Pro Oracle Database 10*g* RAC on Linux

Installation, Administration, and Performance

Julian Dyke and Steve Shaw

Apress®

Pro Oracle Database 10*g* RAC on Linux: Installation, Administration, and Performance

Copyright © 2006 by Julian Dyke and Steve Shaw

All rights reserved. No part of this work may be reproduced or transmitted in any form or by any means, electronic or mechanical, including photocopying, recording, or by any information storage or retrieval system, without the prior written permission of the copyright owner and the publisher.

ISBN-13 (pbk): 978-1-59059-524-4

ISBN-10 (pbk): 1-59059-524-6

Printed and bound in the United States of America 9 8 7 6 5 4 3 2 1

Trademarked names may appear in this book. Rather than use a trademark symbol with every occurrence of a trademarked name, we use the names only in an editorial fashion and to the benefit of the trademark owner, with no intention of infringement of the trademark.

Lead Editors: Matthew Moodie, Tony Davis
Technical Reviewers: Kevin Closson, Wallis R. Pereira, Brett A. Collingwood
Editorial Board: Steve Anglin, Ewan Buckingham, Gary Cornell, Jason Gilmore, Jonathan Gennick, Jonathan Hassell, James Huddleston, Chris Mills, Matthew Moodie, Dominic Shakeshaft, Jim Sumser, Keir Thomas, Matt Wade
Project Managers: Elizabeth Seymour, Beckie Brand
Copy Edit Manager: Nicole LeClerc
Copy Editors: Heather Lang, Nicole LeClerc, Jennifer Whipple
Assistant Production Director: Kari Brooks-Copony
Production Editor: Kelly Winquist
Compositor: Kinetic Publishing Services, LLC
Proofreaders: Lori Bring, Linda Seifert
Indexer: Broccoli Information Management
Artist: Kinetic Publishing Services, LLC
Cover Designer: Kurt Krames
Manufacturing Director: Tom Debolski

Distributed to the book trade worldwide by Springer-Verlag New York, Inc., 233 Spring Street, 6th Floor, New York, NY 10013. Phone 1-800-SPRINGER, fax 201-348-4505, e-mail `orders-ny@springer-sbm.com`, or visit `http://www.springeronline.com`.

For information on translations, please contact Apress directly at 2560 Ninth Street, Suite 219, Berkeley, CA 94710. Phone 510-549-5930, fax 510-549-5939, e-mail `info@apress.com`, or visit `http://www.apress.com`.

The information in this book is distributed on an “as is” basis, without warranty. Although every precaution has been taken in the preparation of this work, neither the author(s) nor Apress shall have any liability to any person or entity with respect to any loss or damage caused or alleged to be caused directly or indirectly by the information contained in this work.

The source code for this book is available to readers at `http://www.apress.com` in the Source Code section.

CHAPTER 4

RAC Design

When designing a RAC cluster, you should fully define and identify your business requirements. Once you have defined your business requirements, your next step is to identify your architectural decisions, which will, in turn, allow you to specify storage and hardware requirements. Your architectural decisions will be reflected in the logical and physical design of the application. The logical design of your application will result in the physical design, and hence will be reflected in the RAC implementation.

In this chapter, we will examine some of the design decisions you need to consider before implementing a RAC cluster.

Business Requirements

The first things to consider are the business needs of your application. The business requirements will be reflected in resource requirements, which may change over time. By thoroughly understanding the business requirements of your application, you will be able to do a good job designing and sizing your RAC implementation.

Oracle documentation commonly makes a distinction between OLTP applications and DSS, also known as data warehouses, but in our experience, almost all RAC applications fall somewhere between these two extremes.

Obviously, the RAC cluster must be sized correctly to handle the required workload. Therefore, you need to accurately estimate the number of users, the number of database connections, and the physical workload, including peaks and seasonal variations. Pay particular attention to projected growth in the workload over the lifetime of the cluster.

Technical Requirements

For RAC users, many of whom develop their own applications and deploy bespoke systems, it is important to establish how many and which types of resources the application will require and when. Resource consumption is generally measured in terms of CPU usage, memory consumption, and disk I/O. In the last case, distinguishing between read and write operations is important, as these have significantly different effects at storage level.

While it is possible, in the light of experience, to make an educated guess at the performance of a planned RAC system, it is impossible to guarantee that performance without actually testing the application with a representative user load. Even minor changes to application code can have a major impact on scalability.

If you are using a commercially available package, verify that the vendor has tested and certified the application for RAC. Many Oracle applications may not have been designed for a RAC environment. The application vendor should preferably have had access to an in-house RAC system for development and testing. Also, check whether any of their other customers are using RAC.

If you are using an existing package, seek advice on hardware requirements from the vendor. You may also be able to find other customers running the same package on RAC.

By far, the best way to evaluate the impact of running your application is to subject it to load testing specific to your environment for the number of users and projected workload. Testing the application in this manner will highlight any weaknesses in your application or environment that need to be addressed before it can be put into production on a RAC cluster.

Most of the larger hardware vendors, such as Dell, HP, and Intel, have porting and testing centers where you can test your application on various hardware configurations. This exercise can also reduce the learning curve, as you will receive assistance and instruction in RAC best-practices from their employees, who do this kind of work on a regular basis for their customers. In recent years, Oracle has also invested heavily in assisting users to deploy their application on RAC.

If the application is in production in your environment, then you should be able to measure the amount of physical I/O that it currently requires. Remember that when the application is ported from a single-instance database to a multinode RAC database, from one platform to another, or from one Oracle version to another, the I/O characteristics will most likely differ in the initialization parameters, cache sizes, and enhancements to the cost-based optimizer (CBO). CPU utilization may also be affected by these factors and by other improvements, such as optimizations in the PL/SQL compiler, Parallel Execution, new background processes, and so on.

Do not automatically assume that you will see performance improvements when moving to a newer version of Oracle. Many recent enhancements have been introduced on the basis that they should improve performance for the majority of systems. The most important thing that you can do is test an upgrade before you actually upgrade your production systems. Testing will allow you to proactively measure the potential performance gains.

One of the main benefits of a RAC cluster is that the database continues to be available in the event of a node failure. It is essential that the remaining nodes are capable of handling the required workload following a failure. You may decide all users do not need access to the cluster or that you are willing to run in a degraded performance mode until the node is repaired. You may even choose to restrict certain classes of users while one or more nodes are unavailable. In Oracle 10*g*, this can easily be implemented using database services.

However, following a node failure, the remaining nodes must have sufficient capacity to handle the essential workload to preserve the prefailure performance expectation of the users. Depending on the performance of your application, the user's expectations might lead you to implement a cluster consisting of four two-CPU nodes, as opposed to two four-CPU nodes. In the event of a failure, the former configuration would have six CPUs remaining, whereas the latter configuration would have four CPUs. In the unlikely event of a second node failure, the former configuration would still have four CPUs, but the latter configuration would have suffered a complete failure, and the database would be inaccessible.

Business Constraints

You will also need to identify the constraints under which your application must run. At one extreme, it may need to run 24-7, 365 days a year; at the other, it may only need to be available during normal working hours. The reality for most users is somewhere between these two extremes.

Your design will be affected by the level of availability required for the production platform. If a high level of availability is required on the production system, you will almost certainly need a separate test system, on which you can explore performance issues and test patches and other changes.

Check any SLAs that may exist in your organization. In addition, question any existing practices to ensure that they are still relevant for Oracle 10*g*. For example, if the previous system was implemented using Oracle 7, it may still be halted once a week for a cold backup, requiring a time window of several hours. Systems of this age, which still use dictionary managed tablespaces, are often defragmented on a regular basis. While always questionable in prior versions, this exercise has been unnecessary since

the introduction of locally managed tablespaces in Oracle 8.1.5. However, convincing operational staff, DBAs, or senior management that tasks such as these are no longer necessary is often quite difficult, though, particularly if they perceive that these tasks' elimination constitutes a risk to the business.

