY S.SEGMENT_NAME;
```

This query displays the number of blocks allocated to each rollback segment. When there are many unnecessary forced reads / writes on the undo blocks, try using releasable locks. (By default all rollback segments are protected by releasable locks.)

The parameter GC_ROLLBACK_LOCKS takes arguments much like the GC_FILES_TO_LOCKS parameter, for example:

GC_ROLLBACK_LOCKS=“0=100:1-10=10EACH:11-20=20EACH”

In this example, rollback segment 0 (the system rollback segment) has 100 locks; rollback segments 1 through 10 have 10 locks each; and rollback segments 11 through 20 have 20 locks each.

Note: GC_ROLLBACK_LOCKS cannot be used to make undo segments share locks. The first example below is invalid, but the second is valid, since each of the undo segments has 100 locks to itself:

Invalid: GC_ROLLBACK_LOCKS=“1-10=100”

Valid: GC_ROLLBACK_LOCKS=“1-10=100EACH”

Tuning Your PCM Locks

This section discusses several issues you must consider before tuning your PCM locks:

- How to Detect False Pinging
- How Long Does a PCM Lock Conversion Take?
- Which Sessions Are Waiting for PCM Lock Conversions to Complete?
- What Is the Total Number of PCM Locks and Resources Needed?

How to Detect False Pinging

False pinging occurs when you down-convert a lock element which protects two or more blocks that are concurrently updated from different nodes. Assume that each node is updating a different block. In this event, each node must write its own copy of the block, even though the other node is not updating it. This is necessary because the same lock covers both blocks.

There are no direct statistics of false pinging—only indications which you can interpret. This section describes some indications you can watch out for.

The following SQL statement shows the number of lock operations which caused a write, and the number of blocks actually written:

```
SELECT VALUE/(A.COUNTER + B.COUNTER + C.COUNTER) "PING RATE"
  FROM V$SYSSTAT,
       V$LOCK_ACTIVITY A,
       V$LOCK_ACTIVITY B,
       V$LOCK_ACTIVITY C
 WHERE A.FROM_VAL = 'X'
   AND A.TO_VAL = 'NULL'
   AND B.FROM_VAL = 'X'
   AND B.TO_VAL = 'S'
   AND C.FROM_VAL = 'X'
   AND C.TO_VAL = 'SSX'
   AND NAME = 'DBWR cross instance writes';
```

Table 15–1 shows how to interpret the ping rate.

Table 15–1 Interpreting the Ping Rate

Ping Rate	Meaning
< 1	False pings may be occurring, but there are more lock operations than writes for pings. DBWR is writing out blocks fast enough, causing no write for a lock activity. This is also known as a "soft ping" (no I/O is required for the ping, only lock activity).
= 1	Each lock activity which involves a potential write, does indeed cause the write to happen. False pinging may be occurring.
> 1	False pings are definitely occurring.

Use this formula to calculate the percentage of pings that are definitely false:

$$\frac{(ping\_rate - 1)}{ping\_rate} * 100$$

Then check the total number of writes and calculate the number due to false pings:

```
SELECT Y.VALUE "ALL WRITES",
       Z.VALUE "PING WRITES",
       Z.VALUE * pingrate "FALSE PINGS",
  FROM V$SYSSTAT Z,
       V$SYSSTAT Y,
 WHERE Z.NAME = 'DBWR cross instance writes'
   AND Y.NAME = 'physical writes';
```

Here, *ping_rate* is given by the following SQL statement:

```
CREATE OR REPLACE VIEW PING_RATE AS
SELECT ((VALUE/(A.COUNTER+B.COUNTER+C.COUNTER))-1)/
       (VALUE/(A.COUNTER+B.COUNTER+C.COUNTER)) RATE
  FROM V$SYSSTAT,
       V$LOCK_ACTIVITY A,
       V$LOCK_ACTIVITY B,
       V$LOCK_ACTIVITY C
 WHERE A.FROM_VAL = 'X'
   AND A.TO_VAL   = 'NULL'
   AND B.FROM_VAL = 'X'
   AND B.TO_VAL   = 'S'
   AND C.FROM_VAL = 'X'
   AND C.TO_VAL   = 'SSX'
   AND NAME = 'DBWR cross instance writes';
```

Needless to say, the goal is not only to reduce overall pinging, but also to reduce false pinging. To do this, look at the distribution of instance locks in GC_FILES_TO_LOCKS, and check the data in the files.

How Long Does a PCM Lock Conversion Take?

Be sure to check the amount of time needed for a PCM lock to convert. This time differs between systems. Enter the following SQL statement to find lock conversion time:

```
SELECT *
  FROM V$SYSTEM_EVENT
 WHERE EVENT = 'lock element cleanup'
```

This SQL statement displays a table like the following:

```
                     TOTAL_    TOTAL_      TIME_     AVERAGE_
EVENT                WAITS     TIMEOUTS    WAITED    WAIT
-------------------  ------    -------     ------    ----------
lock element cleanup 32709     44          685660    20.9624262
```

This means that a lock conversion took 20.9 hundredths of a second (0.209 seconds).

Which Sessions Are Waiting for PCM Lock Conversions to Complete?

Enter the following SQL statement to see which sessions are currently waiting, and which have just waited for a PCM lock conversion to complete:

```
SELECT *
  FROM V$SESSION_WAIT
 WHERE EVENT = 'lock element cleanup'
```

What Is the Total Number of PCM Locks and Resources Needed?

This section explains how you can determine the number of PCM locks and resources your system requires. This is the value you need to set for the LM_LOCKS and LM_RESS parameters.

Formula for PCM Locks and Resources

To find this value, add the number of *fixed (non-releasable)* locks set *per instance* (the sum of GC_FILES_TO_LOCKS and GC_ROLLBACK_LOCKS—fixed locks only) to the total number of *releasable* locks (the value of GC_RELEASABLE_LOCKS), and multiply by two.

2 * (GC_FILES_TO_LOCKS + GC_ROLLBACK_LOCKS*fixed* + GC_RELEASABLE_LOCKS)

This figure represents the maximum number of PCM locks and resources your system requires. Note that this calculation is independent of the number of instances.

Bear in mind the following considerations:

- GC_FILES_TO_LOCKS: default value is releasable; all instances must have the same setting.
- GC_ROLLBACK_LOCKS: default is releasable; all instances must have the same setting.
- GC_RELEASABLE_LOCKS: default is releasable, set to value of DB_BLOCK_BUFFERS.

Calculating PCM Locks and Resources: Example

For example, assume that your system has the following settings *for each instance*:

GC_FILES_TO_LOCKS="1=100:2-5=1000:6-10=1000EACH:11=100R"

GC_ROLLBACK_LOCKS="1-10=10EACH:11-20=20EACH"

GC_RELEASABLE_LOCKS=50,000

You add the GC_FILES_TO_LOCKS values as follows: File 1 has 100 fixed locks. files 2, 3, 4, and 5 share 1000 locks. File 6 has 1000 fixed locks, file 7 has 1000 fixed locks, file 8 has 1000 fixed locks, file 9 has 1000 fixed locks, file 10 has 1000 fixed locks. File 11 has no fixed locks. Hence there is a total of 6,100 fixed locks set by GC_FILES_TO_LOCKS.

You add the GC_ROLLBACK_LOCKS values as follows: Files 1 through 10 have 10 fixed locks each, and Files 11 through 20 have 20 fixed locks each, for a total of 300 fixed locks.

Entering these figures into the formula, you would calculate as follows:

2 * (6,100 + 300 + 50,000) = 112,800

You would thus set the LM_LOCKS and LM_RESS parameters to 112,800.

See Also: "GC_FILES_TO_LOCKS Syntax" on page 15-8

16

Ensuring IDLM Capacity for All Resources & Locks

To reduce contention for shared resources and gain maximum performance from the parallel server, you must ensure that the Integrated Distributed Lock Manager (Integrated DLM, or IDLM) is adequately configured for all the locks and resources your system requires. This chapter covers the following topics:

- Overview
- Planning IDLM Capacity
- Calculating the Number of Non-PCM Resources
- Calculating the Number of Non-PCM Locks
- Adjusting Oracle Initialization Parameters
- Minimizing Table Locks to Optimize Performance

See Also: Chapter 10, “Non-PCM Instance Locks”, for a conceptual overview.

Overview

Planning PCM locks alone is *not* sufficient to manage locks on your system. Besides explicitly allocating parallel cache management locks, you must actively ensure that the Integrated DLM is adequately configured, on each node, for all the required PCM and non-PCM locks and resources. Bear in mind that larger databases and higher degree of parallelism require increased demand for many resources.

Many different types of non-PCM lock exist, and each is handled differently. Although you cannot directly adjust their number, you can estimate the overall number of non-PCM resources and locks required, and adjust the LM_* or GC_* initialization parameters (or both) to guarantee adequate space. You also have the option of minimizing table locks to optimize performance.

Planning IDLM Capacity

Carefully plan and configure an appropriate number of resources and locks to be managed by the Integrated DLM. You allocate these locks and resources using the initialization parameters LM_LOCKS and LM_RESS. Although additional locks and resources can be allocated dynamically, this should be avoided.

Avoiding Dynamic Allocation of Resources and Locks

If the number of locks or resources required becomes greater than the amount you have statically allocated, then the additional locks or resources will be allocated from the SGA shared pool. This feature prevents the instance from stopping.

When dynamic allocation occurs, a message is written to the alert file indicating that you should recompute and adjust the initialization parameters for the next time the database is started. Since performance and memory usage may be adversely affected by dynamic allocation, it is highly recommended that you correctly compute your lock and resource needs.

Computing Lock and Resource Needs

Use the following approach to carefully plan IDLM capacity, on a per node basis, for the total number of PCM and non-PCM resources and locks needed.

1. Consider failover requirements.

   In case of failover, you need to have enough resources configured on the remaining instances so that the system can continue to operate. Thus if resources are divided over 10 instances and 5 instances were to fail, you would still want the system to be able to run on the remaining 5 instances. This means that you should allow some leeway in the system by accounting for overhead and setting large enough values for the Oracle initialization parameters determining IDLM locks and resources for each instance.

2. Consider the sizing of each instance on each node: number of users, volume of transactions, and so on. Determine the values you will assign to each instance's initialization parameters.

3. Calculate the number of non-PCM resources and locks required, by filling in the worksheets provided in this chapter.

4. Calculate the number of PCM resources and locks required, by using the script in "What Is the Total Number of PCM Locks and Resources Needed?" on page 15-19.

5. Configure the IDLM to accommodate the required number of:

   - non-PCM resources
   - non-PCM locks
   - PCM resources
   - PCM locks

Monitoring Resource Utilization

The V$RESOURCE_LIMIT view provides information about global resource utilization for some of the system resources. Using this view you can monitor the current and the maximum resource utilization, and be forewarned if the values approach the limit. With this information you can make better decisions when choosing values for resource limit-controlling parameters.

See Also: "Setting LM_* Parameters" on page 18-11
Oracle8 Reference regarding V$RESOURCE_LIMIT
Oracle8 Tuning for a complete discussion of resource limits

Calculating the Number of Non-PCM Resources

Use the following worksheet to analyze your system resources.

1. In the following worksheet, enter values for the PROCESSES, DML_LOCKS, TRANSACTIONS, and ENQUEUE_RESOURCES initialization parameters for each instance.
2. *For each instance,* enter the value of the DB_FILES parameter, which is the same for all instances.
3. Enter values for Enqueue Locks for each instance. For each instance, you can calculate this value as follows:

   Enqueue Locks = 20 + (10 * SESSIONS) + DB_FILES + GC_LCK_PROCS + (2 * PROCESSES) + (DB_BLOCK_BUFFERS / 64)

4. For each instance, enter values for Parallel Query Overhead to cover inter-instance communication. For individual instances, you can calculate this value as follows:

   PQ Overhead = 7 + (MAXINSTANCES * PARALLEL_MAX_SERVERS) + PARALLEL_MAX_SERVERS + MAXINSTANCES

5. Add the entries horizontally to obtain the Subtotals: # of Non-PCM Resources per Instance.
6. Add the per-instance subtotals to obtain the Total Number of Non-PCM Resources System-Wide.

Table 16–1 Worksheet: Calculating Non-PCM Resources

Inst. No.	PRO-CESSES	DML_ LOCKS	TRANS-ACTIONS	ENQUEUE_ RESOURCES	DB_FILES (on one or more instances)	Enqueue Locks	PQ Over-head	Over-head	Subtotals: # NonPCM Locks per Instance
1								200	
2								200	
3								200	
4								200	
Total Number of Non-PCM Locks System-Wide:									

7. Finally, use the figures derived from this worksheet to ensure that LM_RESS is set to accommodate all non-PCM resources (see step 6 on page 16 - 4).

   Note: The worksheet incorporates a standard overhead value of 200 for each instance.

Table 16–2 shows sample values for a system with four instances, and with PARALLEL_MAX_SERVERS set to 8 for instances 1 and 3, and set to 4 for instances 2 and 4. The buffer cache size is assumed to be 10K.

Table 16–2 Calculating Non-PCM Resources (Example)

Inst. No.	PRO-CESSES	DML_ LOCKS	TRANS-ACTIONS	ENQUEUE_ RESOURCES	DB_FILES (on one or more instances)	Enqueue Locks	PQ Over-head	Over-head	Subtotals: # Non-PCM Locks per Instance
1	200	500	50	800	30	2808	51	200	4639
2	350	600	100	1000	--	4128	31	200	6409
3	175	400	75	800	--	2453	51	200	4154
4	225	350	125	1200	--	3103	31	200	5234
Total Number of Non-PCM Locks System-Wide:									20436

Calculating the Number of Non-PCM Locks

Use the following worksheet to analyze your system's lock needs.

1. In the following worksheet, enter values for the PROCESSES, DML_LOCKS, TRANSACTIONS, and ENQUEUE_RESOURCES parameters for each instance.
2. *For each instance*, enter the value of the DB_FILES parameter, which is the same for all instances.
3. Enter values for Enqueue Locks for each instance. For each instance, you can calculate this value as follows:

   Enqueue Locks = 20 + (10 * SESSIONS) + DB_FILES + GC_LCK_PROCS + (2 * PROCESSES) + (DB_BLOCK_BUFFERS / 64)

4. For each instance, enter values for Parallel Query Overhead to cover inter-instance communication. For individual instances, you can calculate this value as follows:

   PQ Overhead = 7 + (MAXINSTANCES * PARALLEL_MAX_SERVERS) + PARALLEL_MAX_SERVERS + MAXINSTANCES

5. Add the entries horizontally to obtain the Subtotals: # of Non-PCM Locks per Instance.
6. Add the per-instance Subtotals to obtain the Total Number of Non-PCM Locks System-Wide.

Table 16–3 Worksheet: Calculating Non-PCM Locks

Inst. No.	PRO-CESSES	DML_ LOCKS	TRANS-ACTIONS	ENQUEUE_ RESOURCES	DB_FILES (for all instances)	Enqueue Locks	PQ Over-head	Over-head	Subtotals: # of Non-PCM Locks per Instance
1								200	
2								200	
3								200	
4								200	
Total Number of Non-PCM Locks System-Wide:									

7. Finally, use the figures derived from this worksheet to ensure that the LM_LOCKS parameter is configured to accommodate all non-PCM locks (see step 6 on page 16 - 4).

Note: The worksheet incorporates a standard overhead value of 200 for each instance.

Table 16–4 shows sample values for a system with four instances, again assuming that PARALLEL_MAX_SERVERS is set to 8 for instances 1 and 3, and set to 4 for instances 2 and 4. The buffer cache size is assumed to be 10K.

Table 16–4 Calculating Non-PCM Locks (Example)

Inst. No.	PRO-CESSES	DML_ LOCKS	TRANS-ACTIONS	ENQUEUE_ RESOURCES	DB_FILES (for all instances)	Enqueue Locks	PQ - Over-head	Over-head	Subtotals: # of Non-PCM Locks per Instance
1	200	500	50	800	30	2808	51	200	4639
2	350	600	100	1000	30	4128	31	200	6439
3	175	400	75	800	30	2453	51	200	4184
4	225	350	125	1200	30	3103	31	200	5264
Total Number of Non-PCM Locks System-Wide:									20526

Adjusting Oracle Initialization Parameters

Another way to ensure that your system has enough space for the required non-PCM locks and resources is to adjust the values of the following Oracle initialization parameters:

DB_BLOCK_BUFFERS
DB_FILES
DML_LOCKS
GC_LCK_PROCS
PARALLEL_MAX_SERVERS
PROCESSES
SESSIONS
TRANSACTIONS

Begin by experimenting with these values *in the worksheets* supplied in this chapter. You could artificially inflate parameter values in the worksheets, in order to see the IDLM ramifications of providing extra room for failover.

Do not, however, specify actual parameter values considerably greater than needed for each instance. Setting these parameters unnecessarily high entails overhead in a parallel server environment.

Minimizing Table Locks to Optimize Performance

This section describes two strategies for improving performance by minimizing table locks:

- Setting DML_LOCKS to Zero
- Disabling Table Locks

Obtaining table locks (DML locks) for inserts, deletes, and updates can hurt performance in a parallel server environment. Locking a table in a parallel server is very undesirable because all instances holding locks on the table must release those locks. Consider disabling these locks entirely.

Note: If you use either of these strategies you cannot perform DDL commands against either the instance or the table.

Setting DML_LOCKS to Zero

Table locks are set with the initialization parameter DML_LOCKS. If the DROP TABLE, CREATE INDEX, and LOCK TABLE commands are not needed, set DML_LOCKS to zero in order to minimize lock conversions and gain maximum performance.

Note: If DML_LOCKS is set to zero on one instance, it must be set to zero on *all* instances. With other values, this parameter need *not* be identical on all instances.

Disabling Table Locks

To prevent any user from acquiring a table lock, you can use the following command:

```
ALTER TABLE table_name DISABLE TABLE LOCK
```

Any user attempting to lock a table when its table lock is disabled will receive an error.

To re-enable table locking, the following command is used:

```
ALTER TABLE table_name ENABLE TABLE LOCK
```

The above command waits until all currently executing transactions commit before enabling the table lock. Note that the command does not need to wait for new transactions which start after the enable command was issued.

To determine whether a table has its table lock enabled or disabled, you can query the column TABLE_LOCK in the data dictionary table USER_TABLES. If you have select privilege on DBA_TABLES or ALL_TABLES, you can query the table lock state of other users tables.

17

Using Free List Groups to Partition Data

This chapter explains how to allocate free lists and free list groups to partition data. By doing this you can minimize contention for free space when using multiple instances.

The chapter describes:

- Overview
- Deciding How to Partition Free Space for Database Objects
- Setting FREELISTS and FREELIST GROUPS in the CREATE Statement
- Associating Instances, Users, and Locks with Free List Groups
- Pre-allocating Extents (Optional)
- Dynamically Allocating Extents
- Identifying and Deallocating Unused Space

See Also: Chapter 11, “Space Management and Free List Groups”, for a conceptual overview.

Overview

Use the following procedure to manage free space for multiple instances.

1. Analyze database objects and decide how to partition free space and data.
2. Set FREELISTS and FREELIST GROUPS in the CREATE statement for each table, cluster, and index.
3. Associate instances, users, and locks with free lists.
4. Allocate blocks to free lists.
5. Pre-allocate extents, if desired.

By managing free space effectively, you may improve performance of an application configuration which is not ideally suited to a parallel server.

Attention: For optimal system performance, use care in setting the FREELIST and FREELIST GROUPS options; these values cannot be reset.

Deciding How to Partition Free Space for Database Objects

This section provides a worksheet to help you analyze database objects and decide how to partition free space and data for optimal performance.

- Database Object Characteristics
- Free Space Worksheet

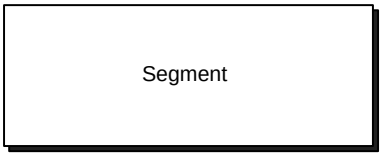
Database Object Characteristics

Analyze the database objects you will create and sort them into the categories described in this section.

Objects in a Static Table

If a table does not have high insert activity, it does not need free lists or free list groups.

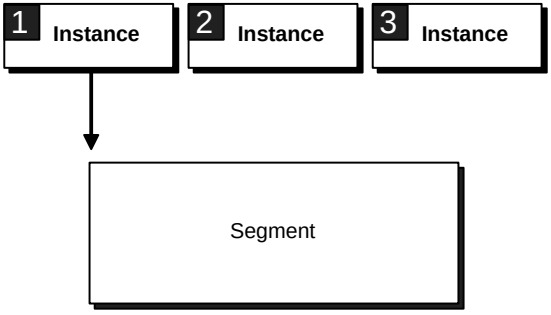
Figure 17–1 Database Objects in a Static Table



Objects in a Partitioned Application

With proper partitioning of certain applications, only one node needs to insert into the table or segment. In such cases, free lists may be necessary if there are a large number of users, but free list groups are not necessary.

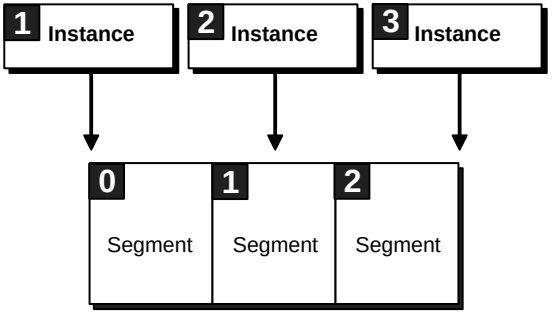
Figure 17–2 Database Objects in a Partitioned Application



Objects Relating to Partitioned Data

Multiple free lists and free list groups are not necessary for objects with partitioned data.

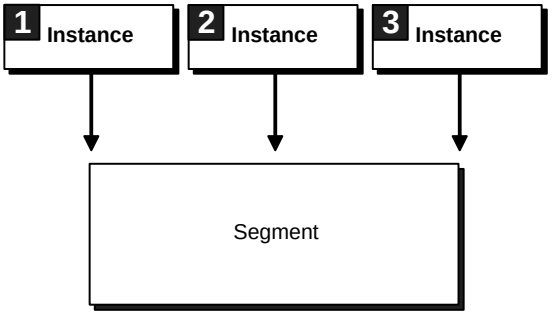
Figure 17–3 Database Objects Relating to Partitioned Data



Objects in a Table with Random Inserts

Free lists and free list groups are needed when random inserts from multiple instances occur in a table. All instances writing to the segment must check the master free list to determine where to write. There would thus be contention for the segment header, which contains the master free list.

Figure 17–4 Database Objects in a Table with Random Inserts



Free Space Worksheet

List each of your database objects (tables, clusters, and indexes) in a worksheet like the following, and plan free lists and free list groups for each.

Table 17–1 Free Space Worksheet for Database Objects

Database Object Characteristics	Free List Groups	Free Lists
Objects in Static Tables	NA	NA
	NA	NA
	NA	NA
	NA	NA
Objects in Partitioned Applications	NA	
	NA	
	NA	
	NA	
Objects Related to Partitioned Data	NA	NA
	NA	NA
	NA	NA
	NA	NA
Objects in Table w/Random Inserts		

Note: Do not confuse partitioned data here with Oracle8 partitions, which may or may not be in use.

Setting FREELISTS and FREELIST GROUPS in the CREATE Statement

This section covers the following topics:

- FREELISTS Option
- FREELIST GROUPS Option
- Creating Free Lists for Clusters
- Creating Free Lists for Indexes

You can create free lists and free list groups by specifying the FREELISTS and FREELIST GROUPS storage options in CREATE TABLE, CLUSTER or INDEX statements. This can be done while accessing the database in either exclusive mode or shared mode.

Attention: Once you have set these storage options you *cannot* change their values with the ALTER TABLE, CLUSTER, or INDEX statements.

See Also: The STORAGE clause in *Oracle8 SQL Reference* for the syntax of these options.

FREELISTS Option

FREELISTS specifies the number of free lists in each free list group. The default value of FREELISTS is 1, which is also the minimum value; the maximum depends on the data block size. If you specify a value that is too large, an error message informs you of the maximum value. The optimal value of FREELISTS depends on the expected number of concurrent inserts per free list group for this table.

FREELIST GROUPS Option

Each free list group is associated with one or more instances at startup. The default value of FREELIST GROUPS is 1, which means that the table's free lists (if any) are available to all instances. Typically, you should set FREELIST GROUPS to the number of instances in the parallel server. Using free list groups also partitions data. Blocks allocated to one instance, freed by another instance, are no longer available to the first instance.

Note: Even in a non-shared environment, multiple free list groups can benefit performance. With multiple free list groups, the free list structure is delinked from the segment header, thereby reducing contention for the segment header. This is very useful when there is a high volume of UPDATE and INSERT transactions.

Example The following statement creates a table named DEPT that has seven free list groups, each of which contains four free lists:

```
CREATE TABLE dept
            (deptno   NUMBER(2),
             dname    VARCHAR2(14),
             loc      VARCHAR2(13) )
            STORAGE ( INITIAL 100K        NEXT 50K
                      MAXEXTENTS 10       PCTINCREASE 5
                      FREELIST GROUPS 7   FREELISTS 4 );
```

Creating Free Lists for Clusters

You cannot specify the FREELISTS and FREELIST GROUPS storage options in the CREATE TABLE statement for a clustered table. The free list options must be specified for the *whole* cluster, rather than for individual tables. This is because the tables in a cluster use the storage parameters of the CREATE CLUSTER statement.

Clusters are an optional method of storing data in groups of tables that have columns in common. Related rows of two or more tables in a cluster are physically stored together within the database to improve access time. A parallel server allows clusters (other than hash clusters) to use multiple free lists and free list groups.

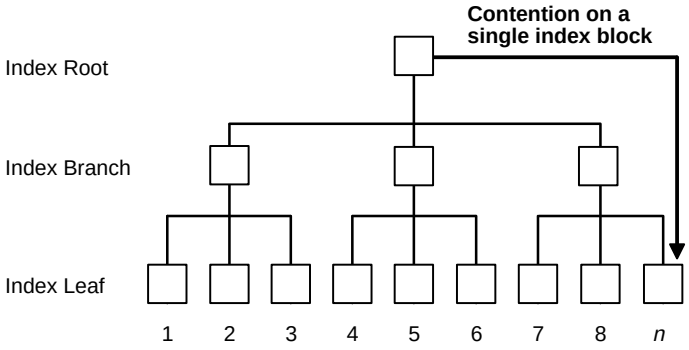
Some hash clusters can also use multiple free lists and free list groups, if they are created with a user-defined key for the hashing function, and the key is partitioned by instance.

Creating Free Lists for Indexes

You can use the FREELISTS and FREELIST GROUPS storage options of the CREATE INDEX statement to create multiple free space lists for concurrent user processes. Use these options in the same manner as described for tables.

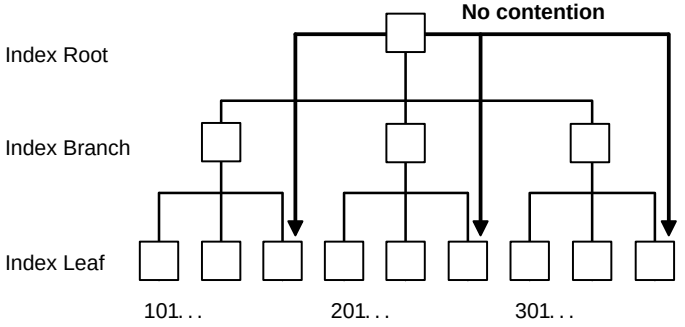
When multiple instances concurrently insert rows into a table having an index, contention for index blocks decreases performance unless index values can be separated by instance. Figure 17–5 illustrates a situation in which all instances are trying to insert into the same leaf block (*n*) of an index.

Figure 17–5 Contention for One Index Block



To avoid this problem, have each instance insert into its own tree, as illustrated in Figure 17–6.

Figure 17–6 No Index Contention



Compute the index value with an algorithm such as

instance_number * (100000000) + *sequence_number*

Associating Instances, Users, and Locks with Free List Groups

This section explains how you can associate the following with free list groups:

- Associating Instances with Free List Groups
- Associating User Processes with Free List Groups
- Associating PCM Locks with Free List Groups

Associating Instances with Free List Groups

You can associate an instance with extents or free list groups as follows:

INSTANCE_NUMBER parameter	You can use various SQL options with the INSTANCE_NUMBER initialization parameter to associate extents of data blocks with instances.
SET INSTANCE option	You can use the SET INSTANCE option of the ALTER SESSION command to ensure that a session uses the free list group associated with a particular instance, regardless of the instance to which the session is connected. For example: ALTER SESSION SET INSTANCE = *inst_no*

The SET INSTANCE feature is useful when an instance fails and a user connects to another instance. For example, consider a database where space is pre-allocated to the free list groups in a table and users are distributed across instances such that the data is well partitioned, with the result that minimal pinging of data blocks occurs. If an instance fails, moving all users onto other instances does not need to ruin the data partitioning because each new session can use the original free list group associated with the failed instance.

Associating User Processes with Free List Groups

User processes are automatically associated with free lists based on the Oracle process ID of the process in which they are running, as follows:

(*oracle_pid* modulo *#free_lists_for_object*) + 1

You can use the ALTER SESSION SET INSTANCE statement if you wish to use the free list group associated with a particular instance.

Associating PCM Locks with Free List Groups

If each extent in the table is in a separate datafile, you can use the GC_FILES_TO_LOCKS parameter to allocate specific ranges of PCM locks to each extent, so that each set of PCM locks is associated with only one group of free lists.

See Also: "Free Lists Associated with Instances, Users, and Locks" on page 11-14.

Pre-allocating Extents (Optional)

This section explains how to pre-allocate extents. Note, however, that this useful but static approach requires a certain amount of database administration overhead.

- The ALLOCATE EXTENT Option
- Setting MAXEXTENTS, MINEXTENTS, and INITIAL Parameters
- Setting the INSTANCE_NUMBER Parameter
- Examples of Extent Pre-allocation

The ALLOCATE EXTENT Option

The ALLOCATE EXTENT option of the ALTER TABLE or ALTER CLUSTER statement enables you to pre-allocate an extent to a table, index or cluster with options to specify the extent size, datafile, and a group of free lists.

The syntax of the ALLOCATE EXTENT option is given in the descriptions of the ALTER TABLE and ALTER CLUSTER statements in *Oracle8 SQL Reference*.

Exclusive and Shared Modes. You can use the ALTER TABLE (or CLUSTER) ALLOCATE EXTENT statement while the database is running in exclusive mode, as well as in shared mode. When an instance is running in exclusive mode, it still follows the same rules for locating space. A transaction can use the master free list or the specific free list group for that instance.

The SIZE Option. This option of the ALLOCATE EXTENT clause is the extent size in bytes, rounded up to a multiple of the block size. If you do not specify SIZE, the extent size is calculated according to the values of storage parameters NEXT and PCTINCREASE.

The value of SIZE is *not* used as a basis for calculating subsequent extent allocations, which are determined by NEXT and PCTINCREASE.

The DATAFILE Option. This option specifies the datafile from which to take space for the extent. If you omit this option, space is allocated from any accessible datafile in the tablespace containing the table.

Note that the filename must exactly match the string stored in the control file, even with respect to the case of letters. You can check the DBA_DATA_FILES data dictionary view for this string.

The INSTANCE Option. This option assigns the new space to the free list group associated with instance number *integer*. Each instance acquires a unique instance number at startup that maps it to a group of free lists. The lowest instance number is 1, not 0; the maximum value is operating system specific. The syntax is as follows:

ALTER TABLE *tablename* ALLOCATE EXTENT (... INSTANCE *n*)

where *n* will map to the free list group with the same number. If the instance number is greater than the number of free list groups, then it is hashed as follows to determine the free list group to which it should be assigned:

modulo(*n*,*#_freelistgroups*) + 1

If you do not specify the INSTANCE option, the new space is assigned to the table but not allocated to any group of free lists. Such space is included in the master free list of free blocks as needed when no other space is available.

Note: Use a value for INSTANCE which corresponds to the number of the free list group you wish to use—rather than the actual instance number.

See Also: "Instance Numbers and Startup Sequence" on page 18-14.

Setting MAXEXTENTS, MINEXTENTS, and INITIAL Parameters

You can prevent automatic allocations by pre-allocating extents to free list groups associated with particular instances, and setting MAXEXTENTS to the current number of extents (pre-allocated extents plus MINEXTENTS). You can minimize the initial allocation when you create the table or cluster by setting MINEXTENTS to 1 (the default) and setting INITIAL to its minimum value (two data blocks, or 10 K for a block size of 2048 bytes).

To minimize contention among instances for data blocks, you can create multiple datafiles for each table and associate each instance with a different file.

If you expect to increase the number of nodes in your loosely coupled system at a future time, you can allow for additional instances by creating tables or clusters with more free list groups than the current number of instances. You do not have to allocate any space to those free list groups until they are needed. Only the master free list of free blocks has space allocated to it automatically.

For a data block to be associated with a free list group, either it must be brought below PCTUSED by a process running on an instance using that free list group or it

must be specifically allocated to that free list group. Therefore, a free list group that is never used does not leave unused free data blocks.

Setting the INSTANCE_NUMBER Parameter

The INSTANCE_NUMBER initialization parameter allows you to start up an instance and ensure that it uses the extents allocated to it for inserts and updates. This will ensure that it does not use space allocated for other instances. The instance cannot use data blocks in another free list unless the instance is restarted with that INSTANCE_NUMBER.

Note that you can also override the instance number during a session by using an ALTER SESSION statement.

Examples of Extent Pre-allocation

This section provides examples in which extents are pre-allocated.

Example 1 The following statement allocates an extent for table DEPT from the data-file DEPT_FILE7 to instance number 7:

```
ALTER TABLE dept
ALLOCATE EXTENT ( SIZE 20K
                  DATAFILE 'dept_file7'
                 INSTANCE 7);
```

Example 2 The following SQL statement creates a table with three free list groups, each containing ten free lists:

```
CREATE TABLE table1 ... STORAGE (FREELIST GROUPS 3 FREELISTS 10);
```

The following SQL statement then allocates new space, dividing the allocated blocks among the free lists in the second free list group:

```
ALTER TABLE table1 ALLOCATE EXTENT (SIZE 50K INSTANCE 2);
```

In a parallel server running more instances than the value of the FREELIST GROUPS storage option, multiple instances share the new space allocation. In this example, every third instance to start up is associated with the same group of free lists.

Example 3 The following CREATE TABLE statement creates a table named EMP with one initial extent and three groups of free lists, and the three ALTER TABLE statements allocate one new extent to each group of free lists:

```
CREATE TABLE emp ...
    STORAGE ( INITIAL 4096
              MINEXTENTS 1
              MAXEXTENTS 4
              FREELIST GROUPS 3 );
ALTER TABLE emp
    ALLOCATE EXTENT ( SIZE 100K DATAFILE 'empfile1' INSTANCE 1 )
    ALLOCATE EXTENT ( SIZE 100K DATAFILE 'empfile2' INSTANCE 2 )
    ALLOCATE EXTENT ( SIZE 100K DATAFILE 'empfile3' INSTANCE 3 );
```

MAXEXTENTS is set to 4, the sum of the values of MINEXTENTS and FREELIST GROUPS, to prevent automatic allocations.

When you need additional space beyond this allocation, use ALTER TABLE to increase MAXEXTENTS before allocating the additional extents. For example, if the second group of free lists requires additional free space for inserts and updates, you could set MAXEXTENTS to 5 and allocate another extent for that free list group:

```
ALTER TABLE emp  ...
    STORAGE ( MAXEXTENTS 5 )
    ALLOCATE EXTENT ( SIZE 100K DATAFILE 'empfile2' INSTANCE 2 );
```

Dynamically Allocating Extents

This section explains how to use the !*blocks* option of GC_FILES_TO_LOCKS to dynamically allocate blocks to a free list from the high water mark within a lock boundary. It covers:

- Translation of Block Database Address to Lock Name
- !blocks with ALLOCATE EXTENT Syntax

Translation of Block Database Address to Lock Name

As described in the "Allocating PCM Instance Locks" chapter, the syntax for setting the GC_FILES_TO_LOCKS parameter specifies the translation between the database address of a block, and the lock name that will protect it. Briefly, the syntax is:

GC_FILES_TO_LOCKS = "{ *file_list=#locks* [!*blocks*] [EACH] [:] } ..."

The following entry indicates that 1000 distinct lock names should be used to protect the files in this bucket. The data in the files is protected *in groups of 25 blocks*.

```
GC_FILES_TO_LOCKS = "1000!25"
```

!*blocks* with ALLOCATE EXTENT Syntax

Similarly, the !*blocks* parameter enables you to control the number of blocks which are available for use within an extent. (To be available, blocks must be put onto a free list.). You can use !*blocks* to specify the rate at which blocks are allocated within an extent, up to 255 blocks at a time. Thus,

```
GC_FILES_TO_LOCKS = 1000!10
```

means that 10 blocks will be made available each time an instance requires the allocation of blocks.

See Also: Chapter 15, "Allocating PCM Instance Locks".

Identifying and Deallocating Unused Space

This section covers:

- How to Determine Unused Space
- Deallocating Unused Space
- Space Freed by Deletions or Updates

How to Determine Unused Space

The DBMS_SPACE package contains procedures by which you can determine the amount of used and unused space in the free list groups in a table. In this way you can determine which instance needs to start allocating space again. The package is created using the DBMSUTIL.SQL script as described in the *Oracle8 Reference*.

Deallocating Unused Space

Unused space you have allocated to an instance using the ALLOCATE EXTENT command *cannot* be deallocated, because it exists below the high water mark.

Unused space *can* be deallocated from the segment, however, if the space exists within an extent that was allocated dynamically above the high water mark. You can use DEALLOCATE UNUSED with ALTER TABLE or ALTER INDEX command in order to trim the segment back to the high water mark.

Space Freed by Deletions or Updates

Blocks freed by deletions or by updates that shrank rows will go to the free list and free list group of the process that deletes them.

Part IV

OPS System Maintenance Procedures

18

Administering Multiple Instances

Justice is a machine that, when someone has once given it the starting push, rolls on of itself.

John Galsworthy: *Justice. Act II.*

This chapter describes how to administer instances of a parallel server. It includes the following topics:

- Overview
- Oracle Parallel Server Management
- Defining Multiple Instances with Parameter Files
- Setting Initialization Parameters for the Parallel Server
- Setting LM_* Parameters
- Creating Database Objects for Multiple Instances
- Starting Up Instances
- Specifying Instances
- Shutting Down Instances
- Limiting Instances for the Parallel Query

Overview

This chapter explains how to set up and then start up instances for a parallel server using the following general procedure:

1. Define multiple instances by setting up parameter files.
2. Set initialization parameter values for multiple instances.
3. Determine the number of PCM and non-PCM locks your system will require.
4. Set LM_* initialization parameters.
5. Create database objects for multiple instances.
6. Start up instances.

See Also: "Starting Up and Shutting Down" in *Oracle8 Administrator's Guide*.

Oracle Parallel Server Management

Note also that Oracle Parallel Server Management (OPSM) is available. This is a comprehensive and integrated system management solution for the Oracle Parallel Server. OPSM allows you to manage multi-instance databases running in heterogeneous environments through an open client-server architecture.

In addition to managing parallel databases, OPSM allows you to schedule jobs, perform event management, monitor performance, and obtain statistics to tune parallel databases.

For more information about OPSM, refer to the *Oracle Parallel Server Management Configuration Guide for UNIX* and the *Oracle Parallel Server Management User's Guide*. For installation instructions, refer to your platform-specific installation guide.

Defining Multiple Instances with Parameter Files

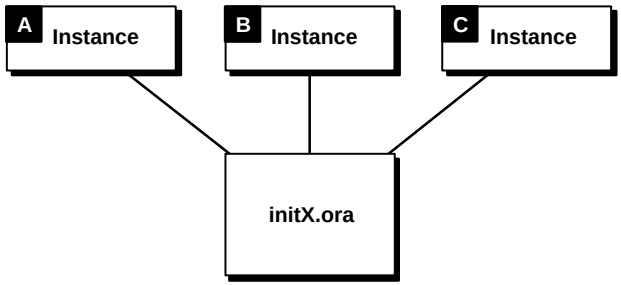
When an instance starts up, Oracle uses the values found in an initialization parameter file to create the System Global Area (SGA) for that instance. You can use various approaches to define multiple instances:

- Using a Common Parameter File for Multiple Instances
- Using Individual Parameter Files for Multiple Instances
- Embedding a Parameter File Using IFILE
- Specifying a Non-default Parameter File with PFILE

Using a Common Parameter File for Multiple Instances

A common parameter file for all instances, shown in Figure 18–1, can make administration easy. If file systems are shared among nodes, you can update all instances by making a change in only one place.

Figure 18–1 Instances with a Common Parameter File

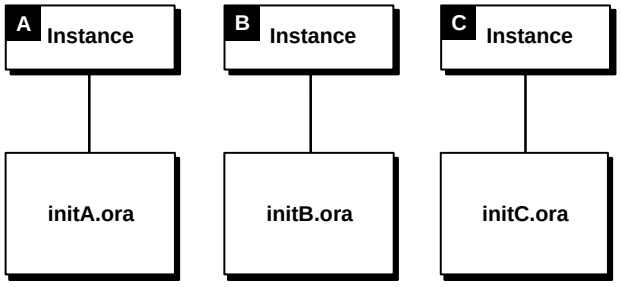


Most clustering systems, however, do not share file systems. In this case you would have to make for each node a separate physical copy of the common file.

Using Individual Parameter Files for Multiple Instances

Individual parameter files are useful when many parameters should differ from instance to instance. For example, initialization parameters to create difference size SGAs for different size machines may improve performance dramatically.

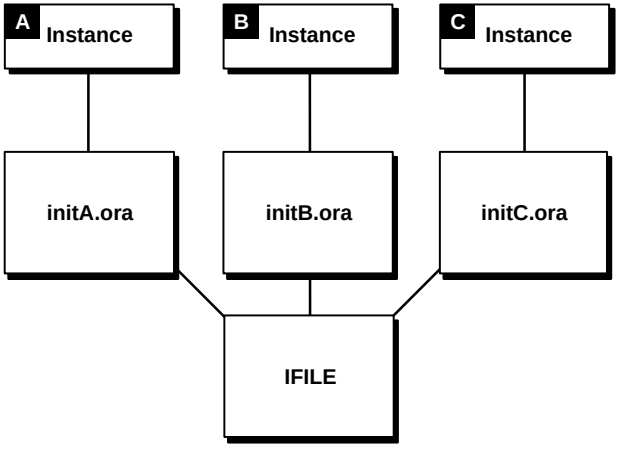
Figure 18–2 Instances with Individual Parameter Files



Embedding a Parameter File Using IFILE

By setting the IFILE parameter, each individual parameter file can embed an additional parameter file containing common values. This approach is illustrated in Figure 18–3.

Figure 18–3 Instances with Individual Parameter Files and IFILE



In a parallel server, some initialization parameters must have the same values for every instance, whether individual or common parameter files are used. By referencing the same file using the IFILE parameter, instances which have individual parameter files can ensure that they have the correct parameter values for those which must be identical, while allowing individual values for parameters which can differ.

Instances *must* use individual parameter files in the following cases:

- Every instance which uses private rollback segments must have its own parameter file, but instances which only use public rollback segments can all use the same parameter file.
- Every instance which specifies the INSTANCE_NUMBER or THREAD parameter must have its own parameter file.

Example

For example, a Server Manager session on the local node can start up two instances on remote nodes using individual parameter files named INIT_A.ORA and INIT_B.ORA:

```
SET INSTANCE instance1;
STARTUP PFILE=init_a.ora;
SET INSTANCE instance2;
STARTUP PFILE=init_b.ora;
```

Here, “instance1” and “instance2” are Net8 aliases for the two respective instances, as defined in TNSNAMES.ORA.

Both individual parameter files can use the IFILE parameter to include parameter values from the file INIT_COMMON.ORA. They can reference this file as follows:

INIT_A.ORA:

```
IFILE=INIT_COMMON.ORA
INSTANCE_NUMBER=1
THREAD=1
```

INIT_B.ORA:

```
IFILE=INIT_COMMON.ORA
INSTANCE_NUMBER=2
THREAD=2
```

The INIT_COMMON.ORA file can contain the following parameter settings, which must be identical on both instances.

```
DB_NAME=DB1
CONTROL_FILES=(CTRL_1,CTRL_2,CTRL_3)
GC_FILES_TO_LOCKS="1=600:2-4,9=500EACH:5-8=800"
GC_ROLLBACK_SEGMENTS=10
GC_SEGMENTS=10
LOG_ARCHIVE_START=TRUE
PARALLEL_SERVER=TRUE
```

Each parameter file must contain the same values for the CONTROL_FILES parameter, for example, because all instances share the control files.

To change the value of a common initialization parameter, you would only have to modify the file INIT_COMMON.ORA, rather than changing both individual parameter files.

IFILE Usage

When you specify parameters which have identical values in a common parameter file referred to by the IFILE parameter, you can omit parameters for which you are using the default values.

If you use multiple Server Manager sessions on separate nodes to start up the instances, each node must have its own copy of the common parameter file (unless the file systems are shared).

If a parameter is duplicated in an instance-specific file and the common file, or within one file, the last value specified overrides earlier values. You can therefore ensure the use of common parameter values by placing the IFILE parameter at the end of an individual parameter file. Placing IFILE at the beginning of the individual file allows you to override the common values.

You can specify IFILE more than once in a parameter file to include multiple common parameter files. Unlike the other initialization parameters, IFILE does not override previous values. For example, an individual parameter file might include a file INIT_COMMON.ORA and separate command files for the LOG_* and GC_* parameters:

```
IFILE=INIT_COMMON.ORA
IFILE=INIT_LOG.ORA
IFILE=INIT_GC.ORA
LOG_ARCHIVE_START=FALSE
THREAD=3
ROLLBACK_SEGMENTS=(RB_C1,RB_C2,RB_C3)
```

The individual value of LOG_ARCHIVE_START overrides the value specified in INIT_LOG.ORA, because the IFILE = INIT_LOG.ORA appears before LOG_ARCHIVE_START parameter specification. The individual GC_* values specified in INIT_GC.ORA override any values specified in INIT_COMMON.ORA, because IFILE = INIT_GC.ORA comes after IFILE = INIT_COMMON.ORA.

See Also: "Instance Numbers and Startup Sequence" on page 18-14.
"Redo Log Files" on page 6-3.
"Parameters Which Must Be Identical on Multiple Instances" on page 18-10.

Specifying a Non-default Parameter File with PFILE

The PFILE option of the STARTUP command allows you to specify a parameter file other than the default file when you start up an instance. The parameter file specified by PFILE must be on a disk accessible to the local node, even for an instance on a remote node.

Setting Initialization Parameters for the Parallel Server

This section discusses initialization parameters which are important for a parallel server.

- GC_* Global Constant Parameters
- Parameter Notes for Multiple Instances
- Parameters Which Must Be Identical on Multiple Instances

See Also: *Oracle8 Reference* for details about other Oracle initialization parameters.

GC_* Global Constant Parameters

Initialization parameters with the prefix GC (Global Constant) are relevant only for a parallel server. The settings of these parameters determine the size of the collection of global locks which protect the database buffers on all instances. The settings you choose affect use of certain operating system resources.

The first instance to start up in shared mode determines the values of the global constant parameters for all instances. The control file records the values of the GC_* parameters when the first instance starts up.

When another instance attempts to start up in shared mode, Oracle compares the values of the global constant parameters in its parameter file with those already in use and issues a message if any values are incompatible. The instance cannot mount the database unless it has correct values for its global constant parameters.

The global constant parameters for a parallel server are:

GC_FILES_TO_LOCKS	controls data block locks
GC_LATCHES	controls lock element latches for LCK processes
GC_LCK_PROCS	controls number of background lock processes
GC_ROLLBACK_LOCKS	controls undo block locks
GC_RELEASABLE_LOCKS	controls the number of locks

See Also: Chapter 15, “Allocating PCM Instance Locks”.

Parameter Notes for Multiple Instances

Multi-instance issues concerning initialization parameters are summarized in Table 18–1.

Table 18–1 Initialization Parameter Notes for Multiple Instances

Parameter	Parallel Server Notes
CHECKPOINT_PROCESS	In Oracle Parallel Server your database may have more datafiles. To speed up checkpoints, enable the CHECKPOINT_PROCESS parameter.
DELAYED_LOGGING_BLOCK_CLEANOUTS	The default value, True, helps reduce pinging between instances.
DML_LOCKS	Must be identical on all instances only if set to zero.
INSTANCE_NUMBER	If specified, this parameter must have unique values for different instances.
LOG_ARCHIVE_FORMAT	You must include thread number.
MAX_COMMIT_PROPAGATION_DELAY	If you want commits to be seen immediately on remote instances, you may need to change the value of this parameter.
NLS_* parameters	Can have different values for different instances.
PARALLEL_SERVER	To enable parallel server this parameter must be set to TRUE in the initialization file. It defaults to FALSE.
PROCESSES	This parameter must have a value large enough to allow for all background processes and all user processes in an instance. Some operating systems can have additional DBWR processes. Defaults for the SESSIONS and TRANSACTIONS parameters are derived directly or indirectly from the value of the PROCESSES parameter If you do not use the defaults, you may want to increase some of these parameter values to allow for LCK*n* and other optional background processes.
RECOVERY_PARALLELISM	To speed up the roll forward or cache recovery phase, you may want to set this parameter.
ROLLBACK_SEGMENTS	Specify the private rollback segments for each instance.
THREAD	If specified, this parameter must have unique values for different instances.

See Also: *Oracle8 Reference* for details about each parameter.

Parameters Which Must Be Identical on Multiple Instances

Certain initialization parameters that are critical at database creation or that affect certain database operations must have the same value for every instance in a parallel server. For example, the values of DB_BLOCK_SIZE and CONTROL_FILES must be identical for every instance. Other parameters can have different values for different instances. The following initialization parameters must have identical values for every instance in a parallel server:

CACHE_SIZE_THRESHOLD
CONTROL_FILES
CPU_COUNT
DB_BLOCK_SIZE
DB_FILES
DB_NAME
DML_LOCKS (must be identical only if set to zero)
GC_FILES_TO_LOCKS
GC_LCK_PROCS
GC_ROLLBACK_LOCKS
LM_LOCKS (identical values recommended)
LM_PROCS (identical values recommended)
LM_RESS (identical values recommended)
LOG_FILES
MAX_COMMIT_PROPAGATION_DELAY
PARALLEL_DEFAULT_MAX_INSTANCES
PARALLEL_DEFAULT_MAX_SCANS
ROLLBACK_SEGMENTS
ROW_LOCKING

See Also: *Oracle8 Reference* for details about each parameter.

Setting LM_* Parameters

Set values for the LM_* initialization parameters. Note that the resources, locks and processes configurations are per OPS instance. For ease of administration, these parameters should be consistent for all the instances.

LM_LOCKS	Number of locks. Where R is the number of resources, N is the total number of nodes, and L is the total number of locks, the following calculation is used: L = R + (R*(N - 1)) / N
LM_PROCS	Number of processes. The value of PROCESSES initialization parameter multiplied by the number of nodes.
LM_RESS	This parameter controls the number of resources that can be locked by the Lock Manager. This parameter covers the number of lock resources allocated for DML, DDL (data dictionary locks), and data dictionary cache locks + file and log management locks.

Increased values will be necessary if you plan to use parallel DML or DML performed on partitioned objects.

See Also: *Oracle8 Reference*
"Planning IDLM Capacity" on page 16-2

Creating Database Objects for Multiple Instances

Creating a database automatically starts up a single instance with parallel server disabled. Before you can start up multiple instances, however, you must perform certain administrative operations. These tasks may include:

- creating extra rollback segments for each additional instance
- adding and enabling a thread for each additional instance
- providing locks for added datafiles

You can perform these operations with a single instance in either exclusive or shared mode.

See Also: "Creating Additional Rollback Segments" on page 14-5
"Redo Log Files" on page 6-3
"What Is the Total Number of PCM Locks and Resources Needed?" on page 15-19

Starting Up Instances

An Oracle instance can start up with parallel server enabled or disabled. This section includes the following topics:

- Enabling Parallel Server and Starting Instances
- Starting up with Parallel Server Disabled
- Starting Up in Shared Mode

Enabling Parallel Server and Starting Instances

Note: In Oracle8 the keywords SHARED, EXCLUSIVE, and PARALLEL are obsolete in the STARTUP and ALTER DATABASE MOUNT statements.

Starting an Instance Using SQL

1. To enable parallel server in Oracle8, you must set the PARALLEL_SERVER parameter to TRUE in the initialization file. It defaults to FALSE.
2. Start up any required operating system specific processes.

   For more information, see your Oracle system-specific documentation.
3. Start up Group Membership Services (GMS).

   See "Using Group Membership Services" on page 18-21 for more information.
4. Connect with SYSDBA or SYSOPER privileges.

   ```
   CONNECT username/password AS SYSDBA
   ```
5. Make sure the PARALLEL_SERVER initialization parameter is set to TRUE if you wish to run with parallel server enabled, or to FALSE if you wish to run with parallel server disabled.
6. Start up an instance.

   ```
   STARTUP NOMOUNT
   ```
7. Mount a database.

   ```
   ALTER DATABASE database_name MOUNT
   ```
8. Open the database.

   ```
   ALTER DATABASE OPEN
   ```

Starting an Instance Using Server Manager

Note: The Server Manager command STARTUP with the OPEN option performs steps 4, 5, and 6 of the procedure given above.

1. Start up any required operating system specific processes.

   For more information, see your Oracle system-specific documentation.

2. Set the PARALLEL_SERVER initialization parameter to TRUE if you wish to run with parallel server enabled, or to FALSE if you wish to run with parallel server disabled.

3. Start up Group Membership Services (GMS).

   See "Using Group Membership Services" on page 18-21 for more information.

4. Start Server Manager.

5. Start up an instance with the OPEN option:

   `STARTUP OPEN` *database_name*

Starting up with Parallel Server Disabled

Parallel server must be disabled whenever you change the archiving mode (ARCHIVELOG or NOARCHIVELOG). To change the archiving mode, the database must be mounted but not open.

If an instance mounts a database with PARALLEL_SERVER set to FALSE, no other instance can mount the database.

Before you can start up an instance in exclusive mode, you must shut down all instances running in shared mode. A single instance running in shared mode is not the same as an instance running in exclusive mode, and the last instance running in shared mode does not automatically revert to exclusive mode.

An instance starting up with parallel server disabled can specify an instance number with the INSTANCE_NUMBER parameter. This is only necessary if the instance will perform inserts and updates and if the tables in your database use the FREELIST GROUPS storage option to allocate free space to instances. If you start up an instance just to perform administrative operations with parallel server disabled, you can omit the INSTANCE_NUMBER parameter from the parameter file.

An instance starting up with parallel server disabled can also specify a thread other than 1, to use the online redo log files associated with that thread.

See Also: Chapter 17, “Using Free List Groups to Partition Data”

Starting Up in Shared Mode

In a parallel server, each instance must mount the database in shared mode. Each initialization parameter file for each instance must have the SINGLE_PROCESS parameter set to FALSE and the PARALLEL_SERVER parameter set to TRUE. Before you start up multiple instances in shared mode, you must create at least one rollback segment for each instance sharing the same database and enable a thread containing at least two groups of redo log files for each additional instance.

If one instance mounts a database in shared mode, other instances can also mount the database in shared mode, but not in exclusive mode.

If PARALLEL_SERVER is set to FALSE, the instance tries to start up with parallel server disabled by default.

Retrying to Mount a Database in Shared Mode

If you attempt to start an instance and mount a database in shared mode while another instance is currently recovering the same database, your new instance cannot mount the database until the recovery is complete.

Rather than repeatedly attempting to start the instance, you can use the STARTUP RETRY statement. This causes the new instance to retry every five seconds to mount the database until it succeeds or has reached the retry limit. For example:

```
STARTUP OPEN maildb RETRY
```

To set the maximum number of times the instance attempts to mount the database, use the Server Manager SET command with the RETRY option; you can specify either an integer (such as 10) or the keyword INFINITE.

If the database cannot be opened for some reason other than recovery by another instance, then the RETRY will not repeat. For example, if the database was mounted in exclusive mode by one instance, then trying the STARTUP RETRY command in shared mode will not work for another instance.

Instance Numbers and Startup Sequence

When an instance starts up, it acquires an instance number which maps the instance to one group of free lists for each table created with the FREELIST GROUPS storage option.

An instance can specify its instance number explicitly by using the initialization parameter INSTANCE_NUMBER when it starts up with parallel server enabled or disabled. If an instance does not specify the INSTANCE_NUMBER parameter, it automatically acquires the lowest available number.

Startup order determines the instance numbers for instances which do not specify the INSTANCE_NUMBER parameter. Startup numbers are difficult to control if instances start up in parallel, and they can change after instances shut down and restart.

Instances which use the INSTANCE_NUMBER parameter must specify different numbers. The Server Manager command SHOW PARAMETERS INSTANCE_NUMBER can show the current instance number each instance is using. This command displays a null value if an instance number was assigned based on startup order.

After an instance shuts down, its instance number becomes available again. If a second instance starts up before the first instance restarts, the second instance can acquire the instance number previously used by the first instance.

Instance numbers based on startup order are independent of instance numbers specified with the INSTANCE_NUMBER parameter. After an instance acquires an instance number by one of these methods (either with or without INSTANCE_NUMBER), another instance cannot acquire the same number by the other method. All numbers are unique, regardless of the method by which they are acquired.

Always use the INSTANCE_NUMBER parameter if you need a consistent allocation of extents to instances for inserts and updates. This allows you to maintain data partitioning among instances.

See Also: "Rollback Segments" on page 6-8
"Creating Additional Rollback Segments" on page 14-5
"Redo Log Files" on page 6-3
Chapter 17, “Using Free List Groups to Partition Data”, for information about allocating free space for inserts and updates.

Specifying Instances

When performing administrative operations in a multi-instance environment, you must be sure that you have specified the correct instance. This section includes the following topics:

- Differentiating Between Current and Default Instance
- How SQL Statements Apply to Instances
- How Server Manager Commands Apply to Instances
- Specifying Instance Groups
- Using a Password File to Authenticate Users on Multiple Instances

Differentiating Between Current and Default Instance

Some Server Manager commands apply to the instance to which Server Manager is currently connected, and others apply to the default instance.

default instance	The *default instance* is on the machine where you initiate Server Manager. Server Manager commands which *cannot* be used while you are connected to an instance (such as executing a host command) apply to the default instance.
current instance	The *current instance* is determined by the CONNECT command. Server Manager commands which *can* be used while you are connected to an instance apply to the current instance.

The current instance can be different from the default instance if you specify a connect string in the CONNECT command.

Net8 must be installed to use the SET INSTANCE or CONNECT command for an instance running on a remote node.

See Also: Your platform-specific Oracle documentation, for more information about installing Net8 and the exact format required for the connect string used in the SET INSTANCE and CONNECT commands.

How SQL Statements Apply to Instances

Instance-specific SQL statements apply to the current instance. For example, the statement ALTER DATABASE ADD LOGFILE only applies to the instance to which you are currently connected, rather than the default instance or all instances.

ALTER SYSTEM CHECKPOINT LOCAL applies to the current instance. By contrast, ALTER SYSTEM CHECKPOINT or ALTER SYSTEM CHECKPOINT GLOBAL applies to *all* instances.

ALTER SYSTEM SWITCH LOGFILE applies only to the current instance. To force a global log switch, you can use ALTER SYSTEM ARCHIVE LOG CURRENT. The THREAD option of ALTER SYSTEM ARCHIVE LOG allows you to archive online redo log files for a specific instance.

How Server Manager Commands Apply to Instances

When you initiate Server Manager, the commands you enter are relevant to the default instance, which is also the current instance.

This is true until you use the SET INSTANCE command to set the current instance. From that point onwards, all Server Manager commands operate on the current instance.

Table 18–2 How Server Manager Commands Apply to Instances

Server Manager Command	Associated Instance
ARCHIVE LOG	always applies to the current instance
CONNECT	uses the default instance if no instance is specified in the CONNECT command
HOST	applies to the node running the Server Manager session, regardless of the location of the current and default instances
MONITOR	MONITOR display screens identify the current instance, not the default instance, in the upper left corner.
RECOVER	does not apply to any particular instance, but rather to the database
SHOW INSTANCE	displays information about the default instance, which can be different from the current instance
SHOW PARAMETERS	displays information about the current instance
SHOW SGA	displays information about the current instance
SHUTDOWN	always applies to the current instance. A privileged Server Manager command.
STARTUP	always applies to the current instance. A privileged Server Manager command.

Note: The security mechanism invoked when you use privileged Server Manager commands depends on the operating system you are using. Most operating systems have a secure authentication mechanism when logging onto the operating system. On these systems, your default operating system privileges will usually determine whether you can use STARTUP and SHUTDOWN. For more information, see your Oracle system-specific documentation.

The SET INSTANCE and SHOW INSTANCE Commands

You can change the default instance with the Server Manager statement:

SET INSTANCE *instance_path*

where *instance_path* is a valid Net8 connect string (without a user ID / password). If you are connected to an instance, you must disconnect before using SET INSTANCE. Alternatively, if you do not wish to disconnect from the current instance, you may use the CONNECT command with *instance_path*.

You can use the SET INSTANCE command to specify an instance on a remote node for the commands STARTUP and SHUTDOWN. The parameter file for a remote instance must be on the local node.

The SHOW INSTANCE command displays the connect string for the default instance. SHOW INSTANCE returns the value *local* if you have not used SET INSTANCE during the Server Manager session.

To reset to the default instance, use SET INSTANCE without specifying a connect string or specify LOCAL (but not DEFAULT, which would indicate a connect string for an instance named “DEFAULT”).

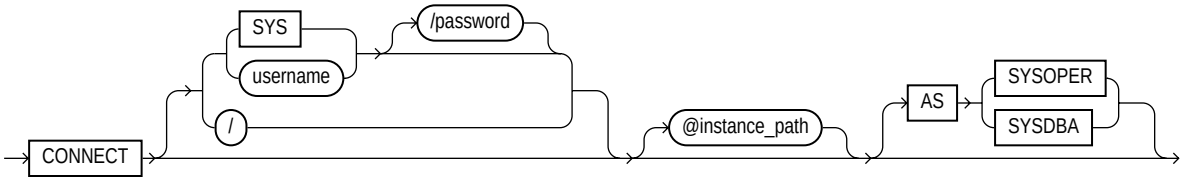
The following Server Manager line mode examples illustrate the relationship between SHOW INSTANCE and SET INSTANCE:

```
SHOW INSTANCE
Instance                       local

SET INSTANCE node1
Oracle8 Server Release 8.0 - Production
With the distributed, parallel query and Parallel Server options
PL/SQL V8.0 - Production

SHOW INSTANCE
Instance                       node2

SET INSTANCE
ORACLE8 Server Release 8.0 - Production
With the procedural, distributed, and Parallel Server options
PL/SQL V8.0 - Production

SHOW INSTANCE
Instance                       local

SET INSTANCE DEFAULT
ORA-06030: NETDNT: connect failed, unrecognized node name
```

The CONNECT Command

The CONNECT command can associate Server Manager with either the default instance or an instance which you specify explicitly. The instance to which Server Manager connects becomes the *current instance*.

The CONNECT command has the following syntax:



where *instance-path* is a valid Net8 connect string. CONNECT without the argument *@instance-path* connects to the default instance (which may have been set previously with SET INSTANCE).

Connecting as SYSOPER or SYSDBA allows you to perform privileged operations, such as instance startup and shutdown.

Multiple Server Manager sessions can connect to the same instance at the same time. When you are connected to one instance, you can connect to a different instance without using the DISCONNECT command. Server Manager disconnects you from the first instance automatically whenever you connect to another one.

The CONNECT *@instance-path* command allows you to specify an instance before using the Server Manager commands MONITOR, STARTUP, SHUTDOWN, SHOW SGA, and SHOW PARAMETERS.

See Also: *Oracle Server Manager User's Guide* for syntax of Server Manager commands.
Oracle Net8 Administrator's Guide for the proper specification of *instance_path*.
Oracle8 Administrator's Guide for information on connecting with SYSDBA or SYSOPER privileges.

Using Group Membership Services

Group Membership Services (GMS) is used by the Lock Manager (LM) and other Oracle components for inter-instance initialization and coordination. Instances of a distributed service can register with the GMS and retrieve the current set of instances providing the same service cluster-wide. The GMS monitors each of its clients and notifies the other instances of a given service when one instance stops or is shut down. It obtains a view of the current cluster membership from the (system specific) cluster management software.

If a GMS instance or a node stops, the remaining GMS instances are informed through the cluster manager. Providers of distributed services are then be notified by the GMS if any of their peers stopped as a result of the node stoppage.

Platforms which use the **opsctl** program start GMS automatically. For other platforms, you must start this process by manually issuing the **ogmsctl** command. This program has the following options:

start	start up the GMS instance on the current node
stop	stop the GMS instance on the current node
abort	kill the GMS instance on the current node
status	check the status of the GMS on the current node
ogms_home=*X*	set the GMS home directory to *X*, where debugging trace files and the communication key file will be written during its operation. The directory must be either local to its own node or different from other GMS directories in the network environment. Multiple GMS instances could be running at the same time in a cluster. When this option is specified, the corresponding OGMS_HOME initialization parameter needs to be specified as well to pick up the GMS listen port id information.
trace=*n*	set the GMS trace level to a number n between 0 and 10 (with 10 being the highest trace level). You can use this option with the start option or specify it separately after the GMS has been started.

When you have installed the Oracle Parallel Server option, you must start the GMS, even to bring the instance up with parallel server disabled. If OPS is linked in, Oracle starts the Integrated Distributed Lock Manager and connects to the GMS to obtain a mount lock. This prevents you from accidentally mounting the database exclusive on more than one mode.

See Also: Your Oracle system-specific documentation to determine whether GMS is started automatically, and whether it requires additional cluster configuration.

Specifying Instance Groups

For ease in administration, you can logically group different instances together and perform parallel operations upon all of the associated instances at once. You can define an *instance group* as a set of instances used for a specific purpose (such as resource allocation, parallel query or other parallel operations). They thus enable you to partition your resources effectively.

Sometimes, for example, a DBA may wish to prevent users or query processes from obtaining resources on all instances. The DBA may want to keep certain instances available only for users running OLTP processes, and restrict users running parallel queries only to a particular set of instances.

For example, you might create instance groups such that between 9 AM and 5 PM users can use group B, but after 5 PM they can use group D. Or, you might use group C for normal OLTP inserts and updates but use group D for big parallel tasks, to avoid interfering with OLTP performance.

- You should define all potentially desirable group configurations during startup since you cannot add and delete instances from groups dynamically while the instance is up.
- One instance can be a member of more than one group at any given time. Groups may overlap one another.
- You can define as many groups as you wish, but only use them as needed. Instance groups do not incur much overhead, and you are not required to refer to them, once they are defined.

If you simply set the degree of parallelism, the system chooses which specific instances to use (given disk affinity, and the number of instances actually running). By specifying instance groups you can directly specify the instances which should be used for parallel operations.

Note that the instance from which you initiate a query, need not be a member of the group of instances which carry out the query. The parallel coordinator does run on the current instance.

How to Specify Instance Groups

To specify instance groups, set the INSTANCE_GROUPS initialization parameter within the parameter file of each instance you wish to associate to the group. This parameter at once defines a group and adds the current instance to the group.

For example, instance 1 could set the parameter as follows:

```
INSTANCE_GROUPS = groupB, groupD
```

Instance 3 could set it as follows:

```
INSTANCE_GROUPS = groupA, groupD
```

As a result, instances 1 and 3 would both belong to instance group D, but would also belong to other groups as well.

Note that INSTANCE_GROUPS cannot be changed dynamically.

How to Use Instance Groups

You can use instance groups for two purposes:

- to identify a group to be used for a parallel operation (with PARALLEL_INSTANCE_GROUP)
- to identify instances which should return information in a GV$ fixed-view query (with OPS_ADMIN_GROUP)

The default for PARALLEL_INSTANCE_GROUP and OPS_ADMIN_GROUP is a group consisting of all currently running instances.

To use a particular instance group for a given parallel operation, specify the following parameter in the initialization parameter file:

```
PARALLEL_INSTANCE_GROUP = groupname
```

All parallel operations initiated from that instance will spawn processes only within that group, using the same algorithm as before (either randomly or with disk affinity).

PARALLEL_INSTANCE_GROUP is a dynamic parameter which you can change using an ALTER SESSION or ALTER SYSTEM statement. You can use it to refer to only *one* instance group; by default it is set to a default group which includes all currently active instances. The instance upon which you are running need not be a part of the instance group which you are going to use for a particular operation.

To determine the instances which should return information in a GV$*viewname* query, set the OPS_ADMIN_GROUP parameter.

See Also: *Oracle8 Reference* for complete information about initialization parameters and views.
"Global Dynamic Performance Views" on page 20-3 for information about the OPS_ADMIN_GROUP parameter.

How to List the Members of Instance Groups

To find out the members of the different instance groups you can query the GV$ global dynamic performance view GV$PARAMETER. Look at all entries for the INSTANCE_GROUPS parameter name.

Instance Group Example

In this example, instance 1 has the following settings in its initialization parameter file:

```
INSTANCE_GROUPS = Ga, Gb
PARALLEL_INSTANCE_GROUP = Gb
```

Instance 2 has the following setting in its initialization parameter file:

```
INSTANCE_GROUPS = Gb, Gc
PARALLEL_INSTANCE_GROUP = Gc
```

On instance 1, if you enter the following statements, the instances in Gb will be used. Two server processes will be spawned on instance 1, and 2 server processes on instance 2.

```
ALTER TABLE table PARALLEL (DEGREE 2 INSTANCES 2);
SELECT COUNT(*) FROM table;
```

If you enter the following statements on instance 1, Gc will be used. Two server processes will be spawned on instance 2 only.

```
ALTER SESSION SET PARALLEL_INSTANCE_GROUP = 'Gc';
SELECT COUNT(*) FROM table;
```

If you enter the following statements on instance 1, the default instance group (all currently running instances) will be used. Two server processes will be spawned on instance 1, and 2 server processes on instance 2.

```
ALTER SESSION SET PARALLEL_INSTANCE_GROUP = '';
SELECT COUNT(*) FROM table;
```

Using a Password File to Authenticate Users on Multiple Instances

You can use a password file to authenticate users performing database administration when running multiple instances on a parallel server. In this case, the environment variable for each instance must point to the same password file. Similarly, the REMOTE_LOGIN_PASSWORDFILE initialization parameter for each instance must be set to the appropriate, identical value.

See Also: *Oracle8 Administrator's Guide* for information about the REMOTE_LOGIN_PASSWORDFILE parameter.

For more information on the exact name of the password file, or for the name of the environment variable used to specify this name for your operating system, see your Oracle system-specific documentation.

Shutting Down Instances

Use the following procedure to shut down an instance:

1. Connect with SYSDBA.

   ```
   CONNECT username/password AS SYSDBA
   ```

2. Close the database.

   ```
   ALTER DATABASE database_name CLOSE
   ```

3. Dismount the database.

   ```
   ALTER DATABASE database_name DISMOUNT
   ```

Alternatively, you can use the Server Manager command SHUTDOWN, which performs all three of these steps for the current instance.

In a parallel server, shutting down one instance does not interfere with the operations of any instances still running.

To shut down a database which is mounted in shared mode, you must shut down every instance in the parallel server. The parallel server allows you to shut down instances in parallel from different nodes. When an instance shuts down abnormally, Oracle forces all user processes running in that instance to log off the database. If a user process is currently accessing the database, Oracle terminates that access and returns the message "ORA-1092: Oracle instance terminated. Disconnection forced". If a user process is not currently accessing the database when the instance shuts down, Oracle returns the message "ORA-1012: Not logged on" upon the next call or request made to Oracle.

After a NORMAL or IMMEDIATE shutdown, instance recovery is not required. Recovery is required, however, after the SHUTDOWN ABORT command or after an instance terminates abnormally. The SMON process of an instance which is still running performs instance recovery for the instance which shut down. If no other instances are running, the next instance to open the database performs instance recovery for any instances which need it.

If multiple Server Manager sessions are connected to the same instance simultaneously, all but one must disconnect before the instance can be shut down normally. You can use the IMMEDIATE or ABORT option of the SHUTDOWN command to shut down an instance when multiple Server Manager sessions (or any other sessions) are connected to it.

See Also: "Starting Up and Shutting Down" in *Oracle8 Administrator's Guide* for options of the SHUTDOWN command.

Limiting Instances for the Parallel Query

Although the parallel query feature does not require the Oracle Parallel Server, some aspects of parallel query apply only to a parallel server.

The INSTANCES keyword of the PARALLEL clause of the CREATE TABLE, ALTER TABLE, CREATE CLUSTER, and ALTER CLUSTER commands allows you to specify that a table or cluster is split up among the buffer caches of all available instances of a parallel server when the table is scanned in a parallel query.

If you do not want tables to be dynamically partitioned among all the available instances, you can specify the number of instances that can participate in scanning or caching with the PARALLEL_DEFAULT_MAX_INSTANCES parameter or the ALTER SYSTEM command.

If you want to specify the number of instances to participate in parallel query processing at startup time, you can specify a value for the initialization parameter PARALLEL_DEFAULT_MAX_INSTANCES.

If you want to limit the number of instances available for parallel query processing dynamically, use the ALTER SYSTEM command. For example, if your parallel server has ten instances running, but you want only eight to be involved in parallel query processing, while the remaining two instances will be dedicated for other use, you can issue the following command:

```
ALTER SYSTEM SET SCAN_INSTANCES = 8;
```

Thereafter, if a table's definition has a value of ten specified for the INSTANCES keyword, the table will be scanned by query servers on only eight of the ten instances. Oracle selects the first eight instances in this example. You can set the PARALLEL_MAX_SERVERS initialization parameter to zero on the instances that you do not want to participate in parallel query processing.

If you wish to limit the number of instances that cache a table, you can issue the following command:

```
ALTER SYSTEM SET CACHE_INSTANCES = 8;
```

Thereafter, if a table definition has 10 specified for the INSTANCES keyword and the CACHE keyword was specified, the table will be divided evenly among eight of the ten available instances' buffer caches.

See Also: "Specifying Instance Groups" on page 18-22.

Oracle8 Reference for more information about parameters.

19

Tuning the System to Optimize Performance

Last of the gods, Contention ends her tale.

Aeschylus, *Antigone*

This chapter provides an overview of tuning issues. It covers the following topics:

- General Guidelines
- Contention
- Tuning for High Availability

See Also: "Oracle Parallel Server Management" on page 18-2

General Guidelines

This section covers the following topics:

- Overview
- Keep Statistics for All Instances
- Statistics to Keep
- Change One Parameter at a Time

Overview

With experience, you can anticipate most parallel server application problems prior to rollout and testing of the application. This can be done using the methods described in this document. In addition, a number of tunable parameters can enhance system performance. Tuning parameters can have a major influence and improve system performance, but they cannot overcome problems caused by a poor analysis of potential LM lock contention.

In tuning OPS applications the techniques used for single-instance applications are still valid. It is still important, however, to effectively tune the buffer cache, shared pool and all the disk subsystems. OPS introduces some additional parameters and statistics that you must collect and understand.

Note: In Oracle8, locks are mastered and remastered dynamically, so the instances do not need to be started in any particular order.

When collecting statistics to monitor the performance of the OPS, the following general guidelines will make debugging and monitoring of the system simpler and more accurate.

Keep Statistics for All Instances

It is important to monitor all instances in the same way, but keep separate statistics for each instance. This is particularly true if the partitioning strategy results in a highly asymmetrical solution. By monitoring the instances you can determine the highest loaded node and test how well the system partitioning has been performed.

Statistics to Keep

The best statistics to monitor within the database are those kept within the SGA: the “V$” and “X$” tables, for example. It is best to snapshot these views over a period of time. In addition, good operating system statistics should be kept to assist

in the monitoring and debugging process. The most important of these are CPU usage, disk I/O, virtual memory usage, network usage and lock manager statistics.

Change One Parameter at a Time

In benchmarking or capacity planning exercises it is important to manage effectively the changes to the setup of the system. By documenting each change and effectively quantifying its effect on the system, you can profile and understand the mechanics of the system and application. This is particularly important when debugging a system or determining whether more hardware resources are required. You must adopt a systematic approach for the measurement and tuning phases of a performance project. Although this approach may seem time consuming, it will save time and system resources in the long term.

Contention

This section covers the following topics:

- Detecting Lock Conversions
- Pinpointing Lock Contention within an Application

Detecting Lock Conversions

To detect whether a large number of lock conversions is taking place, you can examine the "V$" tables which enable you to see that locks are being upgraded and downgraded. The best views for initially determining whether a lock contention problem exists are V$LOCK_ACTIVITY and V$SYSSTAT.

To determine the number of lock converts over a period of time, query the V$LOCK_ACTIVITY table. From this you should be able to determine whether you have reached the maximum lock convert rate for the LM. If this is the case, you must repartition the application to remove the bottleneck. In this situation, adding more hardware resources such as CPUs, disk drives, and memory is unlikely to improve system performance significantly.

Note: Maximum lock convert rate depends on the implementation of the IPC mechanism on your platform.

To determine whether lock converts are being performed too often, calculate how often the transaction requires a lock convert operation when a data block is accessed for either a read or a modification. Query the V$SYSSTAT table.

In this way you can calculate a lock hit ratio which may be compared to the cache hit ratio. The value calculated is the number of times there occur data block

accesses that do not require lock converts, compared to the total number of data block accesses. The lock hit ratio is computed as:

$$\frac{consistent\_gets - global\_lock\_converts\_(async)}{consistent\_gets}$$

A SQL statement that computes this ratio is as follows:

```
SELECT (b1.value - b2.value) / b1.value ops_ratio
  FROM V$SYSSTAT b1, V$SYSSTAT b2
 WHERE b1.name = 'consistent gets'
   AND b2.name = 'global lock converts (async)';
```

If this ratio drops below 95%, optimal scaling of performance may not be achieved as additional nodes are added.

Another indication of too many PCM lock conversions is the ping / write ratio, which is determined as follows:

$$ping\_write\_ratio = \frac{DBWR\_cross\_instance\_writes}{physical\_writes}$$

See Also: "Tuning Your PCM Locks" on page 15-16

Pinpointing Lock Contention within an Application

If an application shows performance problems and you determine that excessive lock convert operations are the major problem, you must identify the transactions and SQL statements which are causing the problem. When excessive lock contention occurs it is likely to be caused by one of three problem areas when setting up the OPS environment. These key areas are as follows:

- contention for a common resource
- lack of locks
- constraints

Excessive Lock Convert Rates: Contention for a Common Resource

This section describes excessive lock conversion rates associated with contention for a common resource.

In some cases within OPS applications the system may not be performing as anticipated. This may be because one small area of the database setup or application design overlooked some database blocks that must be accessed in exclusive mode by all instances, for the entire time that the application runs. This forces the whole system to effectively single thread with respect to this resource.

This problem can also occur in single instance cases where all users require exclusive access to a single resource. In an inventory system, for example, all users may wish to modify a common stock level.

In OPS applications the most common points for contention are associated with contention for a common set of database blocks. To determine whether this is happening you can query an additional set of V$ tables (VBH, VCACHE and V$PING). All these tables yield basically the same data, but V$CACHE and V$PING have been created as views joining additional data dictionary tables to make them easier to use. These tables and views examine the status of the current data blocks within an instance's SGA. They enable you to construct queries to see how many times a block has been pinged between nodes, and how many revisions of the same data block exist within the SGA at the same time. You can use both of these features to determine whether excessive single threading upon a single database block is occurring.

Note: GV$BH, GV$CACHE, and GV$PING views are also available, enabling you to query across all instances.

The most common areas of high block contention tend to be:

- Free list contention by INSERT statements requiring more free space to insert into a table. Often you can recognize this by querying V$PING and noticing that a single block has multiple copies in the SGA. If this is the second block in the file, free list contention is probably occurring. This problem may actually be solved by use of free list groups and multiple free lists. Free list contention may also occur on single-instance systems, especially SMP machines with a large number of CPUs. This problem can be determined by querying the V$WAITSTAT table.
- Segment header contention for transactions sharing the same space header management block. This is likely to occur during parallel index creates, when the parallel query slaves allocate sorting space from the temporary tablespace. Each segment that undergoes simultaneous space management in a parallel server requires approximately 9 distributed locks dedicated to coordinating space management activities.
- Index contention by INSERT and DELETE statements that operate upon an indexed table. By querying V$PING you can see that a number of data blocks within the first extent of the index have both multiple copies within the SGA and experience a high number of block pings. This problem cannot be solved by tuning; to solve this problem it is important to localize all access (read or write) to this index to a single instance.

  This will involve routing transactions that alter this table to a single instance and running the system asymmetrically. If this cannot be done further consideration should be given to partitioning the table and using a data dependent routing strategy.

Excessive Lock Convert Rates through Lack of Locks

In tables that have random access for SELECT, UPDATE and DELETE statements, each node will need to perform a number of PCM lock upgrades and downgrades. If these lock convert operations require a disk I/O they will be particularly expensive and performance will be affected.

If, however, many of the lock converts can be satisfied by just converting the lock without a disk I/O, a performance improvement can be made. This is often referred to as an I/O less ping. The reason that the lock convert can be achieved without an I/O is that the database is able to age the data blocks out of the SGA via the database writer, as it would with a single instance. This is only likely when the table is very large in comparison to the size of the SGA. Small tables are likely to require a disk I/O, since they are unlikely to be aged out of the SGA.

With small tables where random access occurs you can still achieve performance improvements by reducing the number of rows stored in a data block. You can do this by increasing the table PCTFREE value and by reducing the number of data blocks managed by a PCM lock. The process of adjusting the number of rows managed per PCM lock can be performed until lock converts are minimized or the hardware configuration runs out of PCM locks.

The number of PCM locks managed by the LM is not an infinite resource. Each lock requires memory on each OPS node, and this resource may be quickly be exhausted. Within an OPS environment the addition of more PCM locks lengthens the time taken to restart or recover an OPS instance. In environments where high availability is required, the time taken to restart or recover an instance may determine the maximum number of PCM locks that you can practically allocate.

Excessive Lock Convert Rates Due to Constraints

In certain situations excessive lock conversion rates cannot be reduced due to certain constraints. In large tables, clusters, or indexes many gigabytes in size, it becomes impossible to allocate enough PCM locks to prevent high lock convert rates even if these are all false pings. This is mainly due to the physical limitations of allocating enough locks. In this situation a single PCM lock may effectively manage more than a thousand data blocks.

Where random access is taking place, lock converts are performed even if there is not contention for the same data block. In this situation tuning the number of locks is unlikely to enhance performance, since the number of locks required is far in excess of what can actually be created by the lock manager.

In such cases you must either restrict access to these database objects or else develop a partitioned solution.

Tuning for High Availability

Failure of an Oracle instance on one Parallel Server node may be caused by problems that may or may not require rebooting the failed node. If the node fails and requires reboot or restart, the recovery process on remaining nodes will take longer. Assuming a full recovery is required the recovery process will be performed in three discreet phases:

- Detection of Error
- Recovery and Re-mastering of IDLM Locks
- Recovery of Failed Instance

Detection of Error

The first phase of recovery is to detect that either a node or an OPS instance has failed. Complete node failure or failure of an Oracle instance is detected through the operating system node management facility.

Recovery and Re-mastering of IDLM Locks

If a complete node failure has occurred, the remaining nodes will be required to re-master the locks held by the failed node. On non-failed instances at this point all database processing will halt until recovery has completed. To speed this process for the Integrated DLM it is important to have the minimum number of PCM locks. This will eventually be reflected in a trade-off between database performance and availability requirements.

Recovery of Failed Instance

Once the IDLM has recovered all lock information, one of the remaining nodes can get an exclusive lock on the failed instance's IDLM instance lock. This node enables the failed instance to recover by providing roll forward / roll backward recovery of the failed instance's redo logs. This process is performed by the SMON background process. The time needed to perform this process depends upon the quantity of redo logs to be recovered, a function of how often the system was checkpointed at runtime. Again, this is a trade-off between system runtime performance, which favors a minimum of checkpoints, and system availability requirements.

See Also: "Phases of Oracle Instance Recovery" on page 22-14.

20

Monitoring Views & Tuning a Parallel Server

A needless Alexandrine ends the song,
That like a wounded snake drags its slow length along.
—Alexander Pope

This chapter describes how to monitor performance of a parallel server by querying data dictionary views and dynamic performance views. It also explains how to tune a parallel server.

- Monitoring Data Dictionary Views with CATPARR.SQL
- Monitoring Dynamic Performance Views
- Querying V$LOCK_ACTIVITY to Monitor Instance Lock Activity
- Querying the V$PING View to Detect Pinging
- Querying V$CLASS_PING, V$FILE_PING, and V$BH
- Querying the V$WAITSTAT View to Monitor Contention
- Querying V$FILESTAT and V$DATAFILE to Monitor I/O Activity
- Querying and Interpreting V$SESSTAT and V$SYSSTAT Statistics

See Also: "Oracle Parallel Server Management" on page 18-2

Monitoring Data Dictionary Views with CATPARR.SQL

The SQL script CATPARR.SQL creates parallel server data dictionary views. To run this script, you must have SYSDBA privileges and either log in with the SYS username or use the CONNECT INTERNAL command.

Note: CONNECT INTERNAL may not be supported in future releases.

CATALOG.SQL creates the standard V$ dynamic views, as described in the *Oracle8 Reference*, including:

- GV$CACHE
- GV$PING
- GV$CLASS_PING
- GV$FILE_PING

You can rerun CATPARR.SQL if you want the EXT_TO_OBJ table to contain the latest information after you add extents. Note that if you drop objects without rerunning CATPARR.SQL, EXT_TO_OBJ may display misleading information.

The following data dictionary views, created by CATPARR.SQL, are available to monitor a parallel server:

- FILE_LOCK
- FILE_PING

See Also: *Oracle8 Reference* for more information on dynamic views and monitoring your database.

Monitoring Dynamic Performance Views

This section covers the following topics:

- Global Dynamic Performance Views
- The V$ Views

Global Dynamic Performance Views

Tuning and performance information for the Oracle database is stored in a set of dynamic performance tables (the V$ fixed views). Each active instance has its own set of fixed views. In a parallel server environment, you can query a global dynamic performance (GV$) view to retrieve the V$ view information from all qualified instances. A global fixed view is available for all of the existing dynamic performance views except for V$ROLLNAME, V$CACHE_LOCK, V$LOCK_ACTIVITY, and V$LOCKS_WITH_COLLISIONS.

The global view contains all the columns from the local view, with an additional column, INST_ID (datatype INTEGER). This column displays the instance number from which the associated V$ information was obtained. You can use the INST_ID column as a filter to retrieve V$ information from a subset of available instances. For example, the query:

```
SELECT * FROM GV$LOCK WHERE INST_ID = 2 or INST_ID = 5
```

retrieves the information from the V$ views on instances 2 and 5.

Each global view contains a GLOBAL hint which creates a parallel query that fetches the contents of the local view on each instance. You can use the GV$ views to return information on groups of instances defined with the OPS_ADMIN_GROUP parameter. Note that a query over G$V views will only return data from instances in instance group g1.

If you have reached the limit of PARALLEL_MAX_SERVERS on an instance and you attempt to query a GV$ view, one additional parallel server process will be spawned for this purpose. This extra process will serve any subsequent GV$ queries until expiration of the PARALLEL_SERVER_IDLE_TIME, at which point the process will terminate. The extra process is not available for any parallel operation other than GV$ queries.

Note: If PARALLEL_MAX_SERVERS is set to zero for an instance, then no additional parallel server process will be allocated to accommodate a GV$ query.

If you have reached the limit of PARALLEL_MAX_SERVERS on an instance and issue multiple GV$ queries, all but the first query will fail--unless

ALLOW_PARTIAL_SN_RESULTS is set. This parameter permits partial results to be returned on excess queries to global dynamic performance views, even if a corresponding parallel server process cannot not be allocated on the instance. In most parallel queries, if a server process could not be allocated this would result in either an error or a sequential execution of the query by the query coordinator. For global views, it may be acceptable to continue running the query in parallel and return the data from the instances which could allocate servers for the query. If the desired behavior is to report an error if server allocation on an instance fails, then the value of ALLOW_PARTIAL_SN_RESULTS should be set to FALSE. If it is acceptable to retrieve results only from instances where server allocation succeeded, then the value of the parameter should be set to TRUE.

See Also: "Specifying Instance Groups" on page 18-22

Oracle8 Reference for restrictions on GV$ views, and complete descriptions of all the parameters and V$ dynamic performance views.

The V$ Views

The following dynamic views are available to monitor a parallel server:

V$BH	GV$BH
V$CACHE	GV$CACHE
V$CACHE_LOCK	
V$CLASS_PING	GV$CLASS_PING
V$DLM_LOCKS	GV$DLM_LOCKS
V$FALSE_PING	GV$FALSE_PING
V$FILE_PING	GV$FILE_PING
V$LOCK_ACTIVITY	
V$LOCK_ELEMENT	GV$LOCK_ELEMENT
V$LOCKS_WITH_COLLISIONS	
V$PING	GV$PING

The V$ views are accessible to the user with SYSDBA privileges. You can grant PUBLIC access to V$ views by running the script MONITOR.SQL, or you can grant individual users SELECT access to new views based on the dynamic views, as described in the “Data Dictionary Reference” chapter of *Oracle8 Administrator’s Guide*.

The VBH, VCACHE, and V$PING views contain statistics about the frequency of PCM lock conversion due to contention between instances. Each row in these views represents one block in the buffer cache of the current instance.

The COUNTER Column

In the V$LOCK_ACTIVITY view, the COUNTER column shows the number of times each type of PCM lock conversion has occurred since the instance started up.

The XNC Column

In the VBH, VCACHE, and V$PING views, the XNC column shows the number of times the PCM lock covering that block has converted from X (exclusive) to NULL at the request of another instance since the block entered the buffer cache. XNC therefore indicates the amount of contention for data. If the PCM lock covers a set of blocks, some or all of the lock conversions could be caused by requests for other blocks in that set.

Each block starts with an XNC value of zero when it first enters the buffer cache. This value is incremented whenever the instance releases the PCM lock covering that block. If a PCM lock covers multiple blocks, they can have different values of XNC because they may enter the buffer cache at different times.

Note: A single block can appear in multiple rows of the VBH, VCACHE, and V$PING views. Each row represents a different copy (version) of the block. Multiple versions created for read-consistent queries appear with the status CR. For tuning purposes, you only need consider the current copy (status XCUR or SCUR) that contains the greatest value of XNC.

When an instance writes a block to disk and reuses that buffer for other data, XNC is reset to zero. If the block returns to the buffer cache while other versions of that block are still in the cache, it starts with the greatest value of XNC for any version of the same block, rather than starting with zero.

Null Values

Null values appear in rows for distributed locks on temporary segments, such as sort blocks. Null values can also appear in some rows of the dynamic views after you create or modify database objects, or after the Oracle Server allocates new extents to database objects; in this case, you should update the views by rerunning CATPARR.SQL.

Use the following procedure to monitor and tune the distributed lock activity in a parallel server.

Querying V$LOCK_ACTIVITY to Monitor Instance Lock Activity

The V$LOCK_ACTIVITY view lists the frequencies of various types of PCM lock conversions for all buffers in the SGA of the current instance; it does not contain information about particular blocks, files, or database objects.

This section covers the following topics:

- Analyzing V$LOCK_ACTIVITY
- Monitoring and Tuning Lock Activity

Analyzing V$LOCK_ACTIVITY

Query the V$LOCK_ACTIVITY view for each instance of a parallel server periodically. The Server Manager command CONNECT *@instance-path* allows you to specify an instance before querying its dynamic performance views. Net8 must be installed to use the CONNECT command for an instance on a remote node. When analyzing the V$LOCK_ACTIVITY view, note that:

- Many PCM locks are initially converted when an instance is started.
- Rapid increases in the number of lock conversions in successive queries (for example, increments of 500+) indicate system contention problems.
- Excessive lock conversions (for example, exceeding 5,000 per minute) indicate contention problems on the system.

For example, the following query could display rows as shown:

```
SELECT * FROM V$LOCK_ACTIVITY;
FROM TO   ACTION                                          COUNTER
---- ---- ------------------------------------------------ -------
NULL S    Lock buffers for read                             5953
NULL X    Lock buffers for write                            1118
S    NULL Make buffers CR (no write)                        6373
S    X    Upgrade read lock to write                        2077
X    NULL Make buffers CR (write dirty buffers)                1
X    S    Downgrade write lock to read (write dirty buffers)  3164
X    SSX  Write transaction table/undo blocks               1007
SSX  NULL Transaction table/undo blocks (write dirty buffers)    2
SSX  S    Make transaction table/undo block available share      1
SSX   X    Rearm transaction table write mechanism          1007
```

See Also: Your platform-specific Oracle documentation for information about connecting with Net8.

Monitoring and Tuning Lock Activity

Use the following procedure to control distributed lock activity.

1. Repeatedly query each instance that you want to monitor with the following SQL statement:

   ```
   SELECT * FROM V$LOCK_ACTIVITY;
   ```

2. If this increases rapidly for any instance, identify the types of lock conversions that are most active in the instance with the following SQL statement:

   ```
   SELECT * FROM V$LOCK_ACTIVITY;
   ```

   Any lock activities from X to a lower mode (such as X to S, X to Null, X to SSX, or S to N) indicate that there is contention among instances for blocks in the buffer cache (blocks are being "pinged") and the instance is releasing locks at the request of other instances. Query the instance repeatedly to find out whether the number of conversions is increasing rapidly.

3. Query the V$LOCK_ACTIVITY view of each instance to identify which instances have the most NULL to S conversions or S to X conversions. These instance are making most of the requests for data that is locked by other instances ("pinging").

   If the pinging occurs mainly between two instances, you should consider letting the applications on those instances run on a single instance.

   If pinging occurs on several instances at approximately the same rate, you may need to tune your PCM lock allocations (see Step 7) or you may have a set of data that the instances access equally, in which case you need to tune your applications (see Step 8).

4. Identify which blocks are pinging by querying the V$PING view of an instance you are monitoring:

   ```
   SELECT * FROM V$PING;
   ```

   You might want to restrict this query with a qualifier to display the blocks that have undergone the most contention; for example:

   ```
   SELECT * FROM V$PING WHERE  FORCED_READS > 10 OR FORCED_WRITES > 10;
   ```

   or:

   ```
   SELECT NAME, KIND, STATUS, SUM(FORCED_READS), SUM(FORCED_WRITES)
   FROM V$PING
   ```

```
GROUP BY NAME, KIND, STATUS
ORDER BY SUM(FORCED_READS);
```

Note: Querying V$BH is faster than querying V$PING or V$CACHE. You can query V$BH to find the block numbers and file numbers of interest. Since V$BH has an OBJD (object number) field, you can join with OBJ$ to find the name of the object, as follows:

```
SELECT O.NAME, BH.*
FROM V$BH BH, OBJ$ O
WHERE O.OBJ# = BH.OBJD
AND (BH.FORCED_READS > 10 OR BH.FORCED_WRITES > 10);
```

5. For blocks that show high rates of pinging, compare FILE# with the datafiles specified in GC_FILES_TO_LOCKS to find out whether their PCM locks cover multiple blocks. If so, also note whether the locks cover blocks in multiple files.
6. If the PCM locks cover multiple blocks, you should determine whether other instances require data from the same block or from different blocks in the same set. To do this, query V$CACHE (or V$BH) in other instances for the BLOCK# that corresponds to a high value of XNC in the instance you are monitoring.
7. If the block does not appear in another instance, there is unnecessary contention (false pinging) because instances that require different blocks are using the same PCM lock for those blocks. To minimize unnecessary contention within one or more datafiles, reduce the number of blocks per lock by allocating more PCM locks to the files with the GC_FILES_TO_LOCKS parameter. If the PCM locks cover multiple files, you can reduce contention by allocating separate sets of locks to individual files.
8. If the same blocks show up in multiple buffer caches, the instances are contending for the same data.

   When multiple instances frequently need to modify data in the same block, you may be able to improve performance by running the applications that require the data on the same instance.

   If the instances modify different rows within the same block, you can re-create the table using the FREELIST GROUPS storage option, then alter the table to allocate extents to particular instances and update selectively to place the data in the appropriate extents.

   For a small table, you can use the PCTFREE and PCTUSED parameters to ensure that a block only contains one row.

If the contention is for rows that are used to generate unique numbers, you can change the applications so that they use SEQUENCE numbers instead of generating their own numbers.

Note: Contention for data blocks and other shared resources does not necessarily have a significant effect on performance. If the response time of your applications is acceptable and you do not anticipate substantial increases in system usage, you may not need to tune your parallel server.

Querying the V$PING View to Detect Pinging

"Pinging" is a catchall term for contention. It includes

- "pings", which are lock down-conversions
- forced reads and forced writes, which constitute the I/O subset of contention. Note that some pings can cause multiple forced writes.

"False pinging" occurs when different instances request different blocks, which happen to map to the same PCM lock. This pinging is unnecessary because it can be reduced by decreasing the granularity of the PCM locks.

Use the following procedure to detect pings.

1. Query V$PING to display summary statistics about lock conversions.

```
SQL> SELECT NAME, FILE#, CLASS#, MAX(XNC) FROM V$PING
     2  GROUP BY NAME, FILE#, CLASS#
     3  ORDER BY NAME, FILE#, CLASS#;

NAME              FILE#     CLASS#       MAX(XNC)
------------    --------     ------      ---------
...
DEPT                 8     1                 492
DEPT                 8     4                  10
EMP                  8     1                3197
EMP                  8     4                  29
...
```

2. Query V$PING again to display the frequency of PCM lock conversions and information for blocks in file 8.

```
SQL> SELECT * FROM V$PING WHERE FILE# = 8;
FILE#   BLOCK#   STAT    XNC  CLASS#   NAME            KIND
------  ------   ----  -----  ------   --------------  -------
     8       98   XCUR    450       1   EMP             TABLE
     8      764   SCUR     59       1   DEPT            TABLE
```

3. Query the EMP table to display the rows contained in block 98. Convert the BLOCK# to a hexadecimal value and compare it to the ROWID. (98 equals 62 in hexadecimal.)

```
SQL> SELECT ROWID, EMPNO, ENAME FROM EMP
     2  WHERE chartorowid(rowid) like '00000062%';
ROWID                 EMPNO  ENAME
------------------  ------  ----------
00000062.0000.0008   12340  JONES
00000062.0000.0008    6491  CLARK
...;
```

Querying V$CLASS_PING, V$FILE_PING, and V$BH

Using dynamic performance views you can separate out, by file, the block classes that are causing most of the contention.

The V$CLASS_PING view helps you identify which class of blocks (such as rollback segments) are being pinged the most. It provides a detailed breakdown by lock conversion type (such as Null to Shared), with read and write physical I/O incurred due to the conversion. Its statistics are cumulative since instance startup. To distribute the contention, you can move different classes of blocks to separate files. For example, you might want to separate rollback segments and datablocks into different files.

The V$FILE_PING view helps you identify which files are being pinged the most. Its statistics are also cumulative since instance startup. To distribute the contention, you can move to other files the objects contained within a heavily pinged file. If a table is heavily pinged, you could partition the table, and place the partitions on separate files.

The V$BH view is a changing snapshot of the buffer cache at any given time. You should periodically sample it, and see how it changes over time. Its statistics are dynamic, not cumulative since startup. V$BH should be sampled periodically to get an idea of ping activity at different points in time during the workload (as stated earlier). You can use V$BH to identify objects in the buffer cache that are undergoing pings, and to determine the forced read/write I/O caused by these pings. V$BH has the object identifier, which can be joined with OBJ$ to get the object name.

Note: You can also monitor the global (GV$) dynamic performance view corresponding to each of these views.

See Also: *Oracle8 Reference* for more information on dynamic views.

Querying the V$WAITSTAT View to Monitor Contention

Use this view to display block contention statistics for resources such as rollback segments and free lists.

This section covers the following topics:

- Monitoring Contention for Blocks in Free Lists
- Monitoring Contention for Rollback Segments

Monitoring Contention for Blocks in Free Lists

Use the following procedure to monitor contention for blocks in free lists.

1. To check the number of waits for free blocks in free lists:

```
SQL> SELECT CLASS, COUNT FROM V$WAITSTAT
     2 WHERE CLASS = 'free list';
CLASS                COUNT
------------------ -------
free list               12
```

2. Compare the COUNT obtained with total number of requests (SUM) for data over the same period.