It is important to identify how much downtime will be available; acceptable levels of downtime should be clearly identified in your business requirements. In our experience, most sites can survive a short amount of downtime on a scheduled, fairly regular basis. One of the advantages of RAC is that downtime can be planned with reasonable certainty. You may also need to restrict access while patches are applied to the data dictionary and other persistent structures. In general, the less maintenance time that is available, the more expensive the system will become to support.

Remember to identify any seasonal peaks or variations in workload. Many systems are implemented during quiet periods, such as the summer or holiday weekends, when demand for the system is lower. However, testing should reflect the maximum load that you anticipate. For example, mobile telecommunication operators' annual peak is reached between 12:00 a.m. and 1:00 a.m. on January 1, and their systems must be able to handle the traffic generated at this time.

Keep an open mind when examining business constraints and question why particular rules, practices, and procedures exist. These are often embedded in the corporate culture and, on closer examination, apply only to legacy systems. The introduction of a new system presents an excellent opportunity to eliminate these unnecessary business constraints.

Check and eliminate tasks that serve no useful purpose in current versions of Oracle, such as defragmentation or frequent computation of statistics for very large tables. In particular, look for tasks currently performed serially that could be parallelized, which can be achieved three ways in a RAC database. The most obvious way is to identify tasks that can be performed at the same time. For example, if you are using `RMAN` to back up your RAC database, you may be able to execute some reports or run some batch jobs concurrently. While there may be some contention between the backup and the processing for resources, the overall times for the jobs may still be reduced, because these processes tend to run in the off hours, away from general online users.

The second way is to use the parallelization features built into Oracle, including parallel execution (parallel query and parallel DML). Over the past ten years, support for parallelization has improved significantly and has also been extended to various tools and utilities. Many of these operations are designed into the database and are automatically turned on and off without your knowledge. For example, in Oracle 10.1 and above, you can export and import data using the Data Pump utilities. The original `imp` utility operates serially, whereas the Data Pump import utility loads data in parallel.

The third way to implement parallelization is to use RAC itself. You can direct workloads with different characteristics to specific nodes or instances. For example, the backup might run on one node while the batch jobs run on another, optimizing the use of the SGA in each instance and minimizing the amount of interconnect traffic. Database services provide an excellent way of allocating tasks to different instances, with the assurance that if the instance fails, processing can be resumed on another instance.

Upgrade Policy

Another consideration when designing your cluster is your upgrade policy. Some sites implement the latest version of Oracle and then update the patches at every opportunity, which can be justified if your application makes use of recently introduced features. However, you should remember that each patch set introduces a certain amount of risk, so there should be a definable benefit to your business in performing the upgrade.

At the other end of the scale, many sites wait until an Oracle release is completely stable before implementing it. At the time of this writing, Oracle 10*g* has been available for over two years, but many sites are still implementing new systems on Oracle 9.2. While this delay in switching to Oracle 10*g* may be prudent in a single-instance environment, it is not advisable in a RAC environment, where

the pace of development is much faster, and the benefits of using newer releases are much more tangible. For RAC on Linux in particular, we strongly recommend that you consider Oracle 10.2 as a platform for initial deployment.

Many users implement new systems and plan never to upgrade them once they go live. If adequate testing has been performed prior to going into production, this can be a good option, as project teams often disperse as their members seek new challenges once a project has gone live. However, these users are often in a minority. For many users, the realities of their businesses mean that their IT systems continually change as events arise, such as mergers, acquisitions, growth, downsizing, consolidation, and so on. For these users, the demands on their database estate are dynamic and must be addressed on a continuing basis.

Architecture

Having established your business requirements, you are now in a position to design the architecture. Most users have finite budgets, and consequently, their optimal architectures are not usually obtainable. Therefore, your architectural design will probably be a compromise between your business requirements and your budget.

For example, your optimal architecture might include a four-node primary RAC cluster, a four-node RAC standby cluster at a separate location, and similar systems for development and quality assurance. However, you may be able to afford only a two-node primary cluster, with a single-instance standby database and a single-instance test database.

We recommend that you start by designing your optimal architecture for your business requirements and then iteratively eliminating the least essential components until your system is within budget. You should then recheck your cluster's capability of meeting your business requirements. If it's not capable, you should revisit either the business requirements or your budget to correctly set the expectations of your management and user community. Implementing a system that does not fulfill its business needs is a guaranteed recipe for failure.

Development and Test Systems

In addition to the production RAC cluster, you may need one or more additional RAC clusters to support the production cluster, depending largely on the nature of your application and the level of availability required. If the application is a purchased package, then you will need a RAC test system to validate patches and updates before applying them to the production system. If the application is developed in-house, you may also need to support RAC development or QA systems that are equivalent to the production system. We advise on some of the fundamentals of testing strategies in Chapter 5. We do not recommend that you ever use your production hardware as a test platform.

Disaster Recovery

Disaster recovery capabilities can be implemented at the storage or database level. If you opt for the latter in a RAC environment, you will probably use Oracle Data Guard to create a physical or logical standby database in a separate location. Ideally, the hardware and storage at the standby location should be as close as possible in specification to the production site. Identical hardware and storage are not always economical or feasible, and consequently, customers frequently lower the specification of the hardware on their standby site accepting a potential throughput reduction in return for

cost savings. However, the main consideration when implementing a standby strategy is to accurately determine whether the business will be able to continue to operate in the event of the total loss of the primary site.

Figure 4-1 shows an Oracle Data Guard configuration for a two-node RAC primary database located in London and a two-node RAC standby database located in Reading. A public network is required between the two locations to transmit the redo log shipping traffic. In reality, this network would probably be dedicated to prevent interference from other competing network traffic. The interconnect or storage network does not need to span the locations, although that can be another alternative. This architecture would be identical for either a physical or a logical standby database.

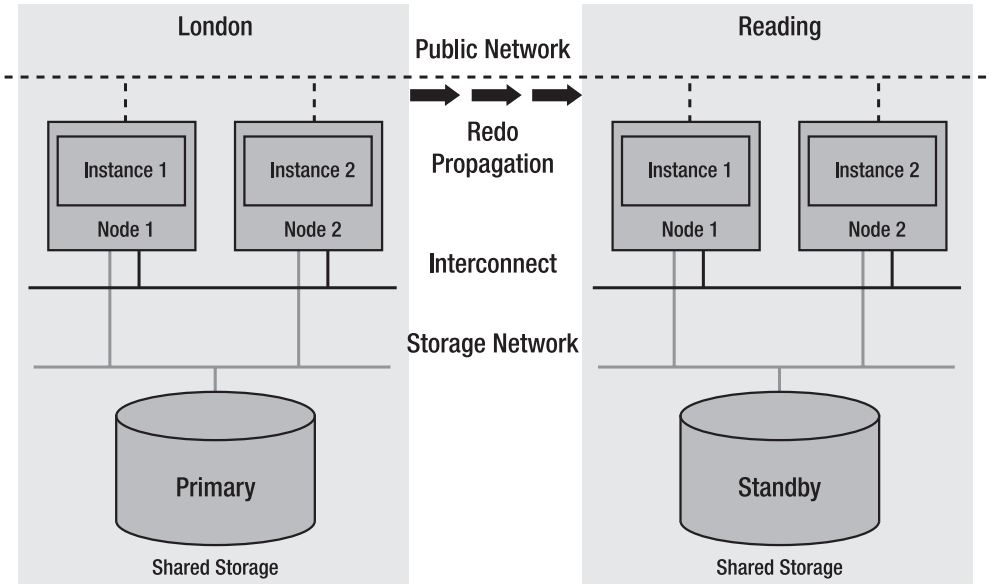

Figure 4-1. *Oracle Data Guard configuration*

You can configure a RAC primary database with either a RAC standby database or a single-instance database. It is common to see a single-instance database being used for the standby, which significantly reduces hardware and software licensing costs.

Stretch Clusters

The *stretch cluster* is an apparently attractive architecture that maximizes the utilization of cluster resources across two or more locations. For example, for a four-node stretch cluster, you might locate two nodes of the cluster at one site and the other two nodes of the cluster at a second site.

Figure 4-2 shows a stretch cluster where two nodes are located in London and the other two are in Reading. The diagram shows that three different network connections are required between the sites: the public network, a private network for the interconnect, and a storage network for block replication between the storage devices at each location.

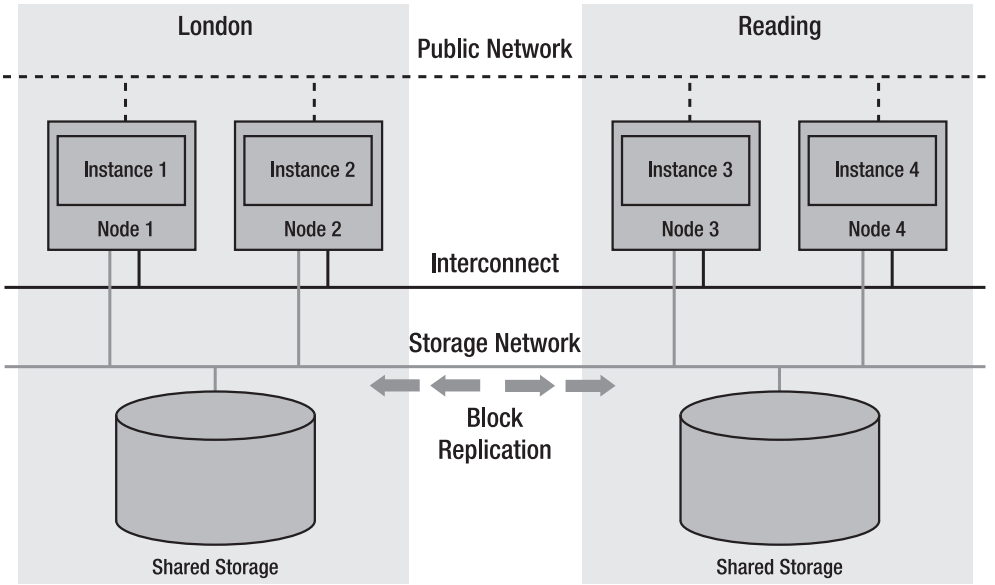

Figure 4-2. *Stretch clusters*

A stretch cluster can be deployed as a disaster recovery solution if each site has its own storage that performs the replication between the sites. This deployment guarantees that, in the event of the loss of one site, remaining sites can continue to operate and the database will remain available, which is important for many 24-7 systems.

The obvious benefit of a stretch cluster is that all nodes contribute resources to the production system. The main disadvantage is that a very fast and possibly expensive network must exist between the sites, as the cluster nodes require a fast interconnect network, and the data storage must be replicated between nodes. While node affinity is vastly improved in Oracle 10.2, nodes will still need to communicate with each other to synchronize access to resources using global enqueues.

From a network point of view, storage-level replication is much less efficient than Data Guard. Consider a transaction that updates a single row. Data Guard will only transmit the redo for the transaction, which consists of the changes made to the row, plus the undo required to reverse the changes. In the case of a small change, this transmission will not amount to more than 200 bytes. However, if replication is performed at the storage level, all affected blocks will be replicated, including blocks from the online redo log, datafiles, undo segment headers, undo blocks, and the archived redo log. Thus, a minimum of five block changes will be transmitted across to storage at the remote site.

Consequently, in a stretch cluster the latency of the network between the sites is the vital limiting factor. Despite increasing interest in new technologies, such as InfiniBand, at the time of this writing, it is still extremely rare to encounter stretch clusters in production.

Reporting Systems

Many sites implement reporting systems that are physically separated from the production database. While not strictly necessary in a RAC environment, a reporting system can act as a useful validation tool to test the backup and recovery procedure if it is built from a backup copy of the database. Alternatively, the reporting system can be built as a physical or logical standby database, thereby validating the disaster recovery capability when this function is tested on a regular basis.

Spare Servers

Compared to the cost of other RAC components, including the storage and the RAC licenses, server hardware is relatively inexpensive. This comparison changes the cost models that systems traditionally operate under. When you are designing your RAC cluster, you should consider whether to carry a stock of spare servers, which can be quickly swapped into the cluster in event of a node failure.

We recommend that you obtain quotes for hardware maintenance contracts at different service levels and compare these to the costs of purchasing hot spares. If you purchase a maintenance contract, remember that it may not be possible for a component in a failed server to be replaced immediately. A replacement server supplied by the maintenance company will need to be built, either with a standard operating system image or manually, before it can be added to the cluster. On the other hand, if you have a spare server in stock, it can be prepared in advance. Having a prebuilt spare on hand will save critical time during an outage, since it can be provisioned much more quickly than a system built from scratch.

If you do carry a spare server, ensure that it contains all the correct components, including CPU, memory, and network and fiber cards. If you need to borrow components from the spare server for the live cluster, ensure that these are replaced and retested as soon as possible. If you follow good change control policies, you will ensure that the spare server is quickly restocked for parts and ready to be provisioned on short notice in case of a failure of the production system.

Storage Requirements

In this section, we will discuss some of the design issues you will encounter when planning storage for your cluster.

You may decide which storage type to use based on your current production environment, but you should ideally decide based on your business uptime and performance requirements. For example, if you already have a SAN, then you will probably wish to utilize this technology for the new cluster. Similarly, if you already have existing NAS, then you will probably wish to use this type of storage for your RAC cluster. We cover in detail the storage options available to you in Chapter 7.

If you do not have an existing storage platform, then you should consider a number of issues when designing your storage configurations for a RAC cluster.

ASM and Cluster File Systems

In this book we describe how to configure your shared storage to using OCFS and ASM, which are both Oracle products. We cover the configuration of OCFS in Chapter 8 and ASM in Chapter 9. Alternatively, a number of third-party cluster file systems are commercially available, including Red Hat Global File System (GFS), PolyServe Matrix Server, or NFS, which is certified for specific NAS configurations.

At the time of this writing, a significant proportion of RAC users still implement raw devices instead of a cluster file system. However, we believe that many of these users are running Oracle 9i or earlier and deployed their systems before OCFS became sufficiently stable to be a viable option. We do not advise new RAC users to implement systems on raw devices, as the manageability of the alternatives, such as ASM, is significantly higher.

You should test the performance of the cluster file system. In test systems, we have seen a performance degradation of up to 15% in both OCFS and ASM over their native equivalents. Clustered file systems provide more flexibility and ease of use from a DBA perspective, and in many cases, this ease of use more than makes up for the decrease in performance of the file system.

If you are using ASM with RAID-protected storage, ensure that you specify external protection when creating the ASM disk group. In theory, providing redundancy at storage level is better than within ASM, because storage-level protection will be managed within the storage device, which is optimized for I/O at the device level, whereas ASM is managed by the operating system and uses local CPU and I/O resources to implement redundancy and disk balancing.