```
SQL> SELECT SUM(VALUE) FROM V$SYSSTAT
     2 WHERE name IN
     3 ('db block gets', 'consistent gets');
SUM (VALUE)
------------
    12050211
```

3. If the number of waits for free blocks (COUNT) is greater than 1% of the total requests (SUM), consider adding more free lists to tables to reduce contention. To add more free lists to a table, recreate the table with a larger value for the FREELISTS storage parameter. Make the value of FREELISTS equal to the number of users that concurrently insert data into the table.

```
SQL> CREATE TABLE new_emp
     2 STORAGE (FREELISTS 5)
     3 AS SELECT * FROM emp;
Table created.
SQL> DROP TABLE emp;
Table dropped.
SQL> RENAME new_emp TO emp;
Table renamed.
```

Monitoring Contention for Rollback Segments

Use the following procedure to monitor contention for rollback segments.

1. Determine contention for rollback segments with the V$WAITSTAT view.

```
SQL> SELECT CLASS, COUNT
     2 FROM V$WAITSTAT
     3 WHERE CLASS IN ('system undo header',
     4 'system undo block','undo header','undo block');
CLASS               COUNT
------------------ -------
system undo header      12
system undo block       11
undo header             28
undo block               6
```

2. Compare the COUNT obtained with total number of requests (SUM) for data over the same period.

```
SQL> SELECT SUM(VALUE) FROM V$SYSSTAT
     2 WHERE name IN
     3 ('db block gets', 'consistent gets');
SUM (VALUE)
------------
    12050211
```

3. If the number of waits for any class of blocks (COUNT) is greater than 1% of the total requests (SUM), use the CREATE ROLLBACK SEGMENT command to add more rollback segments.

See Also: "Data Dictionary Reference" chapter in *Oracle8 Reference*.

Querying V$FILESTAT and V$DATAFILE to Monitor I/O Activity

Use the V$FILESTAT and V$DATAFILE views to monitor statistics on disk/file access and determine the greatest I/O activity in the system.

1. To determine the number of reads and writes to each database file and the name of each datafile, query the V$FILESTAT and V$DATAFILE views.

```
SQL> SELECT NAME, PHYRDS, PHYWRTS
     2  FROM V$DATAFILE df, V$FILESTAT fs
     3  WHERE df.file# = fs.file#;
NAME                            PHYRDS     PHYWRTS
------------------------   -----------  ----------
/test71/ora_system.dbs             7679        2735
/test71/ora_system1.dbs              32         546
```

2. To determine the number of reads and writes to each non-database file, use an operating system utility, such as the UNIX utility *iostat*. The total I/O for each disk is the total number of reads and writes to all files on the disk.

3. Analyze statistics from the V$FILESTAT view to determine whether disk I/O needs to be distributed to avoid overloading one or more disks. To minimize contention for disk I/O:

   - Separate datafiles and redo log files on different disks.
   - Separate (or stripe) table data on different disks.
   - Separate tables and indexes on different disks.
   - Reduce disk I/O not related to the Oracle server.

4. Analyze the statistics from the V$DATAFILE view to determine whether files need to be placed on separate disks to avoid contention for disk I/O.

   - Place frequently accessed datafiles on separate disks to allow multiple processes to access the data with less contention.
   - Place each set of redo log files on a separate disk with little activity. Information in a redo log file is written sequentially; writing can take place much faster if there is no concurrent activity on the same disk.
   - Stripe a large table to store the table data on separate disks.

Note: Consult your hardware documentation to determine disk I/O limits. Any disks operating at or near full capacity are potential sites for disk contention. For example, 40 or more I/Os per second is excessive for most disks on VMS or UNIX operating systems.

Querying and Interpreting V$SESSTAT and V$SYSSTAT Statistics

The V$SESSTAT and V$SYSSTAT views provide parallel statistics for monitoring contention for various resources including data blocks, rollback segment blocks, and free space lists. This section describes how to query and interpret these statistics.

To display system statistics for analyzing your parallel server (class = 32 or class = 40), issue the following command:

```
SQL> SELECT * FROM V$SYSSTAT
 WHERE CLASS = 32 OR CLASS = 40;

STATISTIC# NAME                                 CLASS    VALUE
---------- ---------------------------------- --- --------
        28 global lock gets (non async)        32   225663
        29 global lock gets (async)            32   169023
        30 global lock get time                32    23199
        31 global lock converts (non async)    32   773052
        32 global lock converts (async)        32    93488
        33 global lock convert time            32    65636
        34 global lock releases (non async)    32   381994
        35 global lock releases (async)        32        0
        36 global lock release time            32    13637
        59 DBWR cross instance writes          40      230
        60 remote instance undo writes         40        0
        61 remote instance undo requests       40      255
        62 cross instance CR read              40       24
        69 next scns gotten without going to DLM 32      0
        73 calls to get snapshot scn kcmgss    32      349
        74 kcmsss waited for batching          32        0
        75 kcmgss reads scn without going to DLM 32      0
        84 hash latch wait gets                40        1
18 rows selected.
```

The following tips will help you interpret statistics obtained from these views.

global lock converts (async)	Divide this number by the V$SYSSTAT statistic “user commits” to calculate the percentage of cache hits.
DBWR cross-instance writes	Equals the number of blocks pinged. For large values, reallocate locks based on V$PING statistics.
remote instance undo writes	A large value may signify pinging activity.
remote instance undo requests	A large value indicates that data modified by this instance if often read by another instance; locate applications (and thus transactions) contending for the same data on the same instance.
cross-instance CR read	This is a slow read because every instance has to write out the block; a large value indicates that the instance is spending too much time waiting on blocks modified by other instances. Evaluate the distribution of locks in the GC_FILES_TO_LOCKS parameter and reallocate to keep the value of this statistic small.
next scns gotten without going to DLM	Divide this value by the total number of SCN gets given by the “user commits” statistic to calculate the percentage of SCN gets satisfied from the cache and thus measure the effectiveness of a parallel server’s SCN cache.
hash latch wait gets	If this value is large or rapidly increasing, increase the number of hash latches.
kcmgss waited for batching	An internal call to get a snapshot might have to wait (for an on-going fetch of a SCN to complete) before contacting the distributed lock manager. This statistic value indicates system load and the number of opportunities that Oracle has to batch a single get-snapshot-SCN with other SCN fetches.

kcmgss reads scn without going to LM	If an internal call (to get a snapshot SCN) waits for an on-going SCN fetch, it may use the SCN acquired by the SCN fetch, thus avoiding overhead in using the lock manager. The ratio of "kcmgss reads scn without going to LM" and "kcmgss waited for batching" indicates the effectiveness of the parallel server's SCN batch algorithm.

See Also: *Oracle8 Reference* for definitions of these statistics.

Oracle Server Manager User's Guide descriptions of the MONITOR STATISTICS CACHE display for information about monitoring contention for various kinds of blocks.

21

Backing Up the Database

Those behind cried "Forward!"
And those before cried "Back!"

Thomas Babington, Lord Macaulay: *On Frederic The Great*

To protect your data, you should archive the online redo log files and periodically back up the datafiles. You should also back up the control file for your database and the parameter files for your instances. This chapter discusses:

- Choosing a Backup Method
- Archiving the Redo Log Files
- Checkpoints and Log Switches
- Backing Up the Database

Oracle Parallel Server supports all of the backup features of Oracle in exclusive mode, including both open and closed backup of either an entire database or individual tablespaces.

Choosing a Backup Method

In Oracle8 you can perform backup and recovery operations using two different methods:

- Using Recovery Manager
- Using the operating system (existing method)

The information provided in this chapter is true for both methods, unless specified otherwise.

Note: To avoid confusion between online and offline datafiles and tablespaces, this documentation uses the terms “open” and “closed” to indicate whether a database is available or unavailable during a backup. The term “whole backup” or “database backup” indicates that all datafiles and control files have been backed up. “Full” and “incremental” backups refer only to particular types of backup provided by Recovery Manager.

See Also: *Oracle8 Backup and Recovery Guide* for a complete discussion of backup and recovery operations and terminology.

Archiving the Redo Log Files

This section explains how to archive the redo log files for each instance of a parallel server:

- Archiving Mode
- Automatic or Manual Archiving
- Archive File Format and Destination
- Redo Log History in the Control File
- Backing Up the Archive Logs

Archiving Mode

Oracle provides two archiving modes: ARCHIVELOG mode and NOARCHIVELOG mode. With Oracle in ARCHIVELOG mode, the instance must archive its redo logs as they are filled—before they can be overwritten. The logs can thus be recovered in the event of media failure. In ARCHIVELOG mode, you can make both open and closed backups. In NOARCHIVELOG mode, you can only make closed backups.

Note that archiving is a per-instance operation which can be handled in one of two ways:

- Each instance on a parallel server can archive its own redo log files.
- Alternatively, one or more instances can archive the redo log files manually for all instances, as described in the following section.

See Also: "Open and Closed Database Backups" on page 21-12.

Automatic or Manual Archiving

Archiving can be performed automatically or manually for a given instance, depending on the value you set for the LOG_ARCHIVE_START initialization parameter.

- With LOG_ARCHIVE_START set to TRUE, Oracle automatically archives redo logs as they fill.
- With LOG_ARCHIVE_START set to FALSE, Oracle waits until you instruct it to archive.

For Oracle Parallel Server, each instance can set this parameter differently. Thus, for example, you can manually use SQL commands or Server Manager to have instance 1 archive the redo log files of instance 2, if instance 2 has LOG_ARCHIVE_START set to FALSE.

Automatic Archiving

The ARCH background process performs automatic archiving upon instance startup when LOG_ARCHIVE_START is set to TRUE. With automatic archiving, online redo log files are copied only for the instance that performs the archiving.

In the case of a closed thread, the archiving process in the active instance performs the log switch and archiving for the closed thread. This is done when log switches are forced on all threads to maintain roughly the same range of SCNs in the archived logs of all enabled threads.

Manual Archiving

When LOG_ARCHIVE_START is set to FALSE, you can perform manual archiving in one of the following ways:

- using the ARCHIVE LOG clause of the ALTER SYSTEM command (in SQL)
- enabling automatic archiving (by using the SQL command ALTER SYSTEM ARCHIVE LOG START, or using Server Manager)

Manual archiving is performed by the user process that issues the archiving command; it is not performed by the instance's ARCH process.

ALTER SYSTEM ARCHIVE LOG Options for Manual Archiving

ALTER SYSTEM ARCHIVE LOG manual archiving options include:

ALL	All online redo log files that are full but have not been archived.
CHANGE	The lowest system change number (SCN) in the online redo log file.
CURRENT	The current redo log of every enabled thread.
GROUP *integer*	The group number of an online redo log.
LOGFILE *'filename'*	The filename of an online redo log file in the thread.
NEXT	The next full redo log file that needs to be archived.
SEQ *integer*	The log sequence number of an online redo log file.
THREAD *integer*	The thread containing the redo log file to archive (defaults to the thread number assigned to the current instance).

You can use the THREAD option of ALTER SYSTEM ARCHIVE LOG to archive redo log files in a thread associated with an instance other than the current instance.

See Also: *Oracle8 Reference* for information about the syntax of the ALTER SYSTEM ARCHIVE LOG statement.
"Archiving Redo Information" chapter in *Oracle8 Administrator's Guide* for more information about manual and automatic archiving.
Oracle8 Backup and Recovery Guide for more information about manual and automatic archiving.
"Forcing a Log Switch" on page 21-10 regarding threads and log switches.

Archive File Format and Destination

Archived redo logs are uniquely named as specified by the LOG_ARCHIVE_FORMAT parameter. This operating-system specific format can include text strings, one or more variables, and a filename extension. LOG_ARCHIVE_FORMAT can have the following variables. (Table 21–1 assumes that LOG_ARCHIVE_FORMAT= arch%*parameter*, and the upper bound for all parameters is 10 characters:)

Table 21–1 Archived Redo Log Filename Format Parameters

Parameter	Description	Example
%T	thread number, left-zero-padded	arch0000000001
%t	thread number, not padded	arch1
%S	log sequence number, left-zero-padded	arch0000000251
%s	log sequence number, not padded	arch251

The thread parameters %t and %T are used only with the Parallel Server Option. For example, if the instance associated with redo thread number 7 sets LOG_ARCHIVE_FORMAT to LOG_%s_T%t.ARC, then its archived redo log files are named:

```
LOG_1_T7.ARC
LOG_2_T7.ARC
LOG_3_T7.ARC
...
```

Note: Always specify thread and sequence number in archive log file format for easy identification of the redo log file.

See Also: Your Oracle system-specific documentation for default log archive format and destination.
"Archiving Redo Information" chapter in *Oracle8 Administrator's Guide* for information about specifying the archived redo log filename format and destination.
"Recovery Structures" chapter in *Oracle8 Concepts*.

Redo Log History in the Control File

You can use the MAXLOGHISTORY clause of the CREATE DATABASE or CREATE CONTROLFILE command to enable the control file to keep a history of the redo log files that a parallel server has filled. After creating the database, it is only possible to increase or decrease the log history by creating a new control file. Note that using CREATE CONTROLFILE destroys all log history in the current control file.

The MAXLOGHISTORY option specifies how many entries can be recorded in the archive history. Its default value is operating-system specific. If MAXLOGHISTORY is set to a value greater than zero, then whenever an instance switches from one online redo log file to another, its LGWR process writes the following data to the control file.

- thread number
- log sequence number
- low system change number (SCN)
- low SCN timestamp
- next SCN (that is, the low SCN of the next log in sequence)

Note: LGWR writes log history data to the control file during a log switch, not when a redo log file is archived.

Log history records are small, and are overwritten in a circular fashion when the log history exceeds the limit set by MAXLOGHISTORY.

During recovery, Server Manager prompts for the appropriate file names. Recovery Manager automatically restores the redo logs it requires. You can use the log history to reconstruct archived log file names from an SCN and thread number, for automatic media recovery of a parallel server that has multiple threads of redo. An Oracle instance that accesses the database in exclusive mode with only one thread enabled does not need the log history--but the log history is useful when multiple threads are enabled, even if only one thread is open.

You can query the log history information from the V$LOG_HISTORY view. When you are using Server Manager, V$RECOVERY_LOG also displays information about archived logs needed to complete media recovery; this is derived from information in the log history records.

Multiplexed redo log files do not require multiple entries in the log history. Each entry identifies a group of multiplexed redo log files, not a particular filename.

See Also: Your Oracle system-specific documentation for the default MAXLOGHISTORY value.

"Restoring and Recovering Redo Log Files" on page 22-18 for Server Manager prompts during recovery.

Backing Up the Archive Logs

Archive logs are generally only accessible by the node on which they were created. In a parallel server environment you have two backup options:

- have each node back up its own archive logs
- move the archive logs to one node, and then back them up

Using O/S utilities, you can manually implement either solution.

Backing Up Archive Logs with Recovery Manager

Recovery Manager can automatically enable each node to back up its own archive logs. However, if you wish to move the logs you must do so manually and then use the appropriate rman catalog and change commands to reflect the movement of files. Once Recovery Manager has been informed of the changes you have made, it can back up archive logs from the single node.

If you are using multiple nodes to back up your archive logs, note that when Recovery Manager compiles the list of logs to be archived, it must be able to check that the archived logs exist. To do this it must be able to read the headers of all archived logs on all nodes.

Each node can then back up the archived logs it has created. In the example below, because the initial target database is node 1 (on the rman command line), you must ensure that node 1 is able to read the headers of the archived logs (even those produced by node 2).

```
rman target internal/knl@node1 rcvcat rman/rman@rcat

run {
  allocate channel t1 type 'SBT_TAPE' connect 'internal/knl@node1';
  allocate channel t2 type 'SBT_TAPE' connect 'internal/knl@node2';
  backup
    filesperset 10
    format 'al_%t_%s_%p'
    (archivelog until time 'SYSDATE' thread 1 delete input channel t1)
    (archivelog until time 'SYSDATE' thread 2 delete input channel t2);
}
```

Restoring Archive Logs with Recovery Manager

By default, rman will restore archive logs to the *log_archive_dest* of the instances it connects to. If you are using multiple nodes to restore and recover, this means the archive logs may be restored to any of the nodes doing the restore / recover. The node which will actually read the restored logs and perform the roll-forward is the target node initially connected to. For recovery to use these logs, you must ensure that the logs are readable from that node.

Checkpoints and Log Switches

This section discusses:

- Checkpoints
- Log Switches
- When Checkpoints Occur Automatically
- Forcing a Checkpoint
- Forcing a Log Switch
- Forcing a Log Switch on a Closed Thread

Checkpoints

A checkpoint causes modified datablocks held in the SGA buffer cache to be written to disk. A *global checkpoint* causes all instances to write modified datablocks to disk. An *instance checkpoint* causes one instance to write modified datablocks to disk. Lastly, a *datafile checkpoint* causes all instances to write the modified datablocks for a single datafile to disk. During a checkpoint, the DBWR process of an instance writes the modified datablocks to disk only for that instance.

Because all database changes up to the checkpoint are written to the datafiles, redo log entries before the checkpoint are not needed for instance recovery.

For a single instance with exclusive access to a database, checkpoints determine the maximum recovery time after instance failure, because you only need to recover changes made after the last checkpoint.

For multi-instance systems, checkpoints determine the maximum recovery time for each instance. Since instances usually have different checkpoint intervals, instance failures on different nodes generally require different recovery times.

Log Switches

A *log switch* is the point in time when an instance's LGWR process ceases writing redo log entries in one online redo log file and begins writing redo log entries in the next available redo log file.

The intervals between checkpoints for each instance are determined by the frequency of log switches, which depend on the redo log file size and the amount of redo data generated, and by the values of the parameters LOG_CHECKPOINT_TIMEOUT and LOG_CHECKPOINT_INTERVAL. Additional checkpoints and log switches can be forced by various SQL statements and Server Manager commands, and a parallel server can force a log switch so that an online redo log file can be archived.

When Checkpoints Occur Automatically

An instance performs a checkpoint under any of the following circumstances:

- When the number of redo log blocks written by that instance reaches the limit specified by the LOG_CHECKPOINT_INTERVAL initialization parameter.
- At periods specified by the LOG_CHECKPOINT_TIMEOUT. initialization parameter.
- At log switch time, which occurs when the instance's current online redo log file is full, when Oracle forces a log switch to archive a redo log file, or when the database administrator forces a log switch by one of the following methods:
  - ALTER SYSTEM SWITCH LOGFILE
  - ALTER SYSTEM ARCHIVE LOG CURRENT
- When the database administrator issues the ALTER SYSTEM CHECKPOINT statement
- When the instance shuts down using the NORMAL or IMMEDIATE option.
- When backup of a tablespace in an open database begins, a partial checkpoint is performed by every instance for the datafiles in that tablespace.

See Also: "Forcing a Log Switch" on page 21-10.

Forcing a Checkpoint

The SQL statement ALTER SYSTEM CHECKPOINT explicitly forces Oracle to perform a checkpoint for either the current instance or all instances. Forcing a checkpoint ensures that all changes to the database buffers are written to the datafiles on disk.

The GLOBAL option of ALTER SYSTEM CHECKPOINT is the default. It forces all instances that have opened the database to perform a checkpoint. The LOCAL option forces a checkpoint by the current instance.

A global checkpoint is not finished until all instances that require recovery have been recovered. If any instance fails during the global checkpoint, however, the checkpoint might complete before that instance has been recovered.

To force a checkpoint on an instance running on a remote node, you can change the current instance with the Server Manager command CONNECT.

Note: You need the ALTER SYSTEM privilege to force a checkpoint.

See Also: "Specifying Instances" on page 18-16 for information on specifying a remote node.

Forcing a Log Switch

A parallel server can force a log switch for any instance that fails to archive its online redo log files for some period of time, either because the instance has not generated many redo entries or because the instance has shut down. This prevents an instance's redo log, known as a *thread* of redo, from remaining unarchived for too long. If media recovery is necessary, the redo entries used for recovery are always reasonably recent.

For example, after an instance has shut down, another instance can force a log switch for that instance so that its current redo log file can be archived.

Note: The initialization parameters LOG_CHECKPOINT_TIMEOUT and LOG_CHECKPOINT_INTERVAL can force an inactive instance to perform checkpoints, but these do not force the instance to perform log switches.

The SQL statement ALTER SYSTEM SWITCH LOGFILE forces the current instance to begin writing to a new redo log file, regardless of whether the current redo log file is full.

Forcing a log switch also forces a checkpoint. Oracle returns control to you immediately after beginning the log switch, rather than waiting until the checkpoint is finished.

To force all instances to perform log switches, known as a *global log switch*, use the SQL statement ALTER SYSTEM ARCHIVE LOG CURRENT *omitting* the THREAD keyword. After you issue this statement, Oracle waits until all online redo log files are archived before returning control to you. Use this statement to force a single instance to perform a log switch and archive its online redo log files by specifying the THREAD keyword.

In Server Manager, you can use the Instance Force Log Switch option for the current instance only. There is no global option for forcing a log switch in Server Manager. You may want to force a log switch so that you can archive, drop, or rename the current redo log file.

Note: You need the ALTER SYSTEM privilege to force a log switch.

See Also: "Redo Log Files" on page 6-3 for more information about threads.

Forcing a Log Switch on a Closed Thread

You can force a closed thread to complete a log switch while the database is open. This is useful if you want to drop the current log of the thread. This procedure does not work on an open thread (including the current thread), even if the instance that had the thread open is shut down. For example, if an instance aborted while the thread was open, you could not force the thread's log to switch.

To force a log switch on a closed thread, manually archive the thread, using the Begin Manual Archive dialog box of Server Manager or the SQL command ALTER SYSTEM with the ARCHIVE LOG option. For example:

```
ALTER SYSTEM ARCHIVE LOG GROUP 2;
```

To archive a closed redo log group manually that will force it to log switch, you must connect with SYSOPER or SYSDBA privileges.

See Also: *Oracle8 Administrator's Guide* for information on connecting with SYSDBA or SYSOPER privileges.

Backing Up the Database

This section covers backup operation issues in an Oracle Parallel Server environment. It covers the following topics:

- Open and Closed Database Backups
- Recovery Manager Backup Issues
- Operating System Backup Issues

Open and Closed Database Backups

All backup operations can be performed from any node of a parallel server. Open backups allow you to back up all or part of the database while it is running. Users can access the database and update data in any part of the database during an open backup. With a parallel server you can make open backups of multiple tablespaces simultaneously from different nodes. An open backup includes copies of one or more datafiles and the current control file. Subsequent archived redo log files or incremental backups are also necessary to allow recovery up to the time of a media failure.

When using the operating system, closed backups are taken while the database is closed. When using Recovery Manager, an instance must be started and mounted, but not open, in order to do a closed backup. Before you make a closed backup, you must therefore shut down *all* instances of a parallel server. While the database is closed, you can back up its files in parallel from different nodes. A closed whole database backup includes copies of all datafiles and the current control file.

If you archive redo log files, a closed backup allows recovery up to the time of a media failure. In NOARCHIVELOG mode, full recovery is not possible since a closed backup only allows restoration of the database to the point in time of the backup.

> ***Warning:*** Do not use operating-system utilities to back up the control file in ARCHIVELOG mode, unless you are performing a whole, closed backup.
>
> Never erase, reuse, or destroy archived redo log files until you have done another whole backup (preferably two whole backups), either open or closed.

See Also: *Oracle8 Backup and Recovery Guide.*

"Database Backup" and "Database Recovery" in *Oracle8 Concepts.*

Recovery Manager Backup Issues

Preparing for Snapshot Control Files in Recovery Manager

In an Oracle Parallel Server environment, you must prepare for snapshot control files before you perform a backup using Recovery Manager.

Any node making a backup may need to create a snapshot control file. Therefore, on all nodes used for backup, you must ensure the existence of the directory to which such a snapshot control file will be written.

For example, to specify that the snapshot control file should be written to the file **/oracle/db_files/snapshot/snap_prod.cf**, you would enter:

```
SET SNAPSHOT CONTROLFILE TO '/oracle/db_files/snapshot/snap_prod.cf';
```

You must then ensure that the directory **/oracle/db_files/snapshot** exists on all nodes from which you perform backups.

It is also possible to specify a raw device destination for a snapshot control file, which like other datafiles in an OPS environment will be shared across all nodes in the cluster.

Performing an Open Backup Using Recovery Manager

See the *Oracle8 Backup and Recovery Guide* for complete information on open backups using Recovery Manager.

If you are also backing up archive logs, then issue an ALTER SYSTEM ARCHIVE LOG CURRENT statement after the backup has completed. This ensures that you have all redo to make the files in this backup consistent.

The following sample script distributes datafile and archive log backups across two instances in a parallel server environment. It assumes:

- there are more than 20 files in the database
- 4 tape drives available, two on each node
- redo thread 1 is used by the instance on node 1
- redo thread 2 is used by the instance on node 2
- the archive log files produced by thread 2 are readable by node1

The sample script is as follows:

```
run {
  allocate channel node1_t1 type 'SBT_TAPE' connect 'internal/knl@node1';
  allocate channel node1_t2 type 'SBT_TAPE' connect 'internal/knl@node1';
  allocate channel node2_t3 type 'SBT_TAPE' connect 'internal/knl@node2';
  allocate channel node2_t4 type 'SBT_TAPE' connect 'internal/knl@node2';
  backup
   filesperset 6
   format 'df_%t_%s_%p'
   (database);
  sql 'alter system archive log current';
  backup
    filesperset 10
    format 'al_%t_%s_%p'
    (archivelog until time 'SYSDATE' thread 1 delete input channel node1_t1)
    (archivelog until time 'SYSDATE' thread 2 delete input channel node2_t3);
  }
```

Operating System Backup Issues

Beginning and Ending an Open Backup Using Operating System Utilities

When using the operating system method, you begin an open backup of a tablespace at one instance and can end the backup at the same instance or another instance. For example:

```
ALTER TABLESPACE tablespace BEGIN BACKUP;/* Instance X */
Statement processed.