Shared Oracle Homes vs. Local Oracle Homes

Prior to Oracle 10*g*, most RAC users implemented separate physical Oracle homes tied to individual applications. In other words, they installed multiple copies of the Oracle binaries and supporting software on the local disks of each of the nodes in the cluster. In Oracle 10.1 and above, you can create a shared Oracle home where a single copy of the Oracle binaries is located on shared storage. Each approach has advantages and disadvantages.

The advantage of using a shared Oracle home is that you have only one set of binaries to support. Patching is simplified, as only one location has to be updated. However, this can turn into a disadvantage if the system needs to be available 24-7, as it will be difficult to apply the patches. In this scenario, depending on the severity and composition of the patch, the instances may have to be shut down, preventing user access to the application.

The advantage of using local Oracle homes is that patches can be applied to a subset of servers. The database can then be stopped and any updates can be applied to the database before it is restarted with the newly patched servers. The remaining servers can then be patched and restarted sequentially, minimizing the overall downtime for the cluster.

Backup and Recovery

If you do not have an existing storage platform or SAN, then you will need to think carefully about your current and future requirements. When identifying your storage requirements, you should factor in your backup and recovery strategy and any disaster recovery objectives, namely data replication, Recovery Point Objective (RPO), and mean time to recovery (MTTR). These requirements will drive the design of your storage subsystem and will ultimately determine what database architectures are viable for your environment.

For example, there are several methods for performing backups on a SAN or on NAS. These include triple mirroring and point-in-time snapshots, both of which allow a disk backup to be taken of the database in a matter of seconds, minimizing disruption to users. In addition, both SAN and NAS storage can be replicated over relatively long distances using synchronous and asynchronous methods, thereby providing disaster recovery capabilities.

If your budget and infrastructure are sufficient to support storage-based backup, recovery, and replication, then you should give these methods serious consideration as they are more generic, and often more reliable, than database-based methods. Another consideration of the storage system-based backup and recovery method is the working relationship between the storage and database administrators. Using a storage-based backup strategy will involve significantly more assistance from the storage administrator, which will require a lot of trust and goodwill between the storage and database administration teams to successfully implement this strategy. Storage-based backup solutions are frequently used at sites that maintain large, heterogeneous database estates.

If you wish to use database-based backup and recovery for RAC databases, we recommend that you investigate the `RMAN` utility, which has become the standard backup tool among the vast majority of RAC customers. `RMAN` functionality is implemented generically within Oracle using custom libraries linked to the database. These libraries are provided by storage vendors who imbed hardware knowledge about the tape or disk devices that they provide with their products. Other backup tools also exist, but we recommend using these only if you have a pre-existing relationship with the vendor. If you choose to use third-party tools, ensure that they fully support hot and cold backup, restore, and recovery methods for RAC databases. Also, check for any limitations in the database architectures that they support.

Having selected a suitable backup tool, you need to make a number of design decisions about the nature and location of the backups. Writing the backup files to a central location is preferable for ease of manageability. In the event of a node failure, the central location enables any remaining node to perform the recovery by looking for required files in a centralized, well-known location. If backup files are written to the local disk of one server, then a single point of failure has been introduced if that server is damaged or offline.

RMAN backups can be written either to disk or to tape. If you have sufficient space available, we recommend that you write the backup to disk initially and then copy it onto tape subsequently; this strategy is commonly called the *disk-to-disk-to-tape method* of backup and restore. Restores generally occur much faster when the database image is available online, so that a backup of the database is always available on disk. It is invariably faster to restore from a disk backup than a tape backup. The downside to this scenario is that additional disk space is required to implement a backup-to-disk strategy.

In Oracle 10*g* and above, you can optionally configure a Flashback Recovery Area, that is, an area of disk used for all backup and recovery files, including datafile copies and archived redo logs. You only need to specify a pathname and a maximum size for the Flashback Recovery Area when it is created. Oracle will manage the files contained in the Flashback Recovery Area and purge those that are no longer needed. If there is insufficient space, then Oracle will delete files starting with the oldest ones that have been backed up.

DBAs who learned their trade with Oracle 6.0 and early versions of Oracle 7 often still like to perform regular full exports of the database. While not strictly necessary, having an export file available on disk can be comforting, as it provides some level of reassurance that the database can be recovered in the event of an administrative disaster. Exporting the database by user will allow individual copies of applications to be taken independently of the entire database, which can be handy for recovering individual application components. Performing an export also provides additional validation of the blocks in the database used to store table information, since they are physically checked in the export phase.

Database blocks can become corrupted and remain unidentified until they are accessed on the live system. By performing regular exports of the database, these problems can be identified in advance of their appearance on the production system. Unlike tables where all rows are written to the export file, only the definitions of indexes are included in the export. Unfortunately, the export does not perform any validation of blocks in index segments, and therefore, index corruption is difficult to detect.

If you are performing RMAN backups, there is little value in performing additional exports. However, if you have sufficient resources, then exporting the database once a week can do no harm. Remember, however, that the export file will occupy a significant amount of space on disk. It can, of course, be compressed as it is written out to disk, thereby reducing the amount of space required. If space is at a premium, then in our opinion, make RMAN backup copies of the database rather than using the same disk space for an export.

Your backup and recovery strategy will also be limited by the availability and speed of tape devices. The optimum solution is to have separate networked tape devices that can be shared by many systems. Some users install additional network cards in each server, so that the backup can be written directly to the tape device over a dedicated network to prevent the backup interfering with other network traffic. Whether you use networked tape devices or directly attached tape devices, you should have a minimum of two to eliminate another potential single point of failure, as well as to increase the overall speed of the backup.

Archived Redo Logs

Another area for consideration is the location of the archived redo log files. We strongly recommend placing these in a central location on shared storage in a symmetrical data structure. This structure is straightforward to back up and makes recovery from any node much simpler, because the redo logs are located in a centralized location. By locating the redo log files on a shared storage system, you can take advantage of the safety, reliability, and increased performance of these storage devices.

Figure 4-3 shows the recommended configuration: a four-node RAC database where all archived redo log directories are written to shared storage.

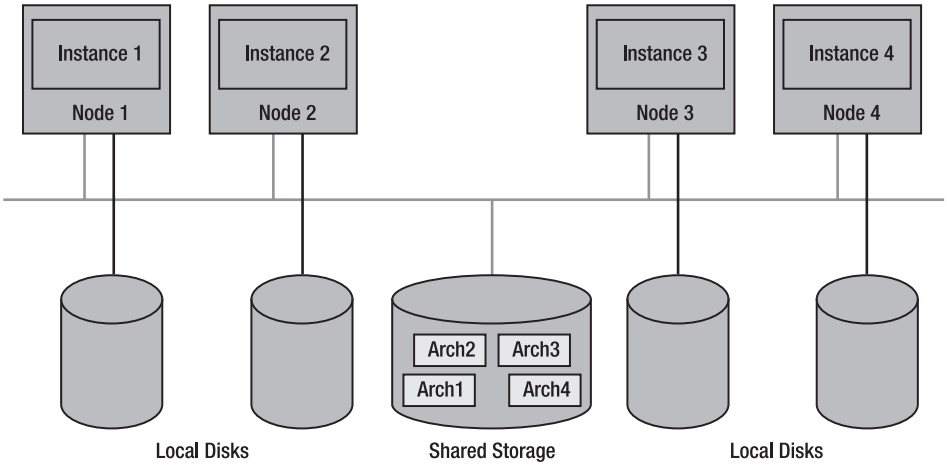

Figure 4-3. *Archived redo log storage*

If you do not have a cluster file system or ASM, then your options are more limited. One acceptable option is to use NFS for archived redo logs. Configure multiple log archive destinations to allow each server to copy its archived redo logs to two or more NFS locations on central servers. Alternatively, some users choose to locate their archived redo logs on the local disk of each server.

Figure 4-4 shows a four-node RAC cluster in which the archived redo logs are written to local disks. This configuration introduces a potential single point of failure in the event that the server dies. To circumvent this problem, some users configure a second log archive destination parameter to write the archived redo logs to the local disks of a second server. The second server eliminates the single point of failure, but it still makes recovery unnecessarily difficult. Another disadvantage of this configuration is the overhead of making duplicate I/Os for each redo log entry. This duplication places additional CPU cycles on the servers that could be better used servicing data requests.

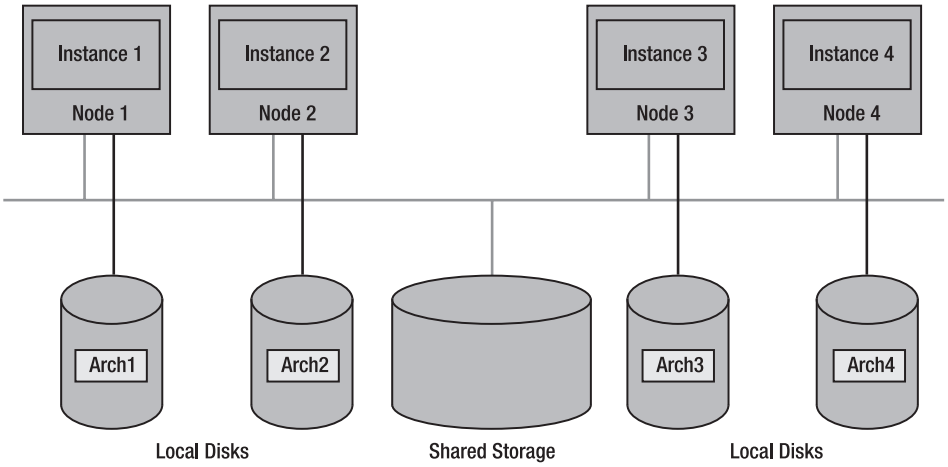

Figure 4-4. *Local archived redo log storage*

Figure 4-5 shows a four-node RAC database with a revised archived redo log configuration, in which each archived redo log is written to one local disk and one remote disk. Remote disks are mounted using NFS.

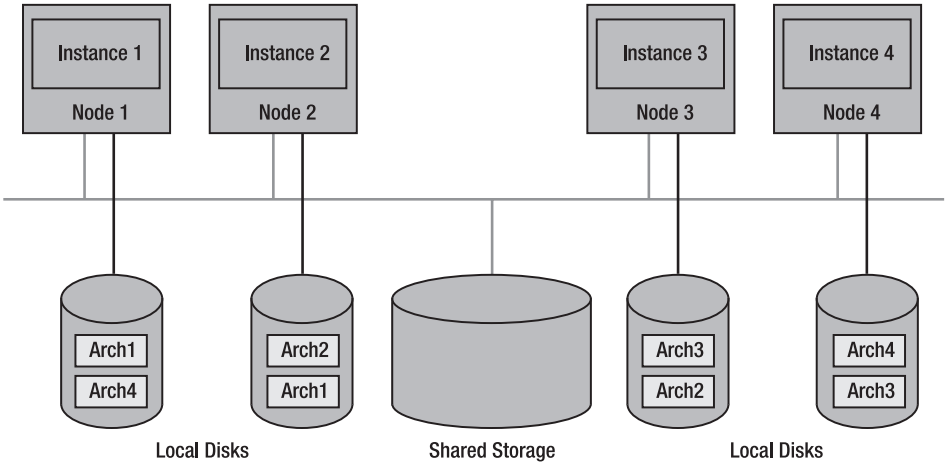

Figure 4-5. *Local and remote archived redo log storage*

Hardware Requirements

In this section, we provide an overview to some of the considerations to be given to selecting the server hardware when designing your RAC configuration. We discuss the hardware options available in Chapter 6.

While RAC will run on almost any industry-standard system configuration, these servers do not need to be of an identical configuration; however, they are much simpler to maintain if they are. Bear in mind that it is almost impossible to source identical servers, as most manufacturers change the specifications of the components on a regular basis. However, using similar hardware has several benefits:

- Initial faults are easier to isolate and fix, because components can be interchanged between servers.
- Support is simpler, and staff training costs are minimized.
- Low-level drivers, patches, and upgrades can be tested on an identically configured server, while the rest of cluster is running for subsequent rollout.

If you expect your cluster to grow over time, then it makes sense to have reliable guarantees that you will be able to source additional compatible servers.

When you are designing your cluster, make sure you understand any SLAs that are in place with your vendors, which may affect your hardware purchase. For example, in a mission-critical application, you may decide to have additional nodes available on a permanent basis in case of a node failure. For less important applications, the additional nodes may be built, but kept as hot spares, and possibly disabled to avoid paying the license fees. If you are building a grid architecture, then the additional nodes might also be deployed on other databases or used in non-mission-critical applications within the grid and redeployed when necessary. Alternatively, you might use database services to configure a node to have a primary purpose, for example, as a report server, when the cluster is running normally and a secondary purpose, such as an additional OLTP server, in the event of the failure of another OLTP node.

Application Design

If you have access to the source code of an existing system or have the advantage of writing an application from scratch, then you should consider a few application design issues, such as the use of bind variables and sequences. These issues are described in the following sections.

Bind Variables

If your application is likely to use shared SQL statements, then you should check that all statements use bind variables. Using bind variables increases the probability that individual SQL statements will reside in the library cache. Cached SQL statements can be shared and reused, consequently reducing the amount of hard parsing and reloading. It is very important that SQL statements are written identically, including the case (lower or upper), to ensure that they remain in the SQL cache for as long as possible.

Sequences

The most widely reported scalability issue in Oracle RAC is caused by sequences implemented within application tables. *Sequences* are sequential numbers generated within the database and are normally used as keys for database rows. The use of sequential numbers is a common feature in application packages that have been designed to be portable across several database platforms, as

not all operating systems provide support for sequences. Managing unique sequential numbers is a performance-intensive operation for the database. Minimizing the use of sequences and/or using the caching feature will help prevent or reduce single points of contention with the database.

Use of sequential numbers can be a highly sensitive subject within organizations. This sensitivity is often a legacy of previous systems and is usually imposed from outside the IT department. Before designing a solution to a sequence numbering problem, it is always worth examining why the sequence is required in the first place. Sometimes it is needed for legal or auditing reasons and cannot be avoided. Many other systems, however, implement sequences simply for convenience, and seeking alternative solutions may be prudent.

Sequential number generation can present a number of performance issues in both single-instance and RAC environments.

In many applications, the maximum value of a sequential number is stored in a database table. When a new value is required, the application reads the current value from the database table, increments it, and then writes the new value back to the database table. Though this approach works well in a single-user database, it does not scale well in a multinode configuration, because the current session must make a row-level lock on the row containing the current sequence number. While this lock is being held, no other sessions can update the sequence until the current session has committed or rolled back.

In Oracle 8.1.5 and above, a PL/SQL autonomous transaction can be used to generate the next sequential number. Using a PL/SQL transaction reduces the amount of time that the transaction will lock the database table but may still cause contention if a large number of sessions are attempting to update the sequence concurrently.

RAC amplifies the impact on performance of sequences implemented using database tables. When a new sequence number is acquired, the session must first perform a consistent read on the database block containing the current sequence number. If the sequence number was last updated by another instance, then a read-consistent image of the database block will need to be obtained from that instance. The sequence number can then be incremented and written back to the database block, requiring the current version of the database block, which must also be obtained from the other instance. It may be necessary to wait until any row-level lock currently being held by a session on the other instance is released. Therefore, sequential number generation implemented using database tables can cause severe contention in a RAC environment.