....operating system commands to copy datafiles...
....copy completed...

ALTER TABLESPACE tablespace END BACKUP;/* Instance Y */
Statement processed.
```

Warning: If the ALTER TABLESPACE ... BEGIN BACKUP command is not issued or does not complete before an operating system backup of the tablespace is started, then the backed up datafiles are not useful for subsequent recovery operations. Attempting to recover such a backup is risky and can cause errors that result in inconsistent data.

It does not matter which instance issues each of these statements, but they must be issued whenever you make an open backup. The BEGIN BACKUP option has no effect on users' access to the tablespace.

For an open backup to be usable for complete or incomplete media recovery, you must retain all archived redo logs spanning the period of time between the execution of the BEGIN BACKUP command and the recovery end-point.

After making an open backup, you can force a global log switch by using ALTER SYSTEM ARCHIVE LOG CURRENT. This statement archives all online redo log files that need to be archived, including the current online redo log files of all enabled threads and closed threads of any instance that shut down without archiving its current redo log file.

See Also: *Oracle8 SQL Reference* for a description of the BEGIN BACKUP and END BACKUP clauses of the ALTER TABLESPACE command.

Performing an Open Backup Using Operating System Utilities

The following steps are recommended if you are using operating system utilities to perform an open backup in a parallel server environment.

1. Before starting the open backup, issue the ALTER SYSTEM ARCHIVE LOG CURRENT command.

   This switches and archives the current redo log file for *all* threads in a parallel server environment, even those threads that are not currently up.

2. Issue the ALTER TABLESPACE *tablespace* BEGIN BACKUP command.
3. Wait for the ALTER TABLESPACE command to successfully complete.
4. In the operating-system environment, issue the appropriate commands to back up the datafiles for the tablespace.
5. Wait for the operating-system backup to successfully complete.
6. Issue the ALTER TABLESPACE *tablespace* END BACKUP command.
7. Back up the control files with ALTER DATABASE BACKUP CONTROLFILE TO *filename*.

For an added measure of safety, back up the control file to a trace file with the ALTER DATABASE BACKUP CONTROLFILE TO TRACE NORESETLOGS command, then identify and back up that trace file.

If you are also backing up archive logs, then issue an ALTER SYSTEM ARCHIVE LOG CURRENT statement after END BACKUP. This ensures that you have all redo to roll to the end backup marker.

22

Recovering the Database

This chapter describes Oracle recovery features on a parallel server. It covers the following topics:

- Overview
- Client-side Application Failover
- Recovery from Instance Failure
- Recovery from Media Failure
- Parallel Recovery

Overview

This chapter discusses client-side application failover, and three types of recovery:

Table 22–1 Types of Recovery

Type of Recovery	Definition
Instance failure	Occurs when a software or hardware problem prevents an instance from continuing work.
Media failure	Occurs when the storage medium for Oracle files is damaged. This usually prevents Oracle from reading or writing data.
Parallel recovery	For Recovery Manager, restore and application of incremental backups are parallelized using channel allocate. Application of redo (whether it is done by Recovery Manager or by Server Manager) is determined by the RECOVERY_PARALLELISM parameter.

Client-side Application Failover

This section covers the following topics:

- What Is Application Failover?
- How to Configure Application Failover
- Planned Shutdown and Dynamic Load Balancing
- Special Failover Topics
- Failover Restrictions

Note: To use application failover, you must have the Oracle8 Enterprise Edition and the Parallel Server Option. For more information, please refer to *Getting to Know Oracle8 and the Oracle8 Enterprise Edition*.

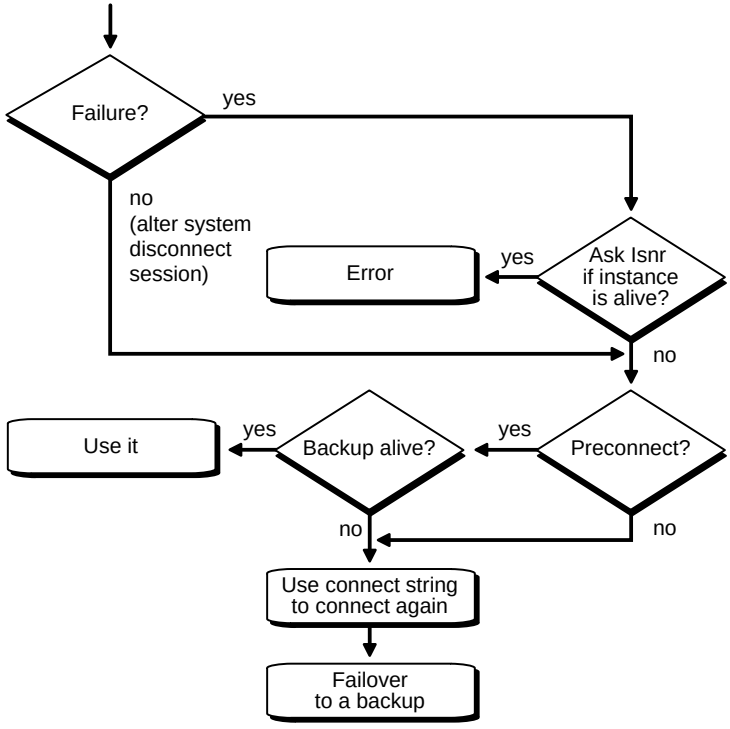
What Is Application Failover?

On Oracle Parallel Server, application failover is the ability of the application to automatically reconnect if the connection to the database is broken. Any active transaction will be rolled back, but the new database connection will otherwise be identical to the original one. This is true only if the connection was lost because the instance died due to ALTER SYSTEM DISCONNECT SESSION.

Often a client really wants to connect to an application rather than to a database instance. With application failover the client sees no loss of connection while there is a surviving instance serving the application, and is normally able to continue SELECTs started before the instance failed. The DBA can control which applications run on various instances, and create a failover order for each application.

Note that after failover, only select or fetch calls are replayed; all other calls will receive an error message. Also, if the client's process dies but the instance does not, then failover will not occur.

Figure 22–1 Failover Flow Chart



How to Configure Application Failover

The DBA can configure the connect string for the application at the names server, or put it in the TNSNAMES.ORA file. Alternatively, the connect string can be hard coded in the application. For each application, the names server provides information about the listener, the instance group, and the failover mode. The connect string *failover_mode* field specifies the type and method of failover. For more information on syntax, please refer to the *Net8 Administrator's Guide*.

TYPE: Failover Mode Functionality Options

The client's failover functionality is determined by the "TYPE" keyword in the connect string. The choices are:

SELECT	This allows users with open cursors to continue fetching on them after failure. However, this mode involves overhead on the client side in normal select operations, so the user is allowed to disable select failover.
SESSION	This fails over the session; that is, if a user's connection is lost, a second session is automatically created for the user on the backup. This type of failover does not attempt to recover selects.
NONE	This is the default, in which no failover functionality is used. This can also be explicitly specified to prevent failover from happening.

METHOD: Failover Mode Performance Options

Improving the speed of application failover often requires putting more work on the backup instance. The DBA can use the METHOD keyword in the connect string to configure the BASIC or PRECONNECT performance options.

BASIC	Establish connections at failover time. This option requires almost no work on the backup server until failover time.
PRECONNECT	Pre-establish connections. This provides faster failover but requires that the backup instance be able to support all the connections from every supported instance.

Failover and Listeners

For application failover to work correctly, the connect string used to attach initially must also go to a valid instance at failover time. This means that the listener to which the client connects at failover time must be able to either establish a connection with the right instance, or refuse the connection so that the client can chose another listener (via a *description_list* or an *address_list* in the connect string).

There are several ways of achieving this. The approach to use depends on the exact situation, and the role that the listeners play in connecting. The solutions are described in this section.

Failover with the Multi-threaded Server. To use application failover with the multi-threaded server, you must set the MTS_SERVICE parameter to the same value for every instance. This value must match the value of the SID in the connect string. If there is more than one listener for the database, then the dispatchers must be configured individually with the (LISTENERS=) clause of the MTS_DISPATCHER parameter.

Also note that the techniques described for dedicated servers will also work for multi-threaded servers as long as the mts_service is the same as the instance SID.

Several dynamic performance views are available to help you tune the MTS dispatcher:

V$DISPATCHER_RATE
V$DISPATCHER_RATE_CURRENT
V$DISPATCHER_RATE_MAXIMUM
V$DISPATCHER_RATE_AVERAGE

These views contain statistics which show message rates and buffer rates. Using these views you can adjust the relationship between the number of events in a loop and how busy the dispatcher is, so as to minimize overhead. Note that the Net8 parameter SDUSIZE determines how large a message you can send at one time.

See Also: *Oracle8 Reference* for details on views and statistics.

Failover with the Connection Load Balancer. The Net8 generic listener can support plug-in load balancing. (This is not required for single instance failover.) On some platforms the Oracle Connect Load Balancing (CLB) module can perform this function. The CLB allows the SID portion of the connect string to be an instance group.

An example of an explicit connect string is:

```
(DESCRIPTION=
(ADDRESS=(PROTOCOL=tcp)(HOST=westreg)(port=1512))
(CONNECT_DATA=(SID=OE)(SERVER=SHARED)
(FAILOVER_MODE=(TYPE=SELECT)(METHOD=BASIC))))
```

The first line announces that this is a connect string. The second line provides the listener's address. The third line begins with the instance group definition. This is the set of instances on which this application is served. The third line concludes by specifying a connection via MTS. The remaining line provides the new data about the failover mode. The TYPE field specifies the functionality option, and the METHOD field specifies the method.

Note that you can specify an instance group in the connect string.

See Also: "Specifying Instance Groups" on page 18-22
Oracle Enterprise Manager documentation
Oracle platform-specific documentation regarding support for CLB and other load balancing packages.

Failover with Dedicated Servers. To accomplish failover with dedicated servers, there must be multiple listeners (unless all the instances are on the same machine). This will usually mean that the connect string will contain either a description list or an address list. For failover to work properly, the listeners must know whether or not the instance is alive. This is accomplished by having the instances register themselves with their listeners, and the LISTENER.ORA file not contain the instance's SID. The instance registration is configured via the LOCAL_LISTENERS parameter (see documentation on this parameter). By using this technique, an address list may be employed to achieve a "primary/backup" relationship between two (or more) instances. If the first instance is not up the connection to the first listener in the *address_list* will fail and the client will proceed to try the second. Likewise, a *description_list* will provide the semantics that connections are spread evenly over the living instances. See the documentation on *address_lists* and *description_lists* for more details.

Preconnect Connect Strings. When using preconnect, the client must make a connection to a backup while the primary instance is still up. The DBA should provide a connect string to use as a backup. In this way, the DBA can be sure that the backup instance will be different from the primary instance. This backup connect string is provided with the BACKUP keyword in the FAILOVER_MODE portion of the connect string. This can be either an explicit address or an alias to look up. If no backup is provided, failover will use the original connect string. This will work, but some percent of clients will get the same backup as primary, and will have to reconnect at failover time.

In the following example, one instance is running on each node. The database is called i1 and the ORACLE_SIDs are i11 and i12. There is one listener on each node listening for the local instance and the instance groups are as follows:

```
group g1=(i11)
group g2=(i12)
group gc=(i11,i12)
```

In the TNSNAMES.ORA file you would enter:

```
 i11 = (DESCRIPTION=
        (ADDRESS=(PROTOCOL=TCP)(HOST=host1)(PORT=1521))
        (CONNECT_DATA=(SID=i11)(SERVER=DEDICATED)
           (FAILOVER_MODE=(TYPE=select)(METHOD=preconnect)
               (BACKUP=i12))
        )
      )
i12 = (DESCRIPTION=
        (ADDRESS=(PROTOCOL=TCP)(HOST=host2)(PORT=1521))
        (CONNECT_DATA=(SID=i12)(SERVER=DEDICATED)
             (FAILOVER_MODE=(TYPE=select)(METHOD=preconnect)
               (BACKUP=i11))
        )
      )
```

This way, each client uses either connect string i11 or i12, and establishes a backup connection using the other connect string. So if a client uses connect string i11, it will initially connect to instance 1, and use instance 2 if instance 1 fails.

Failover Fields in V$SESSION

The view V$SESSION has the following fields related to failover:

FAILED_OVER	TRUE if using the backup, otherwise FALSE
TYPE	One of SELECT, SESSION, or NONE
METHOD	Either BASIC or PRECONNECT

Planned Shutdown and Dynamic Load Balancing

This section explains how the DBA can bring down an instance or a session. For complete syntax of available SQL statements, see the *Oracle8 Server SQL Reference.*

Shutting Down an Instance after Current Transactions

The TRANSACTIONAL option to the SHUTDOWN command enables the DBA to do a planned shutdown of one instance while minimally interrupting clients. This option will wait for ongoing transactions to complete, and is useful for installing patch releases, or other times when the instance must be brought down without interrupting service.

While waiting, no client can start a new transaction on the instance. Clients will be disconnected if they try to start a transaction, and this will trigger failover, if it is enabled. When the last transaction completes the primary instance performs a SHUTDOWN IMMEDIATE. If failover is enabled, SHUTDOWN TRANSACTIONAL normally prevents any clients from losing work, while not requiring all users to log off first, as a SHUTDOWN NORMAL would. Clients are automatically reconnected to the instance.

Disconnecting a Session after the Current Transaction

The ALTER SYSTEM DISCONNECT SESSION POST_TRANSACTION statement disconnects a session on the first call after its current transaction has been finished. The application will failover automatically.

```
ALTER SYSTEM DISCONNECT SESSION 'sid,serial#' POST_TRANSACTION
```

where

sid	is the system identifier
serial#	is the session serial number, from the V$SESSION view

The POST_TRANSACTION option works well with failover as a way for the DBA to control load. If one instance is overloaded, the DBA can manually disconnect a

group of sessions using this option. Since the option guarantees that there is no transaction at the time the session is disconnected, the user should never notice the shift, except for a slight delay executing the next command following the disconnect.

Special Failover Topics

This section describes multiple user handles and callbacks.

Multiple User Handles

Failover is supported for multiple user handles. In OCI the server context handle and the user handle are decoupled. You can have multiple user handles related to the server context handle, and multiple users can thus share the same connection to the database.

If the connection is destroyed, then every user associated with that connection will be failed over. But if a single user process is destroyed then failover does not occur because the connection is still there. Failover will not reauthenticate migrateable user handles.

See Also: *Programmer's Guide to the Oracle Call Interface, Volume I: OCI Concepts,* and *Programmer's Guide to the Oracle Call Interface, Volume II: OCI Reference*

Failover Callback

Frequently failure of one instance and failover to another takes some time. Because of this delay, you may want to inform the user that failover is in progress, and request that the user stand by. Additionally, the session on the initial instance may have received some ALTER SESSION commands. These will not be automatically replayed on the second instance. You may want to ensure that these commands will be replayed on the second instance.

To address these problems, you can register a callback function. Failover will call the callback function several times during the course of reestablishing the user's session. The first call occurs when instance connection loss is first detected, so the application can inform the user of upcoming delay. If failover is successful, the second call occurs when the connection is reestablished and usable. At this time the client may wish to replay ALTER SESSION statements and inform the user that failover has occurred. If failover is unsuccessful, then the callback will be called to inform the application that failover will not take place. Additionally, the callback will be called each time a user handle besides the primary handle is reauthenticated on the new connection.

See Also: *Oracle Call Interface Programmer's Guide*

Tuning Failover Performance

The elapsed time of failover includes instance recovery as well as time needed to reconnect to the database. For best performance of failover, therefore, you should tune instance recovery by having frequent checkpoints, and so on.

Performance can also be improved by having multiple listeners, using the multi-threaded server or dedicated servers. Note in particular that MTS connections tend to be much faster than connections via dedicated servers.

The number of users trying to failover at the same time also affects failover performance. Failover occurs when users attempt to perform actions. In some applications, many users may be logged in, but few may be performing work at any given time. In such as case, if no instance recovery is necessary and only a few users are failing over at any given time, failover may be very fast. Thus the amount of effort you may decide to put in to tuning failover performance will probably be related to the number of concurrent users you expect. In a three-tier application design, for example, it might be best to have few connections; for faster failover each connection could have several sessions associated with it.

Failover Restrictions

When a connection is lost, you will see the following effects:

- All PL/SQL package states on the server will be lost at failover time.
- ALTER SESSION statements will be lost.
- If failover occurs when a transaction is in process, then each subsequent call will cause an error message until the user issues an OCITransRollback call. Then an OCI success with information message is issued. Be sure to check this informational message to see if you must perform any additional operations.
- Continuing work on failed over cursors may cause an error message.
- If the first command after failover is not a SQL SELECT or OCIStmtFetch statement, an error message will result.
- Failover only takes effect if the application is programmed using OCI Release 8.0.
- At failover time, any queries that are in progress will be reissued and processed again from the beginning. This may result in the next fetch taking a long time, if the original query took a long time.

See Also: *Oracle Call Interface Programmer's Guide*

Recovery from Instance Failure

The following sections describe the recovery performed after failure of instances accessing the database in shared mode.

- Single-node Failure
- Multiple-node Failure
- Incremental Checkpointing
- Access to Datafiles for Instance Recovery
- Freezing the Database for Instance Recovery
- Phases of Oracle Instance Recovery

After instance failure, Oracle uses the online redo log files to perform automatic recovery of the database. For a single instance running in exclusive mode, instance recovery occurs as soon as the instance starts up again after it has failed or shut down abnormally.

When instances accessing the database in shared mode fail, online instance recovery is performed automatically. Instances that continue running on other nodes are not affected, as long as they are reading from the buffer cache. If instances attempt to write, the transaction will stop. All operations to the database are suspended until cache recovery of the failed instance is complete.

See Also: *Oracle8 Backup and Recovery Guide*.

Single-node Failure

A parallel server performs instance recovery by coordinating recovery operations through the SMON processes of the other running instances. If one instance fails, the SMON process of another instance notices the failure and automatically performs instance recovery for the failed instance.

Instance recovery does not include restarting the failed instance or any applications that were running on that instance. Applications that were running may continue by failover, as described in "Client-side Application Failover" on page 22-2.

When one instance performs recovery for another instance that has failed, the surviving instance reads the redo log entries generated by the failed instance, and uses that information to ensure that all committed transactions are reflected in the database. No data from committed transactions is lost. The instance that is performing recovery rolls back any transactions that were active at the time of the failure and releases any resources being used by those transactions.

Multiple-node Failure

As long as one instance continues running, its SMON process performs instance recovery for any other instances that fail in a parallel server.

If all instances of a parallel server fail, instance recovery is performed automatically the next time an instance opens the database. The instance does not have to be one of the instances that failed, and it can mount the database in either shared or exclusive mode from any node of the parallel server. This recovery procedure is the same for Oracle running in shared mode as it is for Oracle in exclusive mode, except that one instance performs instance recovery for all of the instances that failed.

Incremental Checkpointing

Incremental checkpointing improves the performance of crash and instance recovery (but *not* media recovery). An incremental checkpoint records the position in the redo thread (log) from which crash / instance recovery needs to begin. This log position is determined by the oldest dirty buffer in the buffer cache. The incremental checkpoint information is maintained periodically with minimal or no overhead during normal processing.

Recovery performance is roughly proportional to the number of buffers that had not been written to the database prior to the crash. You can influence the performance of crash or instance recovery by setting the parameter DB_BLOCK_MAX_DIRTY_TARGET, which specifies an upper bound on the number of dirty buffers that can be present in the buffer cache of an instance at any moment in time. Thus, it is possible to influence recovery time for situations where the buffer cache is very large and / or where there are stringent limitations on the duration of crash / instance recovery. Smaller values of this parameter impose higher overhead during normal processing since more buffers have to be written. On the other hand, the smaller the value of this parameter, the better the recovery performance, since fewer blocks need to be recovered.

Incremental checkpoint information is maintained automatically by Oracle8 Server without affecting other checkpoints (such as log switch checkpoints and user-specified checkpoints). In other words, incremental checkpointing occurs independently of other checkpoints occurring in the instance.

Incremental checkpointing is beneficial for recovery in a single instance as well as a multi-instance environment.

See Also: *Oracle8 Concepts*
Oracle8 Reference

Access to Datafiles for Instance Recovery

An instance that performs recovery for another instance must have access to all of the online datafiles that the failed instance was accessing. When instance recovery fails because a datafile fails verification, the instance that attempted to perform recovery does not fail, but a message is written to the ALERT file.

After you correct the problem that prevented access to the database files, you must use the SQL statement ALTER SYSTEM CHECK DATAFILES to make the files available to the instance.

See Also: "Datafiles" on page 6-2

Freezing the Database for Instance Recovery

With a parallel server you can use the dynamic parameter FREEZE_DB_FOR_FAST_INSTANCE_RECOVERY to control freezing of the database during instance recovery. Note that multiple instances must have the same value for this parameter.

When this parameter is set to TRUE, Oracle freezes the whole database during instance recovery. The advantage of freezing the whole database is to stop all other disk activities except those for instance recovery. Instance recovery may thus complete sooner. The drawback of freezing the whole database is that it becomes unavailable during instance recovery.

When this parameter is set to FALSE, Oracle does not freeze the whole database, thus part of the unaffected database will be accessible during instance recovery.

The system attempts to pick a good default value intelligently.

- If all online datafiles use hash locks, the default value of this parameter is FALSE. This is because, when hash locks are used, most parts of the database can be accessed by the user during instance recovery.
- If any data files use fine grain locks, the default is TRUE. When fine grain locks are used, an instance death may affect a larger portion of the database. Affected data will be accessible only after instance recovery. In this case, setting this parameter to TRUE can potentially make those pars of the database available sooner.

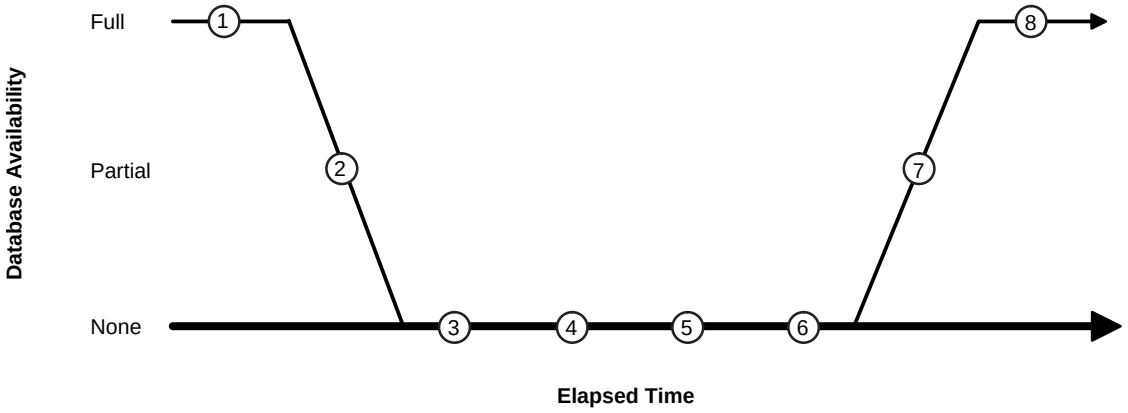
To see the number of times the whole database is frozen for instance recovery after this instance has started up, you can check the “instance recovery database freeze count” statistic in V$SYSSTAT.

See Also: *Oracle8 Reference*

Phases of Oracle Instance Recovery

Figure 22–2 illustrates the degree of database availability during each phase of Oracle instance recovery.

Figure 22–2 Phases of Oracle Instance Recovery



Phases of recovery are these:

1. Oracle Parallel Server is running on multiple nodes.
2. Node failure is detected.
3. The LM is reconfigured; resource and lock management is redistributed onto the set of surviving nodes. One call will get persistent resources. Lock value block is marked as dubious for locks held in exclusive or protected write mode. Lock requests are queued.
4. LCK*n* processes build a list of all invalid lock elements.
5. Roll forward. Redo logs of the dead thread(s) are applied to the database.
6. LCK*n* processes make all invalid lock elements valid.
7. Roll back. Rollback segments are applied to the database for all uncommitted transactions.
8. Instance recovery is complete, and all data is accessible.

During phase 5 (forward application of the redo log), database access is limited by the transitional state of the buffer cache. The following data access restrictions exist

for all user data in all datafiles, regardless of whether you are using hashed or fine grain locking, or any particular features:

- No writes to any of the surviving buffer caches will succeed while the access is limited.
- No disk I/O of any sort via the buffer cache and direct path can be done from any of the surviving instances.
- No lock requests will be made to the LM for any user data.

Reads of buffers already in the cache with the correct global lock can be done, since they do not involve any I/O or lock operations.

The transitional state of the buffer cache begins at the conclusion of the initial lock scan phase when instance recovery is first started by scanning for dead redo threads. Subsequent lock scans are made if new dead threads are discovered. This state lasts while the redo log is applied (cache recovery) and ends when the redo logs have been applied and the file headers have been updated. Cache recovery operations conclude with validation of the invalid locks, which occurs after the buffer cache state is normalized.

Recovery from Media Failure

After a media failure that results in the loss of one or more database files, you must use backups of the datafiles to recover the database.

If you are using Recovery Manager, you might also need to apply incremental backups, archived redo log files and a backup of the control file.

If you are using operating system utilities, you might need to apply archived redo log files to the database and use a backup of the control file.

This section describes:

- Complete Media Recovery
- Incomplete Media Recovery
- Restoring and Recovering Redo Log Files
- Disaster Recovery

See Also: *Oracle8 Backup and Recovery Guide* for procedures to recover from various kinds of media failure.

Complete Media Recovery

You can perform complete media recovery in either exclusive or shared mode. The following table shows what the status of the database must be, for you to recover particular database objects.

Table 22–2 Database Status for Media Recovery

To Recover	Database Status
An entire database or the SYSTEM tablespace	The database must be mounted but not opened by any instance.
A tablespace other than the SYSTEM tablespace	The database must be opened by the instance performing the recovery and the tablespace must be offline.
A datafile	The database can be open with the datafile offline, or the database can be mounted but not opened by any instance. (For a datafile in the SYSTEM tablespace, the database must be mounted but not open.)

You can recover multiple datafiles or tablespaces on multiple instances simultaneously.

Complete Media Recovery Using Operating System Utilities

With operating system utilities you can perform open database recovery of tablespaces or datafiles in shared mode, by using the Server Manager command RECOVER TABLESPACE or RECOVER DATAFILE.

You can use the Server Manager RECOVER DATABASE command to recover a database that is mounted in shared mode, but not open. Only one instance can issue this command in a parallel server.

Note: The recommended method of recovering a database is to use Server Manager. Direct use of the ALTER DATABASE RECOVER SQL command is not recommended.

Complete Media Recovery Using Recovery Manager

With Recovery Manager, you can issue the following statements to restore and recover the files:

RESTORE DATABASE
RESTORE TABLESPACE
RESTORE DATAFILE
RECOVER DATABASE
RECOVER TABLESPACE
RECOVER DATAFILE

Incomplete Media Recovery

Incomplete media recovery can be performed while the database is mounted in shared or exclusive mode, but not open by any instance, using the following database recovery options:

With Recovery Manager:

- UNTIL CHANGE *integer*
- UNTIL TIME *date*
- UNTIL LOGSEQ *integer* THREAD *integer*

With operating system utilities:

- UNTIL CANCEL
- UNTIL CHANGE *integer*
- UNTIL TIME *date*

See Also: *Oracle8 Backup and Recovery Guide*

Restoring and Recovering Redo Log Files

Media recovery of a database accessed by a parallel server may require multiple archived log files to be open at the same time. Because each instance writes redo log data to a separate thread of redo, recovery may require as many as one archived log file per thread.

However, if a thread's online redo log contains enough recovery information, restoring any archived log files for that thread will be unnecessary.

Recovery Using Recovery Manager

Recovery Manager automatically restores and applies the archive logs required. By default, Recovery Manager will restore archive logs to the LOG_ARCHIVE_DEST directory of the instances to which it connects. If you are using multiple nodes to restore and recover, this means that the archive logs may be restored to any of the nodes performing the restore/recover. The nodes which will actually read the restored logs and perform the roll forward is the target node to which the connection was initially made. You must ensure that the logs are readable from that node.

See Also: *Oracle8 Backup and Recovery Guide* for information about overriding the location to which Recovery Manager restores archive logs.

Recovery Using Operating System Utilities

When recovering using Server Manager, you are prompted for the archived log files as they are needed. Messages supply information about the required files, and Server Manager prompts you for the filename.

For example, if the log history is enabled and the filename format is LOG_T%t_SEQ%s, where %t is the thread and %s is the log sequence number, then you might receive these messages to begin recovery with SCN 9523 in thread 8:

```
ORA-00279: Change 9523 generated at 27/09/91 11:42:54 needed for thread 8
ORA-00289: Suggestion : LOG_T8_SEQ438
ORA-00280: Change 9523 for thread 8 is in sequence 438
Specify log: {<RET> = suggested | filename | AUTO | FROM | CANCEL}
```

If you use the ALTER DATABASE statement with the RECOVER clause instead of Server Manager, you receive these messages but not the prompt. Redo log files may be required for each enabled thread in the parallel server. Oracle issues a message when a log file is no longer needed. The next log file for that thread is then requested, unless the thread was disabled or recovery is finished.

If recovery reaches a time when an additional thread was enabled, Oracle simply requests the archived log file for that thread. Whenever an instance enables a

thread, it writes a redo entry that records the change; therefore, all necessary information about threads is available from the redo log files during recovery.

If recovery reaches a time when a thread was disabled, Oracle informs you that the log file for that thread is no longer needed and does not request any further log files for the thread.

Note: If Oracle reconstructs the names of archived redo log files, the format that LOG_ARCHIVE_FORMAT specifies for the instance doing recovery must be the same as the format specified for the instances that archived the files. All instances should use the same value of LOG_ARCHIVE_FORMAT in a parallel server, and the instance performing recovery should also use that value. You can specify a different value of LOG_ARCHIVE_DEST during recovery if the archived redo log files are not at their original archive destinations.

Disaster Recovery

Disaster recovery is used when a failure makes a whole site unavailable. In this case, you can recover at an alternate site using open or closed database backups. (To recover up to the latest point in time, all logs must be available at a remote site; otherwise some work may be lost.)

This section describes disaster recovery using Recovery Manager, and using operating system utilities.

Disaster Recovery Using Recovery Manager

The following scenario assumes:

- you have lost the whole database, all control files and the online redo log.
- you will be distributing your restore over 2 nodes
- there are 4 tape drives (two on each node)
- you are using a recovery catalog

Note: It is highly advisable to back up the database immediately after opening the database reset logs, since all previous backups are invalidated. (This is not shown in the example.)

Note also that the SET UNTIL command is used in case the database structure has changed in the most recent backups, and you wish to recover to that point in time. In this way Recovery Manager restores the database to the same structure the database had at the specified time.

Before You Begin: Before beginning the database restore, you must:

- restore your initialization file, and your recovery catalog from your most recent backup
- catalog any archive logs, datafile copies or backup sets which are on disk, but are not registered in the recovery catalog

  The archive logs up to the logseq number being restored *must* be cataloged in the recovery catalog, or Recovery Manager will not know where to find them.

  If you resync the recovery catalog frequently, and have an up-to-date copy from which you have restored, there should not be many archive logs that need cataloging.

What the Sample Script Does: The following script restores and recovers the database to the most recently available archived log, which is log 124 thread 1. It does the following:

- starts the database NOMOUNT, and restricts connections to DBA-only users
- restores the control file to the location specified
- copies (or replicates) this control file to all the other locations specified by the CONTROL_FILES initialization parameter
- mounts the control file
- catalogs any archive logs not in the recovery catalog
- restores the database files (to the original locations)

  If volume names have changed you must use the statement SET NEWNAME FOR ... before the restore, then perform a switch after the restore (to update the control file with the datafiles' new locations).

- recovers the datafiles by either using a combination of incremental backups and redo, or just redo.

  Recovery Manager will complete the recovery when it reaches the log sequence number specified.

- Opens the database resetlogs.

Note: Only complete the following step if you are certain there are no other archived logs which can be applied.

- Oracle recommends you back up your database after the resetlogs. (This is not shown in the example.)

Restore/Recover Sample Script:

The DBA starts up Server Manager as follows:

```
SVRMGRL> connect scott/tiger as sysdba
Connected.
SVRMGRL> startup nomount restrict
```

The DBA then starts up Recovery Manager and runs the script.

Note: The user specified in the target parameter must have SYSDBA privilege.

```
rman target scott/tiger@node1 rcvcat rman/rman@rcat
run {
  set until logseq 124 thread 1;
  allocate channel t1 type 'SBT_TAPE' connect 'internal/knl@node1';
  allocate channel t2 type 'SBT_TAPE' connect 'internal/knl@node1';
  allocate channel t3 type 'SBT_TAPE' connect 'internal/knl@node2';
  allocate channel t4 type 'SBT_TAPE' connect 'internal/knl@node2';
  allocate channel d1 type disk;
  restore
    controlfile to   '/dev/vgd_1_0/rlvt5';
  replicate
    controlfile from '/dev/vgd_1_0/rlvt5';
  sql 'alter database mount';
  catalog archivelog '/oracle/db_files/node1/arch/arch_1_123.rdo';
  catalog archivelog '/oracle/db_files/node1/arch/arch_1_124.rdo';
  restore
    (database);
  recover
    database;
  sql 'alter database open resetlogs';
  }
```

Disaster Recovery Using Operating System Utilities

Use the following procedure.

1. Restore the last full backup at the alternate site as described in *Oracle8 Backup and Recovery Guide*.
2. Start up Server Manager.
3. Connect as SYSDBA.
4. Start and mount the database with the STARTUP MOUNT statement.
5. Initiate an incomplete recovery using the RECOVER command with the appropriate UNTIL option.

   The following command is an example:

   ```
   RECOVER DATABASE USING BACKUP CONTROLFILE UNTIL CANCEL
   ```

6. When prompted with a suggested redo log file name for a specific thread, use that filename.

   If the suggested archive log is not in the archive directory, specify where the file can be found. If redo information is needed for a thread and a file name is not suggested, try using archive log files for the thread in question.
7. Repeat step 6 until all archive log files have been applied.
8. Stop the recovery operation using the CANCEL command.
9. Issue the ALTER DATABASE OPEN RESETLOGS statement.

Note: If any distributed database actions are used, check to see whether your recovery procedures require coordinated distributed database recovery. Otherwise, you may cause logical corruption to the distributed data.

Parallel Recovery

The goal of the parallel recovery feature is to use compute and I/O parallelism to reduce the elapsed time required to perform crash recovery, single-instance recovery, or media recovery. Parallel recovery is most effective at reducing recovery time when several datafiles on several disks are being recovered concurrently.

Parallel Recovery Using Recovery Manager

With Recovery Manager's RESTORE and RECOVER commands Oracle can automatically parallelize all three stages of recovery.

Restoring Data Files: When restoring data files, the number of channels you allocate in the Recovery Manager recover script effectively sets the parallelism with which Recovery Manager will operate. For example, if you allocate 5 channels, you can have up to 5 parallel streams restoring data files.

Applying Incremental Backups: Similarly, when you are applying incremental backups, the number of channels you have allocated determines the potential parallelism.

Applying Redo Logs: Oracle applies the redo logs in parallel, as determined by the RECOVERY_PARALLELISM parameter.

The RECOVERY_PARALLELISM initialization parameter specifies the number of redo application server processes that participate in instance or media recovery. One process reads the log files sequentially and dispatches redo information to several recovery processes, which apply the changes from the log files to the datafiles. A value of 0 or 1 indicates that recovery is to be performed serially by one process. The value of this parameter cannot exceed the value of the PARALLEL_MAX_SERVERS parameter.

Parallel Recovery Using Operating System Utilities

You can parallelize instance and media recovery in two ways:

- Setting the RECOVERY_ PARALLELISM Parameter
- Specifying RECOVER Command Options

The Oracle Server can use one process to read the log files sequentially and dispatch redo information to several recovery processes to apply the changes from the log files to the datafiles. The recovery processes are started automatically by Oracle, so there is no need to use more than one session to perform recovery.

Setting the RECOVERY_ PARALLELISM Parameter

The RECOVERY_PARALLELISM initialization parameter specifies the number of redo application server processes that participate in instance or media recovery. One process reads the log files sequentially and dispatches redo information to several recovery processes, which apply the changes from the log files to the datafiles. A value of 0 or 1 indicates that recovery is to be performed serially by one process. The value of this parameter cannot exceed the value of the PARALLEL_MAX_SERVERS parameter.

Specifying RECOVER Command Options

When you use the RECOVER command to parallelize instance and media recovery, the allocation of recovery processes to instances is operating system specific. The DEGREE keyword of the PARALLEL clause can either signify the number of processes on each instance of a parallel server or the number of processes to spread across all instances.

See Also: Your Oracle system-specific documentation for more information on the allocation of recovery processes to instances.
Oracle8 Concepts for more information on parallel recovery.

23

Migrating from Single Instance to Parallel Server

This chapter describes database conversion: how to convert from a single instance Oracle8 database to a multi-instance Oracle8 database using the parallel server option.

The chapter is organized as follows:

- Overview
- Deciding to Convert
- Preparing to Convert
- Converting the Database from Single- to Multi-instance
- Troubleshooting the Conversion

Overview

The present chapter explains how to enable your database structure to support multiple instances. It can also prepare you to start a project with a single instance Oracle8 database, while being ready to migrate to multi-instance in the future. In addition, it can help you extend an existing Oracle Parallel Server configuration to additional nodes.

Attention: Before using this chapter to convert to a multi-instance database, use the *Oracle8 Migration* manual to perform any necessary upgrade of the Oracle Server. That manual also provides information on upgrading and downgrading in replicated systems.

Deciding to Convert

This section describes:

- Reasons to Convert
- Reasons Not to Convert

Reasons to Convert

You may wish to convert to a multi-instance database for the following reasons:

- You want to move from a single node to a cluster (when you have designed your application with Oracle Parallel Server in mind).
- You are already running Oracle Parallel Server, but want to extend your database to include more nodes.
- Your application was already designed for Oracle Parallel Server, but you ended up running without enough instances (without enough nodes specified for the database).
- You have enough nodes specified, but need to bring the other nodes online.

Reasons Not to Convert

Do not attempt to convert to a multi-instance database in the following situations:

- You are using a file system which is not shared.
- Your application was not designed for parallel processing; you need to examine your application more.
- You are not using a supported configuration (of shared disks, and so on).

Preparing to Convert

This section describes:

- Hardware and Software Requirements
- Converting the Application from Single- to Multi-instance
- Administrative Issues

Hardware and Software Requirements

To convert to a multi-instance database you must have:

- a supported hardware and OS software configuration
- license for Oracle Parallel Server
- Oracle Server running on all nodes
- Oracle Parallel Server linked in

Converting the Application from Single- to Multi-instance

Just making your database run in parallel does not automatically mean that you have effectively implemented parallel processing. Besides migrating your existing database from single instance Oracle to multi-instance Oracle, you must also migrate any existing application which was designed for single-instance Oracle. Preparing an application for use with a multi-instance database may require application partitioning and physical schema changes.

See Also: Chapter 12, “Application Analysis”, for a full discussion.

Administrative Issues

Note the following ramifications of conversion:

- Your regular backup procedures should be in place before you proceed to convert from single-instance Oracle8 to the Oracle8 Parallel Server.
- Additional archiving considerations apply in an Oracle Parallel Server environment. In particular, the archive file format must have the thread number. Furthermore, archived logs from all nodes are needed for media recovery. If you archive to a file, then on systems where file systems cannot be shared, some method of accessing the archive logs is required.

See Also: Chapter 21, “Backing Up the Database”.

Converting the Database from Single- to Multi-instance

The following procedure explains how to migrate an existing database from single instance Oracle to multi-instance Oracle. Remember that you must also migrate the application from single-instance to multi-instance.

1. Modify your application to make it Oracle Parallel Server ready.
2. Make sure that all necessary files are shared between the nodes.

   Oracle8 Parallel Server assumes that disks are shared between the different instances such that each instance can access all log files, control files, and database files. These files should normally be on raw devices, since the disks are shared through raw devices on most clusters.

   Attention: NFS cannot be used to share files for Oracle8 Parallel Server. NFS does not provide adequate availability: if the node goes down, NFS goes down and the files cannot be reached. Likewise, NFS does not provide adequate consistency: a write may be cached and not written to disk immediately.
3. Check MAXINSTANCES on the single instance.

   The MAXINSTANCES parameter was set at database creation, usually to its default value of 1. With MAXINSTANCES set to 1, only one instance can run the database, and the database cannot run in parallel server mode. Note that the number of rows in V$THREAD is one per created thread. The MAXINSTANCES value may be much higher. You can check V$ACTIVE_INSTANCES to find this value.

   To check the value of MAXINSTANCES you can check V$ACTIVE_INSTANCES. Alternatively, you can dump the control file to a trace file by entering

   ```
   SQL> ALTER DATABASE BACKUP CONTROLFILE TO TRACE;
   ```

   The trace file may look like this:

   ```
   Dump file /mf1/qjones/qj1/rdbms/log/ora_20016.trc
   Oracle8 Server Release 8.0.3
   With the distributed, replication, parallel query and
      Parallel Server options
   PL/SQL Release 3.0
   ORACLE_HOME = /mf1/qjones/qj1
   ORACLE_SID = mf1qj1
   Oracle process number: 19        Unix process id: 20016
   System name:    mf1seq
   Node name:      mf1seq
   ```

```
Release:        3.2.0
Version:        V2.1.1
Machine:        i386
Wed Feb 22 14:30:22 1997
Wed Feb 22 14:30:23 1997
*** SESSION ID:(18.1)
# The following commands will create a new control file and
# use it to open the database.
# No data other than log history will be lost. Additional logs
# may be required for media recovery of offline data files.
# Use this only if the current version of all online logs are
# available.
STARTUP NOMOUNT
CREATE CONTROLFILE REUSE DATABASE “TPCC” NORESETLOGS
NOARCHIVELOG
    MAXLOGFILES 16
    MAXLOGMEMBERS 2
    MAXDATAFILES 62
     MAXINSTANCES 1
    MAXLOGHISTORY 100
LOGFILE
  GROUP 1 ‘/dev/rvol/v-qj80W-log11’  SIZE 200M,
  GROUP 2 ‘/dev/rvol/v-qj80W-log12’  SIZE 200M
DATAFILE
  ‘/dev/rvol/v-qj80W-sys’,
  ‘/dev/rvol/v-qj80W-temp’,
  ‘/dev/rvol/v-qj80W-cust1’,
.
.
.
;
# Recovery is required if any of the datafiles are restored
# backups, or if the last shutdown was not normal or
# immediate.
RECOVER DATABASE
# Database can now be opened normally.
ALTER DATABASE OPEN;
```

4. Edit the control file script to include a larger MAXINSTANCES value.

   Edit the trace file so that it only contains the SQL commands necessary to generate the CREATE CONTROLFILE statement. Then make the following changes:

   a. Set PFILE to point to the correct initialization file.

   b. Increase the MAXINSTANCES parameter to the number of Oracle instances you want to support.

   c. Use a large value for the MAXLOGHISTORY parameter.

   The resulting control file is a script that will recover and reopen your database if necessary.

   Before you run the SQL file, make sure that the current control file(s) are moved to the backup directory.

   A sample script follows:

```
STARTUP NOMOUNT PFILE=$HOME/perf/tkvc/admin/tkvcrun.ora
CREATE CONTROLFILE REUSE DATABASE "TPCC" NORESETLOGS
NOARCHIVELOG
    MAXLOGFILES 16
    MAXLOGMEMBERS 2
    MAXDATAFILES 62
    MAXINSTANCES 1
    MAXLOGHISTORY 100
LOGFILE
  GROUP 1 '/dev/rvol/v-qj80W-log11'  SIZE 200M,
  GROUP 2 '/dev/rvol/v-qj80W-log12'  SIZE 200M
DATAFILE
  '/dev/rvol/v-qj80W-sys',
  '/dev/rvol/v-qj80W-temp',
  '/dev/rvol/v-qj80W-cust1',
.
.
.
;
# Recovery is required if any of the datafiles are restored
# backups, or if the last shutdown was not normal or
# immediate.
RECOVER DATABASE
# Database can now be opened normally.
ALTER DATABASE OPEN;
```

5. Back up the new control file immediately after conversion. Oracle Corporation also recommends that you commence your backup procedures for the database.

6. Decide how to administer the initialization parameter file(s).

   Each instance will have private initialization parameters, but some of the parameters need to have the same value on each instance. There are two alternative ways of administering this.

   One approach is for each instance to have a private parameter file that includes the common parameter file that is shared between the instances. The common parameter file must be on a shared device accessible to all nodes. This way, when you need to make a generic change to one of the common initialization parameters, you need only make the change on one node--rather than on all nodes.

   Alternatively, you can make multiple copies of the parameter file and place one on the private disk of each node that participates in the Oracle Parallel Server. In this case you would need to update all of the parameter files each time you make a generic change.

7. Edit the following parameters in the instance-specific initialization parameter file:

   a. Specify an INSTANCE_NUMBER for this instance. Each instance will be numbered at startup time. The instance number is used in the free list group assignment. If you do not specify the INSTANCE_NUMBER, Oracle will assign a number based on the order of start up.

   b. Specify ROLLBACK_SEGMENTS. Each instance should have a set of private rollback segments to work on.

   c. Specify the THREAD parameter in the initialization parameter file so that the instance always starts with the same set of redo log files. A thread number will be assigned at startup time, to associate an instance with the log files of that thread. By default this value is 0; you can set it to 1 for the first instance.

   d. Add the DB_NAME parameter to the initialization parameter file.

8. Make sure that the following common initialization parameters have the same values for all instances:

   CONTROL_FILES
   DB_BLOCK_SIZE
   DB_FILES
   DB_NAME
   GC_FILES_TO_LOCKS
   GC_ROLLBACK_LOCKS
   LM_LOCKS (identical values recommended)
   LM_PROCS (identical values recommended)
   LM_RESS (identical values recommended)
   LOG_FILES
   MAX_COMMIT_PROPAGATION_DELAY
   ROW_LOCKING
   SINGLE_PROCESS

9. Make sure that the Oracle executable is linked with the Parallel Server Option, and that each node is running the same versions of the executable. The banner displayed upon connection should display the words “Parallel Server”.

   Note: Corruption may occur if one node opens the database in shared mode and another node opens it in exclusive mode.

10. Perform a shutdown normal of the database.

11. Back up the control files using operating system commands.

12. Remove the control files (keep the backups).

13. Run the new script you have built, which will recreate the old control files with new data—larger structures for some of the database objects.

14. Add rollback segments.

15. Add additional threads.

16. Shut down the database.

17. Start up the database in shared mode. The first instance will be started.

18. Add the second instance in shared mode, using the standard procedure described in "Starting Up in Shared Mode" on page 18-14. (Note that the second instance will only succeed if the first instance is in shared mode.) Add redo log files, rollback segments, and so on.

19. Tune the GC_* and LM_* parameters for optimal performance.

Troubleshooting the Conversion

This section explains how to resolve common errors:

- Database Recovery After Conversion
- Loss of Rollback Segment Tablespace
- Inadvisable NFS Mounting of Parameter File

Database Recovery After Conversion

If you should lose your database and Oracle8 files after converting from single-instance Oracle to Oracle Parallel Server, you would have to restore your cold backup and then apply all changes from the redo logs. In this case your old control file would be used, as though you had never done the conversion. You would have to recreate the new control file, if you migrate to Oracle Parallel Server.

Loss of Rollback Segment Tablespace

The following problem may occur if a user has created tablespaces for private rollback segments, and allocated them to specific instances at startup. It may also occur if files that contain rollback segments are lost.

If you lose one rollback segment tablespace or file containing rollback segments due to media failure, all of the instances will fail. To recover, you must shut down all instances. All the other rollback segments must remain offline so that you can bring the one you want to recover off line.

Inadvisable NFS Mounting of Parameter File

It is not advisable to access a common parameter file (or any Oracle file or executable) over NFS. If the NFS disk were to go down, no other instance could start. Note also that access to control files and data files is not supported over NFS.

Part V

Reference

A

Differences from Previous Versions

This appendix describes differences in the Oracle Parallel Server Option from release to release.

- Differences Between Release 8.0.3 and Release 8.0.4
- Differences Between Release 7.3 and Release 8.0.3
- Differences Between Release 7.2 and Release 7.3
- Differences Between Release 7.1 and Release 7.2
- Differences Between Release 7.0 and Release 7.1
- Differences Between Version 6 and Release 7.0

See Also: *Oracle8 Migration* for instructions on upgrading your database.

Differences Between Release 8.0.3 and Release 8.0.4

New Initialization Parameters

The following initialization parameters were added specifically for Oracle Parallel Server:

OGMS_HOME
GC_LATCHES

Obsolete Initialization Parameters

The following initialization parameters are obsolete:

MTS_LISTENER_ADDRESS
MTS_MULTIPLE_LISTENERS

Dynamic Performance Views

The following views changed:

V$DLM_LOCKS

Group Membership Services

A new option has been added for the **ogmsctl** command.

Differences Between Release 7.3 and Release 8.0.3

New Initialization Parameters

The following parameters were added specifically for Oracle Parallel Server:

FREEZE_DB_FOR_FAST_INSTANCE_RECOVERY
LM_LOCKS
LM_PROCS
LM_RESS
INSTANCE_GROUPS
PARALLEL_INSTANCE_GROUP
OPS_ADMIN_GROUP
ALLOW_PARTIAL_SN_RESULTS

See Also: "Setting LM_* Parameters" on page 18-11.

Obsolete GC_* Parameters

The following global cache lock initialization parameters are obsolete:

GC_DB_LOCKS parameter
GC_FREELIST_GROUPS parameter
GC_ROLLBACK_SEGMENTS parameter
GC_SAVE_ROLLBACK_LOCKS parameter
GC_SEGMENTS parameter
GC_TABLESPACES parameter

See Also: "GC_* Global Constant Parameters" on page 18-8.

Changed GC_* Parameters

The values set by the GC_* parameters are not adjusted to prime numbers, but rather are left exactly as entered.

The following parameters have changed:

GC_FILES_TO_LOCKS
GC_ROLLBACK_LOCKS
GC_RELEASABLE_LOCKS

See Also: "GC_* Initialization Parameters" on page 9-13.

Dynamic Performance Views

The following new views were added:

V$RESOURCE_LIMIT
V$DLM_CONVERT_LOCAL
V$DLM_CONVERT_REMOTE
V$DLM_LATCH
V$DLM_MISC
V$FILE_PING
V$CLASS_PING

The following views changed:

V$BH
V$SESSIONS
V$SYSSTAT

Global Dynamic Performance Views

Global dynamic performance views (GV$ fixed views) were added, corresponding to each of the V$ views except for V$ROLLNAME.

See Also: "Global Dynamic Performance Views" on page 20-3.

Integrated Distributed Lock Manager

Oracle Parallel Server release 8.0 is not dependent on an external Distributed Lock Manager. The lock management facility is now internal to Oracle. The Integrated Distributed Lock Manager is dependent on an external node monitor.

LMON and LMD*n* processes have been added.

See Also: Chapter 8, “Integrated Distributed Lock Manager: Access to Resources”.

Instance Groups

The ability to logically group instances together and perform operations upon all of the associated instances was added.

See Also: "Specifying Instance Groups" on page 18-22.

Group Membership Services

Group Membership Services (GMS) is used by the Lock Manager (LM) and other Oracle components for inter-instance initialization and coordination.

See Also: "Using Group Membership Services" on page 18-21.

Fine Grain Locking

In Oracle Parallel Server release 8.0, fine grain locking is available on all platforms. It is enabled by default.

Client-side Application Failover

Oracle8 supports the ability of the application to automatically reconnect if the connection to the database is broken.

See Also: "Client-side Application Failover" on page 22-2.

Recovery Manager

Recovery Manager is now the preferred method of recovery from media failure.

See Also: "Recovery from Media Failure" on page 22-15.

Differences Between Release 7.2 and Release 7.3

Initialization Parameters

The following initialization parameters were added specifically for the Parallel Server Option:

CLEANUP_ROLLBACK_ENTRIES
DELAYED_LOGGING_BLOCK_CLEANOUTS
GC_FREELIST_GROUPS
GC_RELEASABLE_LOCKS

Data Dictionary Views

The following view was added specifically for the Parallel Server Option:

FILE_LOCK

Dynamic Performance Views

The following view changed:

V$BH

The following views were added:

V$SORT_SEGMENT
V$ACTIVE_INSTANCES

Free List Groups

You can now set free list groups for indexes, as well as for tables and clusters.

Fine Grain Locking

In Oracle Parallel Server release 7.3, PCM locks have additional options for configuration using fine grain locking. The changes affect the interpretation of the various parameters that determine the locks used to protect the database blocks in the distributed parallel server cache.

Fine grain locking is a more efficient method for providing locking in a multinode configuration. It provides a reduced rate of lock collision, and reduced space requirements for managing locks, particularly in MPP systems. This feature relies on facilities provided by the hardware and operating system platform, and may not be available on all platforms.

Fine grain locking is discussed in the section "Two Methods of PCM Locking: Fixed and Releasable" on page 9-15.

Instance Registration

This feature enables each instance to register itself and certain of its attributes, and to establish contact with any other instance. Instance registration is transparent to the user, except in the case of parallel query failure on remote instances of a parallel server. If a parallel query dies due to an error on a remote instance, the failed instance is now identified in the error message.

Sort Improvements

This release offers a more efficient way of allocating sort temporary space, which reduces serialization and cross-instance pinging. If you set up this capability correctly, it can particularly benefit OPS performance in parallel mode.

For best results, try to establish stable sort space. Remember that sort space is cached in the instance. One instance does not release the space unless another instance runs out of space and issues a call to the first one to do so. This is an expensive, serialized process which hurts performance. If your system permanently deviates from stable sort space, it is better to overallocate space, or simply not to use temporary tablespaces.

To determine the stability of your sort space, you can check the V$SORT_SEGMENT view. This new view shows every instance's history of sorting. If the FREED_EXTENTS and ADDED_EXTENTS columns show excessive allocation / deallocation activity, you should consider adding more space to the corresponding tablespace. Check also the FREE_REQUESTS value to determine if there is inter-instance conflict over sort space.

Another reason for excessive allocation and deallocation may be that some sorts are just too big. It may be worthwhile to assign a different temporary tablespace for the operations which require huge sorts. The MAX_SORT_SIZE value may help you to determine whether these large sorts have indeed occurred.

See Also: *Oracle8 Administrator's Guide* for more information on sort enhancements.

XA Performance Improvements

Various scalability and throughput improvements have been made that affect XA transactions. These changes have no visible impact, other than improved performance.

The following three latches perform much better, and so enhance scalability:

- Global transaction mapping table latch
- Enqueues latch
- Session switching latch

Transaction throughput is enhanced because most of the common XA calls have reduced code path and reduced round-trips to the database.

XA Recovery Enhancements

Recovery of distributed transactions submitted through a TP monitor using the XA interface is now fully supported in OPS.

The XA_RECOVER call has been enhanced, ensuring correct and complete recovery of one instance from transactions that have failed in another instance.

An option has been added to make the XA_RECOVER call wait for instance recovery. This feature enables one Oracle instance to do recovery on behalf of a failed Oracle instance, when both are part of the same OPS cluster.

The XA_INFO string has a new clause called OPS_FAILOVER. If this is set to true for a given XA resource manager connection, any XA_RECOVER call issued from that connection will wait for any needed instance recovery to complete. The syntax is as follows:

OPS_FAILOVER=T

Upper- or lowercase (T or t) can be used. The default value of OPS_FAILOVER is false (F or f).

Previously, there was no guarantee that an XA_RECOVER call would return the list of in-doubt transactions from the failed instance. Setting OPS_FAILOVER=T ensures that this will happen.

When OPS_FAILOVER is set to true, the XA_RECOVER call will wait until SMON has finished cache recovery, has identified the in-doubt transactions, and added them to the PENDING_TRANS$ table that has a list of in-doubt transactions.

Deferred Transaction Recovery

Transaction recovery behavior has changed to allow:

- Greater database availability during startup
- Transactions to be recovered in parallel, if needed
- Recovery of long transactions without interfering with recovery of short transactions

Fast Warmstart

In previous releases, the database could not be opened until complete transaction recovery was performed after a failure. As of release 7.3, the database is opened for connections as soon as cache recovery is completed. (This only applies when opening the database, as opposed to doing failover in an OPS environment.) In case of an instance failure, the database is available for connections through other running instances.

This means that active transactions as of the time of the failure are not yet rolled back; they appear active (holding row locks) to users of the system. Furthermore, all transactions system-wide that were active as of the time of failure are marked DEAD and the rollback segments containing these transactions are marked PARTIALLY AVAILABLE. These transactions are recovered as part of SMON recovery in the background, or by foreground processes that may encounter them, as described in the next section. The rollback segment is available for onlining.

Transaction Recovery

Given fast warmstart capability, the time needed to recover all transactions does not limit the general availability of the database. All data except the part locked by unrecovered transactions is now available to users. Given an OLTP workload, however, all the requests that were active when the database or instance went down will probably be resubmitted immediately. They will very likely encounter the locks held by the unrecovered transactions. The time needed to recover these transactions is thus still critical for access to the locked data. To alleviate this problem, transactions can now be recovered in parallel, if needed. Recovery can be done by the following operations.

Recovery by Foreground Processes. Rows may be locked by a transaction that has not yet been recovered. Any foreground process that encounters such a row can itself recover the transaction. The current recovery by SMON will still happen--so the entire transaction recovery will complete eventually. But if any foreground process runs into a row lock, it can quickly recover the transaction holding the lock,

and continue. In this way recovery operations are parallelized on a need basis: dead transactions will not hold up active transactions. Previously, active transactions had to wait for SMON to recover the dead transactions.

Recovery is done on a per-rollback segment basis. This prevents multiple foreground processes in different instances from recovering transactions in the same rollback segment, which would cause pinging. The foreground process fully recovers the transaction that it would otherwise have waited for. In addition, it makes a pass over the entire rollback segment and partially recovers all unrecovered transactions. It applies a configurable number of changes (undo records) to each transaction. This allows short transactions to be recovered quickly; without waiting for long transactions to be recovered. The initialization parameter CLEANUP_ROLLBACK_ENTRIES specifies the number of changes to apply.

Recovery by SMON. SMON transaction recovery operations are mostly unchanged. SMON is responsible for recovering transactions marked DEAD within its instance, transaction recovery during startup, and instance recovery. The only change is that it will make multiple passes over all the transactions that need recovery and apply only the specified number of undo records per transaction per pass. This prevents short transactions from waiting for recovery of a long transaction.

Recovery by Onlining Rollback Segment. Onlining a rollback segment now causes complete recovery of all transactions it contains. Previously, the onlining process posted SMON to do the recovery. Note that implicit onlining of rollback segments as part of warmstart or instance startup does not recover all transactions but instead marks them DEAD.

Load Balancing at Connect

In standard Oracle, load balancing now allows multiple listeners and multiple instances to be balanced at SQL*Net connect time. Multiple listeners can now listen on one Oracle instance, and the Oracle dispatcher will register with multiple listeners. The SQL*Net client layer will randomize multiple listeners via the DESCRIPTION_LIST feature.

For more information about load balancing at connect, please see the SQL*Net documentation for Oracle7 Server release 7.3.

Bypassing Cache for Sort Operations

The default value for the SORT_DIRECT_WRITES initialization parameter is now AUTO; it will turn itself on if your sort area is a certain size or greater. This will improve performance. For more information, see the *Oracle8 Tuning*.

Delayed-Logging Block Cleanout

In Oracle7 Server release 7.3, the performance of delayed block cleanout is improved and related pinging is reduced. These enhancements are particularly beneficial for the Oracle Parallel Server.

Oracle7 Server release 7.3 provides a new initialization parameter, DELAYED_LOGGING_BLOCK_CLEANOUTS, which is TRUE by default.

When Oracle commits a transaction, each block that the transaction changed is not immediately marked with the commit time. This is done later, upon demand--when the block is read or updated. This is called *block cleanout*. When block cleanout is done during an update to a current block, the cleanout changes and the redo records of the update are piggybacked with those of the update. In previous releases, when block cleanout was needed during a read to a current block, extra cleanout redo records were generated and the block was dirtied. This has been changed.

As of release 7.3, when a transaction commits, all blocks in the cache changed by the transaction are cleaned out immediately. This cleanout performed at commit time is a “fast version” which does not generate redo log records and does not repin the block. Most blocks will be cleaned out in this way, with the exception of blocks changed by long running transactions.

During queries, therefore, the data block’s transaction information is normally up-to-date and the frequency with which block cleanout is needed is much reduced. Regular block cleanouts are still needed when querying a block where the transactions are still truly active, or when querying a block which was not cleaned out during commit.

During changes (INSERT, DELETE, UPDATE), the cleanout redo log records are generated and piggyback with the redo of the changes.

Parallel Query Processor Affinity

Oracle7 Server release 7.3 provides improved defaults in the method by which servers are allocated among instances for the parallel query option. As a result, users can now specify parallelism without giving any hints.

Parallel query slaves are now assigned based on disk transfer rates and CPU processing rates for user queries. Work is assigned to query slaves that have preferred access to local disks versus remote disks, which is more costly. In this way data locality will improve parallel query performance.

For best results, you should evenly divide data among the parallel server instances and nodes--particularly for moderate to large size tables that substantially domi-

nate the processing. Data should be fairly evenly distributed on various disks, or across all the nodes. For very small tables, this is not necessary.

For example, if you have two nodes, a table should not be divided in an unbalanced way such that 90% resides on one node and 10% on the other node. Similarly, if you have four disks, one should not contain 90% of the data and the others contain only 10%. Rather, data should be spread evenly across available nodes and disks. This happens automatically if you use disk striping. If you do not use disk striping, you must manually ensure that this happens, if you desire optimum performance.

Differences Between Release 7.1 and Release 7.2

Pre-allocating Space Unnecessary

For most parallel server configurations it is no longer necessary to pre-allocate data blocks to retain partitioning of data across free list groups. When a row is inserted, a group of data blocks is allocated to the appropriate free list group for an instance.

Data Dictionary Views

The following views were added specifically for the Parallel Server Option:

FILE_LOCK
FILE_PING

Dynamic Performance Views

The following views changed:

V$BH
V$CACHE
V$PING
V$LOCK_ACTIVITY

The following views were added:

V$FALSE_PING
V$LOCKS_WITH_COLLISIONS
V$LOCK_ELEMENT

Free List Groups

It is now possible to specify a particular instance, and hence the free list group, from a session, using the command:

ALTER SESSION SET INSTANCE = *instance_number*

Table Locks

It is now possible to disable the ability for a user to lock a table using the command:

ALTER TABLE *table_name* DISABLE TABLE LOCK

Re-enabling table locks is accomplished using the following command:

ALTER TABLE *table_name* ENABLE TABLE LOCK

Lock Processes

The PCM locks held by a failing instance are now recovered by the lock processes of the instance recovering for the failed instance.

Differences Between Release 7.0 and Release 7.1

Initialization Parameters

CACHE_SIZE_THRESHOLD was added.

Dynamic Performance Views

The following views changed:

V$BH
V$CACHE
V$PING
V$LOCK_ACTIVITY

Differences Between Version 6 and Release 7.0

This section describes differences between Oracle Version 6 and Oracle7 Release 7.0.

Version Compatibility

The Parallel Server Option for Version 6 is upwardly compatible with Oracle7 with one exception. In Version 6 all instances share the same set of redo log files, whereas in Oracle7 each instance has its own set of redo log files. *Oracle8 Migration* gives full details of migrating to Oracle7. After a database is upgraded to work with Oracle7 it cannot be started using a Oracle Version 6 server. Applications that run on Oracle7 may not run on Oracle Version 6.

File Operations

While the database is mounted in parallel mode, Oracle7 supports the following file operations that Oracle Version 6 only supported in exclusive mode:

- adding, renaming, or dropping a datafile
- taking a datafile offline or online
- creating, altering, or dropping a tablespace
- taking a tablespace offline or online

The instance that executes these operations may have the database open, as well as mounted.

Table A–1 shows the file operations and corresponding SQL statements that cannot be performed in Oracle Version 6 with the database mounted in parallel mode.

Table A–1 SQL Statements Now Supported in Oracle7

Operation	SQL statement
Creating a tablespace	CREATE TABLESPACE tablespace
Dropping a tablespace	DROP TABLESPACE tablespace
Taking a tablespace offline or online	ALTER TABLESPACE tablespace OFFLINE ALTER TABLESPACE tablespace ONLINE
Adding a datafile	ALTER TABLESPACE tablespace ADD DATAFILE
Renaming a datafile	ALTER TABLESPACE tablespace RENAME DATAFILE
Renaming a datafile log file	ALTER TABLESPACE tablespace RENAME FILE
Adding a redo log file	ALTER DATABASE dbname ADD LOGFILE
Dropping a redo log file	ALTER DATABASE dbname DROP LOGFILE
Taking a datafile offline or online	ALTER DATABASE dbname DATAFILE OFFLINE ALTER DATABASE dbname DATAFILE ONLINE

Oracle7 allows all of the file operations listed above while the database is mounted in shared mode.

A redo log file cannot be dropped when it is active, or when dropping it would reduce the number of groups for that thread below two. When taking a datafile online or offline in Oracle7, the instance can have the database either open or closed and mounted. If any other instance has the database open, the instance taking the file online or offline must also have the database open.

Note: Whenever you add a datafile, create a tablespace, or drop a tablespace and its datafiles, you should adjust the values of GC_FILES_TO_LOCKS and GC_DB_LOCKS, if necessary, before restarting Oracle in parallel mode. Failure to do so may result in an insufficient number of locks to cover the new file.

Deferred Rollback Segments

The global constant parameter GC_SAVE_ROLLBACK_LOCKS reserves distributed locks for deferred rollback segments, which contain rollback entries for transactions in tablespaces that were taken offline.

Version 6 does not support taking tablespaces offline in parallel mode, so the initialization parameter GC_SAVE_ROLLBACK_LOCKS is not necessary in Oracle Version 6. In Oracle7, this parameter is required for deferred rollback segments.

Redo Logs

In Oracle Version 6, all instances share the same set of online redo log files and each instance writes to the space allocated to it within the current redo log file.

In Oracle7, each instance has its own set of redo log files. A set of redo log files is called a thread of redo. Thread numbers are associated with redo log files when the files are added to the database, and each instance acquires a thread number when it starts up.

Log switches are performed on a per-instance basis in Oracle7; log switches in Oracle Version 6 apply to all instances, because the instances share redo log files.

Oracle7 introduces mirroring of online redo log files. The degree of mirroring is determined on a per-instance basis. This allows you to specify mirroring according to the requirements of the applications that run on each instance.

ALTER SYSTEM SWITCH LOGFILE

In Oracle Version 6, all instances shared one set of online redo log files. Therefore, the ALTER SYSTEM SWITCH LOGFILE statement forced all instances to do a log switch to the new redo log file.

There is no global option for this SQL statement in Oracle7, but you can force all instances to switch log files (and archive all online log files up to the switch) by using the ALTER SYSTEM ARCHIVE LOG CURRENT statement.

Initialization Parameters

The LOG_ALLOCATION parameter of Oracle Version 6 is obsolete in Oracle7. Oracle7 includes the new initialization parameter THREAD, which associates a set of redo log files with a particular instance at startup.

Free Space Lists

This section describes changes concerning free space lists.

Space Freed by Deletions and Updates

In Oracle Version 6, blocks freed by deletions or by updates that shrank rows are added to the common pool of free space. In Oracle7, blocks will go to the free list and free list group of the process that deletes them.

Free Lists for Clusters

In Oracle Version 6, the FREELISTS and FREELIST GROUPS storage options are not available for the CREATE CLUSTER statement, and the ALLOCATE EXTENT clause is not available for the ALTER CLUSTER statement.

In Oracle7, clusters (except for most hash clusters) can use multiple free lists by specifying the FREELISTS and FREELIST GROUPS storage options of CREATE CLUSTER and by assigning extents to instances with the statement ALTER CLUSTER ALLOCATE EXTENT (INSTANCE *n*).

Hash clusters in Oracle7 can have free lists and free list groups if they are created with a user-defined key for the hashing function and the key is partitioned by instance.

Initialization Parameters

The FREELISTS and FREELIST GROUPS storage options replace the initialization parameters FREE_LIST_INST and FREE_LIST_PROC of Oracle Version 6.

Import/Export

In Oracle Version 6, Export did not export free list information. In Oracle7, Export and Import can handle FREELISTS and FREELIST GROUPS.

SQL*DBA

STARTUP and SHUTDOWN must be done while disconnected in Version 6. In Oracle7, Release 7.0, STARTUP and SHUTDOWN must be issued while connected as INTERNAL, or as SYSDBA or SYSOPER.

In Oracle7, operations can be performed using either commands or the SQL*DBA menu interface, as described in *Oracle8 Utilities*.

Initialization Parameters

This section lists new parameters and obsolete parameters.

New Parameters

The new initialization parameter THREAD associates a set of redo log files with a particular instance at startup.

For a complete list of new parameters, refer to *Oracle8 Reference*.

Obsolete Parameters

The following initialization parameters used in earlier versions of the Parallel Server Option are now obsolete in Oracle7.

ENQUEUE_DEBUG_MULTI_INSTANCE
FREE_LIST_INST
FREE_LIST_PROC
GC_SORT_LOCKS
INSTANCES
LANGUAGE
LOG_ALLOCATION
LOG_DEBUG_MULTI_INSTANCE
MI_BG_PROCS (renamed to GC_LCK_PROCS)
ROW_CACHE_ENQUEUE
ROW_CACHE_MULTI_INSTANCE

For a complete list of obsolete parameters, refer to the *Oracle8 Migration*.

Archiving

In Oracle Version 6, each instance archives the online redo log files for the entire parallel server because all instances share the same redo log files. You can therefore have the instance with easiest access to the storage medium use automatic archiving, while other instances archive manually.

In Oracle7, each instance has its own set of online redo log files so that automatic archiving only archives for the current instance. Oracle7 can also archive closed threads. Manual archiving allows you to archive online redo log files for all instances. You can use the THREAD option of the ALTER SYSTEM ARCHIVE LOG statement to archive redo log files for any specific instance.

In Oracle7, the filenames of archived redo log files can include the thread number and log sequence number.

A new initialization parameter, LOG_ARCHIVE_FORMAT, specifies the format for the archived filename. A new database parameter, MAXLOGHISTORY, in the CREATE DATABASE statement can be specified to keep an archive history in the control file.

Media Recovery

Online recovery from media failure is supported in Oracle7 while the database is mounted in either parallel or exclusive mode.

In either mode, the database or object being recovered cannot be in use during recovery:

- To recover an entire database, it must be mounted but not open.
- To recover a tablespace, the database must be open and the tablespace must be offline.
- To recover datafiles (other than files in the SYSTEM tablespace), the database must be closed or open with the data files offline.

B

Restrictions

This appendix documents Oracle Parallel Server compatibility issues and restrictions.

- Compatibility
- Restrictions

Compatibility

The following sections describe aspects of compatibility between shared and exclusive modes on a parallel server.

- The Export and Import Utilities
- Compatibility Between Shared and Exclusive Modes

The Export and Import Utilities

The Export utility writes data from an Oracle database into operating system files, and the Import utility reads data from those files back into an Oracle database. This feature of Oracle is the same in shared or exclusive mode.

See Also: *Oracle8 Utilities* for more information about Import and Export.

Compatibility Between Shared and Exclusive Modes

A parallel server runs with any Oracle database created in exclusive mode. Each instance must have its own set of redo logs.

Oracle in exclusive mode can access a database created or modified by a parallel server.

If a parallel server allocates free space to a specific instance, that space may not be available for inserts for a different instance in exclusive mode. Of course, all data in the allocated extents is always available.

Restrictions

The following sections describe restrictions.

- Maximum Number of Blocks Allocated at a Time
- Restrictions in Shared Mode

Maximum Number of Blocks Allocated at a Time

The *!blocks* option of the GC_FILES_TO_LOCKS parameter enables you to control the number of blocks which are available for use within a free list group. You can use *!blocks* to specify the rate at which blocks are allocated within an extent, up to 255 blocks at a time.

Restrictions in Shared Mode

Oracle running multiple instances in shared mode supports all the functionality of Oracle in exclusive mode, except as noted in the following sections.

Restricted SQL Statements

In shared mode, the following operations are not supported:

- creating a database (CREATE DATABASE)
- creating a control file (CREATE CONTROLFILE)
- switching the database's archiving mode (the ARCHIVELOG and NOARCHIVELOG options of ALTER DATABASE)

To perform these operations, you must shut down all instances and start up one instance in exclusive mode, as described in "Starting Up Instances" on page 18-12.

Maximum Number of Datafiles

The number of datafiles supported by Oracle is operating system specific. Within this limit, the maximum number allowed depends on the values used in the CREATE DATABASE command, which in turn is limited by the physical size of the control file. This limit is the same in shared mode as in exclusive mode, but the additional instances of a parallel server restrict the maximum number of files more than a single-instance system. For more details, see *Oracle8 SQL Reference*, and your Oracle operating system specific documentation.

Sequence Number Generators

Oracle Parallel Server does not support the CACHE ORDER combination of options for sequence number generators in shared mode. Sequences created with both of these options are ordered but not cached when running in a parallel server.

Free Lists with Import and Export Utilities

The Export utility does not preserve information about multiple free lists and free list groups. When you export data from multiple instances and then, from a single node, import it into a file, the data may not end up distributed across extents in exactly the same way it was initially. The meta-data of the table into which it is imported contains the free list and free list group information which is henceforth associated with the datablocks.

Therefore, if you use Export and Import to back up and restore your data, it will be difficult to import the data so that it is partitioned again.

Index

A

B

C

D

E

F

G

H

I

K

L

M

N

O

P

R

S

T

U

V

W

X