For many years, Oracle has provided a solution to this problem in the form of Oracle sequences, which are a type of database object. For example, an Oracle sequence can be created as follows:

```
CREATE SEQUENCE seq1;
```

The next value of a sequence can be obtained in a single operation using the `NEXTVAL` pseudocolumn as follows:

```
SELECT seq1.NEXTVAL FROM dual;
```

The highest value allocated by a sequence is recorded in the `SEQ$` data dictionary table. To avoid updating this table every time a new value is selected from the sequence, by default sequences cache twenty values. The number of values that should be cached can be specified using the `CACHE` clause. In a RAC environment, each instance maintains its own separate cache of values.

It is often a requirement that sequentially generated numbers should not include gaps. However, gaps are very difficult to avoid because of database shutdowns or application errors. If a transaction rolls back instead of committing, it does not release any sequence values that it may have generated. In addition, if caching is enabled, then the unused sequence values may be lost when the database is stopped and restarted.

As with many forms of database tuning, the best way to solve a performance problem is by not introducing performance problems into the application design. For example, you may wish to review your requirements to determine whether sequence gaps may be acceptable. Sequences are often

used to generate meaningless numbers that guarantee the uniqueness of primary keys. Occasionally, sequences will also be used to generate order numbers, invoice numbers, serial numbers, and so on, in which case they may be reflected in external systems. However, gaps in these sequences may still be acceptable as long as the individual numbers can be guaranteed to be unique, and the key relationships between database rows are preserved.

In many situations, therefore, sequence gaps do not represent a major problem. However, there may be external constraints, such as legal or accounting rules. Completely eliminating sequence gaps using Oracle sequences is not possible, as values are not reused if a transaction rolls back. However, you can limit the number of sequence gaps at the expense of performance and scalability.

In a single-instance environment, the number of unused sequence values that are lost can be minimized by specifying the following `NOCACHE` clause:

```
CREATE SEQUENCE seq1 NOCACHE;
```

However, if the `NOCACHE` clause is specified, then the `SEQ$` data dictionary table will be updated every time a new sequence value is generated, which may cause high levels of contention for blocks in the `SEQ$` table, particularly in RAC environments. Therefore, we do not recommend using `NOCACHE` for busy sequences.

In a RAC environment, each instance maintains its own cache of sequence values. Therefore, sequence values may not be allocated in ascending order across the cluster. If ascending order is a requirement, then you can to specify the `ORDER` clause for a sequence as follows:

```
CREATE SEQUENCE seq1 ORDER;
```

The `ORDER` clause guarantees that the values returned by the sequence are in ascending order. However, you should be aware that specifying the `ORDER` clause will effectively serialize the sequence across all nodes in the cluster. If the sequence is accessed frequently, this may cause contention, and hence possible performance problems between nodes that request the sequence.

Database Design

If you do not have access to the source code, then it may not be possible to modify the application. Even sites with access to the source code may regard changes to the application as being a high-risk activity if, for example, the developers have moved on to other roles. In this section, we will describe some of the ways the DBA can positively influence the resource consumption of a RAC application without access to the source code.

Cursor Sharing

You should ensure that your application uses bind variables to increase the number of SQL statements that can be shared. If it is not possible to modify your application, then you might consider using *cursor sharing*, which replaces literal values in SQL statements with temporary bind variables prior to parsing.

In a RAC environment, statement parsing is significantly more expensive because of the need to lock resources using GES, the locking mechanism that spans the nodes that belong to the RAC cluster. Cursor sharing reduces the amount of hard parsing that is necessary within the instance. It also potentially reduces the amount of memory required to store the cursor in the library cache.

Optimizer Statistics

Maintaining optimizer statistics is important, so that the CBO has sufficient information to base its decisions on when generating execution plans.

However, the statistics do not need to be completely accurate or up-to-date. In Oracle 10.1 and above, the sampling algorithm has improved to the extent that estimated statistics can satisfy the optimizer's needs. For very large tables, even statistics based on a 1% estimate can be sufficient; for smaller tables, estimates in the range of 10%–30% are common.

If tables do not change on a regular basis, then gathering the statistics for them is not necessary. Therefore, you may wish to customize your statistics gathering jobs to concentrate on volatile and growing tables.

We recommend that you use the `DBMS_STATS` package, as opposed to the `ANALYZE` statement. While the latter is still supported, it is deprecated, meaning that it will eventually go away. There are a couple of good reasons for using `DBMS_STATS`:

- Statistics can be gathered in parallel.
- Statistics are calculated correctly for partitioned tables and indexes.

Histograms

Where data is skewed, make sure that you are maintaining histograms for affected columns to allow the optimizer to make better decisions about the execution plans it is generating.

In Oracle 9.0.1 and above, if a statement includes bind variables, then the optimizer bases its decisions on the values of the bind variables supplied the first time the statement is executed, which is known as *bind variable peeking*.

In general, bind variable peeking is beneficial. However, it can lead to very inefficient execution plans in the event that the first value is atypical. If this appears to be a consistent problem, then you may need to consider creating a stored outline for the statement in question.

Dynamic Sampling

The CBO makes assumptions about the selectivity of data in joins that can lead to inefficient execution plans.

By default, dynamic sampling is enabled in Oracle 10.1 and above. When a statement is parsed, if insufficient information is available about the selectivity of columns in a join, then Oracle will scan a sample of the rows in the joined tables to calculate more accurate sample selectivity.

System Statistics

By default, the CBO generates execution plans based on the predicted number of logical I/Os that will be required. Basing execution plans on logical I/O is not entirely realistic, as some execution plans require much more CPU than others, particularly statements including built-in functions, PL/SQL subroutines, Java methods, and embedded regular expressions.

In Oracle 8.1.5 and above, you can optionally enable the gathering of system statistics over an interval of a few hours or more. The kernel records information about the relative costs of CPU and I/O operations in a table called `AUX_STAT$` in the `SYS` schema. This data is used to adjust the costs of operations when the CBO is optimizing SQL statements and may result in the generation of different execution plans. This method of generating execution plans is known as the *CPU cost model*.

We recommend that you experiment with system statistics. For most users, they have a positive effect. However, we suggest that you take a backup of the `AUX_STAT$` table prior to gathering statistics in case you decide to revert to the I/O cost model.

Locally Managed Tablespaces

Locally managed tablespaces, which were introduced in Oracle 8.1.5, use a bitmap to manage space within the tablespace. Locally managed tablespaces replaced dictionary managed tablespaces, which contained tables of used and free extents in the data dictionary. While both types update the data dictionary, locally managed tablespaces perform less work while the dictionary is locked and, consequently, are more efficient.

Locally managed tablespaces are also a prerequisite for ASSM, described in the following section. If your existing database still contains dictionary managed tablespaces, we recommend that you convert them to locally managed tablespaces during the migration process.

Automatic Segment Space Management (ASSM)

Following the hugely successful introduction of locally managed tablespaces, which use bitmaps to manage extent allocation and deallocation, the next logical step was to use bitmaps to manage free space within blocks.

Prior to Oracle 10.1, Oracle always used freelist segment space management. A *freelist* is basically a chain of block pointers that links together all the blocks with free space. When updates or deletions reduce the amount of a data below a threshold specified by the `PCTUSED` attribute, the block is returned to the freelist, which is a single linked list whose head and tail are held on the segment header. The block remains on the list until the amount of data in the block reaches the threshold specified by the `PCTFREE` attribute, at which point it is removed from the freelist. Removing the block requires a further update of the segment header.

Unsurprisingly, on volatile systems, freelist segment space management caused significant contention for the segment header. In order to reduce this contention, multiple freelists were introduced. While these allowed multiple sessions to scan for free space concurrently, they did not address the fundamental issue of segment header contention, so Oracle introduced *freelist groups*, in which each freelist header is allocated to a separate block. In a RAC environment, the number of freelist groups will usually be equivalent to the maximum number of instances in the cluster. Within each freelist group, it is still possible to have multiple freelists, thus allowing sessions to search for free space concurrently.

The introduction of freelist groups significantly reduced the amount of contention experienced for segment headers in RAC environments. However, the space allocation algorithm could still lead to hot spots in the datafiles where multiple sessions were attempting to perform inserts, updates, and deletes against the same range of blocks.

Therefore, Oracle 9.0.1 saw the introduction of ASSM. As the name suggests, the intention of this feature is to reduce the amount of work required by the DBA to configure and maintain the freelists in a tablespace. ASSM is optimized to allow transactions from different instances to concurrently insert data into the same table without contention to locate space for new records. It is used to manage space for transactions in both table and index segments. ASSM must be specified when a tablespace is created as follows:

```
CREATE TABLESPACE ts1
DATAFILE '/dddd' SIZE 10000M
EXTENT MANAGEMENT LOCAL
SEGMENT SPACE MANAGEMENT AUTO;
```

ASSM can only be specified for locally managed tablespaces. It must be specified when the tablespace is created and cannot be subsequently amended.

ASSM uses a hierarchy of bitmap blocks to manage free space. This hierarchy can be up to three levels deep. However, two levels are sufficient for all but the largest segments. Within the level 1 bitmap block, groups of bits are used to represent the amount of remaining free space on each block.

You can still specify a value for the PCTFREE attribute for ASSM tablespaces. While you also can still specify a value for PCTUSED, this value is ignored. Any values specified for FREELISTS and/or FREELIST GROUPS are also ignored.

ASSM has been designed specifically for a RAC environment. The bitmap hierarchy is structured so that each instance can use a different first level bitmap block to allocate and deallocate space within blocks, thus reducing contention. In addition, each instance will also insert new rows into a different range of blocks, also reducing contention for blocks.

While ASSM is effective for insertions, performance cannot be guaranteed for update and delete statements. For example, instance A may be inserting rows into a contiguous set of blocks, but once these rows have been committed, nothing stops instance B from updating the same rows and updating the amount of space required to store these rows. Instance A and instance B, therefore, may be contending for the same ASSM bitmap block. In addition, ASSM cannot guarantee to eliminate contention for inserts, updates, or deletes against indexes. Situations such as those described previously can only be prevented by effective application design, effective physical storage design, or, of course, luck.

Table and index scans behave differently for segments created in ASSM tablespaces. In a freelist-managed tablespace, each segment has a high water mark (HWM). The segment header contains a pointer to the last initialized block in the object. If the last block is full, then the next five blocks are initialized, and the HWM pointer is updated in the segment header. The HWM can only be reset by truncating the table. Even if all rows are deleted from the segment, the HWM will be unaffected, so a table or index potentially contains a large number of empty blocks. When the segment is scanned, each block is read sequentially from the segment header to the HWM. Therefore, even if the segment is empty, a table or index scan can result in a large number of logical I/Os.

In an ASSM tablespace, there are two HWMs, the low high water mark (LHWM) and the high high water mark (HHWM). The LHWM is equivalent to the HWM in a freelist-managed tablespace. Blocks between the LHWM and HHWM may or may not have been initialized, so their existence cannot be guaranteed. Therefore, in an ASSM tablespace, it is not possible to scan all blocks in a segment up to the HHWM. Instead, all blocks are scanned up to the LHWM. Thereafter, the bitmap blocks are used to identify blocks that have been initialized between the LHWM and HHWM, and these blocks are read individually.

In Oracle 10.1 and above, ASSM should be your default method for segment space management. You may still wish to experiment with manual space management for databases experiencing extremely high levels of concurrent insert activity; however, we recommend ASSM for most applications.

Reverse Key Indexes

ASSM can reduce the impact of hot blocks when insertion is performed by multiple instances on the same segment, by eliminating the segment header as a point of contention and by ensuring that each instance has a separate set of blocks. Eliminating this point of contention works well for unsorted or heap data. However, data often needs to be stored in an index to guarantee uniqueness or to support sort operations. In this case, the location of each entry is determined by the sort order, so techniques such as hashing cannot be used to distribute updates across multiple blocks.

Therefore, in Oracle 8.0 and above, you can create reverse key indexes, in which the data in each index column is stored in reverse order, which has the effect of distributing consecutive keys across different leaf blocks in the index. The data in the table columns is unaffected. Reverse key indexes can be created with single columns:

```
CREATE INDEX i1 ON t1 (c1) REVERSE;
```

Single column reverse key indexes are useful in situations where the key value is being generated using a sequence. You can also create a reverse key index with multiple columns, as in the following example:

```
CREATE INDEX i2 ON t2 (c1,c2) REVERSE;
```

The data in each column is reversed separately, so, in the preceding example, if column `c1` in table `t2` contains ABC and column `c2` contains DEF, then the index key will be stored as CBAFED.

You can specify the `REVERSE` clause for both nonpartitioned and partitioned indexes. You can also specify it for compressed indexes. However, you cannot use the `REVERSE` clause with an Index-Organized Table (IOT), nor can you specify it with an IOT Secondary Index.

While reverse key indexes are an excellent way of reducing contention for index blocks, they impose some limitations in the execution plans that can be generated using the index. Reverse key indexes work optimally with queries that include equality predicates on the leading edge of the index key columns. They do not perform well if the execution plan uses range scan operations, such as `INDEX (RANGE SCAN)`, against them. Therefore, the following queries might perform well against table `t2`:

```
SELECT * FROM t2 WHERE c1 = 33;
SELECT * FROM t2 WHERE c1 = 33 AND c2 = 42;
SELECT * FROM t2 WHERE c1 = 33 AND c2 > 42;
```

However, the following queries almost certainly will not perform well if the reverse key index is used:

```
SELECT * FROM t2 WHERE c1 < 33;
SELECT * FROM t2 WHERE c1 >= 33;
```

It is difficult to assess the effect of reversing an index in an existing application that might use the existing index in many different statements. Although you can create `NOSEGMENT` indexes in Oracle 8.1.5 and above to evaluate the impact of adding a new index on execution plans, this technique is does not particularly help you in assessing the effect of reversing an existing index. As with so many other RAC features, there is no substitute for testing reverse key indexes in an environment that mirrors your production database as closely as possible.

Partitioning

In Oracle 8.0, the *partitioning* option was introduced. Initially designed to address some of the problems posed by very large databases (VLDBs), partitioning offers a highly flexible solution to a number of administrative and performance problems.

Over the years, partitioning technology has evolved, but the basic principles have remained unchanged. A *nonpartitioned object* contains only one data segment; a *partitioned object* can potentially contain many data segments.

Both tables and indexes can be partitioned. Various methods of partitioning exist, including range, hash, and list partitioning. In addition, partitioned objects can be further divided into subpartitions. All partitions in the partitioned object share the same logical structure, which is defined in the data dictionary.

Partitioned objects present a number of administrative advantages. Large tables and indexes can be divided into a number of smaller segments, which can be managed independently. Individual partitions can be added, dropped, truncated, merged, and split. Data can also be exchanged between a single partition and a nonpartitioned table with the same logical structure.

Every partitioned object has a partition key that consists of one or more columns. The partition key determines the partition (or segment) that each row will be stored in. The partition key does not necessarily need to be the same as the primary key, nor does it need to be unique.

In Oracle 8.0, the only type of partitioning available was *range partitioning*, in which the partitioning key represents a range of values for each partition. Range partitioning is often used to represent data ranges and is still the most common form of partitioning in use today. An example follows:

```
CREATE TABLE orders
(
       orderkey NUMBER,
       orderdate DATE,
       custkey NUMBER,
       quantity NUMBER,
       valueNUMBER
)
PARTITION BY orderdate
(
       PARTITION 200605
            VALUES LESS THAN (TO_DATE ('31-MAY-2006','DD-MON-YYYY' ));
       PARTITION 200606
            VALUES LESS THAN (TO_DATE ('30-JUN-2006','DD-MON-YYYY' ));
       PARTITION 200607
            VALUES LESS THAN (TO_DATE ('31-JUL-2006','DD-MON-YYYY' ))
);
```

In the preceding example, the partition key is the `ORDERDATE` column. For each partition, an upper bound is defined for the partition key. All values before `31-MAY-2006` will be stored in the first partition, all values less than `30-JUN-2006` will be stored in the second partition, and all remaining values will be stored in the final partition.

An attempt to insert data into a partition that is out of range—for example, `01-SEP-2006`—will result in the following error:

```
ORA-14400: inserted partition key does not map to any partition
```

Most sites using time-based range partitions similar to the one illustrated previously manually create additional partitions as part of their maintenance procedures to ensure that a partition will always exist for the newly inserted data. For the table shown previously, for example, the next partition might be added at the end of July to ensure that space exists for the rows created during August:

```
ALTER TABLE orders
ADD PARTITION 200608
VALUES LESS THAN (TO_DATE ('31-AUG-2006','DD-MON-YYYY' ));
```

Alternatively, you can create an unbounded partition that accepts any data value above the last explicitly defined partition range. This is not particularly meaningful for time-based range partitions, but it can be appropriate for other tables with numeric values such as the following example:

```
CREATE TABLE t1
(
       c1 NUMBER,
       c2 NUMBER
)
PARTITION BY c1
(
       PARTITION p1 VALUES LESS THAN (100),
       PARTITION p2 VALUES LESS THAN (200),
       PARTITION p3 VALUES LESS THAN (MAXVALUE)
);
```

In the preceding example, all rows with a partition key greater than 200 will be stored in partition `p3`.

By default, you can update the partition key as long as the row remains in the same partition. Updates to the partition key that would cause the row to be moved to another partition will result in the following error:

```
ORA-14402: updating partition key column would cause a partition change
```

In Oracle 8.1.5 and above, you can enable movement of updated rows between partitions using the `ENABLE ROW MOVEMENT` clause:

```
ALTER TABLE t1 ENABLE ROW MOVEMENT;
```

From a performance point of view, partitioned tables perform well for queries where the partition key is known. Consider a range-partitioned table containing one partition for each month over the last five years. If there are no suitable indexes, then a query will have to perform a full table scan of all rows in each of the 60 partitions. However, if the query knows the partition key, it can use this information to eliminate unnecessary partitions and scan only a single partition. Similarly, if part of the partition key is known, then a smaller range of partitions might be scanned. Therefore, partitions have the potential to minimize the amount of I/O performed by the database.

You can create indexes against partitioned tables. These indexes may be either nonpartitioned or partitioned. If the index is partitioned, then it may be either local or global. Locally partitioned indexes have the same number of partitions as the table that they index. The columns of a locally partitioned index key must include all the columns in the partition key for the table. When a new partition is added to the table, a corresponding partition will also be added to the local index.

From a performance point of view, local indexes can be highly efficient or highly inefficient, depending on how they are accessed. In a partitioned local index, each partition contains a separate B-tree.

If, for example, the index contains one partition for each month in the last five years, there would be 60 partitions and 60 separate B-trees. When the locally partitioned index is accessed, if the partition key is known, Oracle can identify the partition directly and scan a single B-tree index. It is highly likely that the depth of each partitioned B-tree would be lower than the B-tree depth of an equivalent nonpartitioned index; therefore, using the locally partitioned index would be more efficient.

However, if the partition key is not known when the locally partitioned index is accessed, then Oracle will need to scan all 60 B-trees for the relevant partition. Depending on the number of rows in each partition, the optimizer may still decide to use this index in an execution plan, as it may still be more efficient than performing a full-table scan. It will, however, be significantly less efficient than using the equivalent nonpartitioned index.

Maintenance of local indexes is highly efficient. For example, if you drop a table partition, the corresponding local index partition will also be dropped. The remaining partitions of the local index will be unaffected and will continue to be available.

Sometimes, however, a locally partitioned index is not sufficient. For example, a table may have a secondary key that must have a unique value throughout the entire table. The secondary key cannot be efficiently enforced using locally partitioned indexes, as it would be necessary for the constraint to check every partition each time a DML statement updated any of the key columns. In this case, a globally partitioned index is more appropriate.

A *globally partitioned index* can be partitioned using a different partition key to the table that it indexes. The number of partitions can also vary. The only restriction is that a range-partitioned global index must include a default partition that is declared using the `VALUES LESS THAN (MAXVALUE)` clause. The columns of a globally partitioned index must contain the columns of the partition key.

Globally partitioned indexes can be highly efficient for index lookups. To continue the previous example, you might create a globally partitioned index with 32 partitions. Each of these partitions will contain a B-tree index. If you know the partition key when you use this index to access the table, you can perform partition elimination and access only the partition containing the relevant index entry.

However, globally partitioned indexes can cause significant problems with maintenance. As the one-to-one mapping between index and table has been lost, it is not possible to guarantee the consistency of the index when partition operations, such as DROP or TRUNCATE, are performed on the table. Therefore, when DDL is performed against the table, all global indexes on that table are marked INVALID and need to be rebuilt, which may affect performance for queries that are unable to use the global index to perform lookups and performance in terms of I/O needed to rebuild the index.

In Oracle 9.0.1 and above, the UPDATE GLOBAL INDEXES clause can be specified for DDL statements on partitioned tables:

```
ALTER TABLE t1
DROP PARTITION p1
UPDATE GLOBAL INDEXES;
```

The UPDATE GLOBAL INDEXES clause can be used with the ADD, COALESCE, DROP, EXCHANGE, MERGE, MOVE, SPLIT, and TRUNCATE partition operations. When this clause is specified, Oracle will update all global indexes whenever DDL operations are performed against that table instead of invalidating them. For example, if you need to drop a partition, each row that is dropped from the table also will be dropped from the global index.

Clearly, the UPDATE GLOBAL INDEXES clause is most effective if you are making a small number of changes to a large table. On the other hand, if you are making a significant number of changes to the table, it may still be more efficient to drop the global indexes before the operation and then re-create them afterward.

The range partitioning introduced in Oracle 8.0 was a great leap forward in terms of both administration and performance for very large tables. Many applications, particularly those written by in-house development teams, center on a single table, which can often represent more than 80% of the total data in the database. These tables are typically range partitioned by date and introduce a new performance issue. For example, if the table is partitioned by month, most updates will typically affect one partition, potentially resulting in block contention. While this block contention can be serious in a single-instance environment, it can cause severe problems in a RAC environment, where blocks must be exchanged between instances.

Therefore, in Oracle 8.1.5 and above, two new forms of partitioning were introduced: hash partitioning and range-hash partitioning. When a table is *hash partitioned*, the partition key is mapped to the appropriate partition using an Oracle internal hash algorithm. This algorithm is slightly limited in behavior, so Oracle recommends that the number of partitions in the table is a power of two, that is, 2, 4, 8, 16, 32, and so on.

Like other forms of hashing, hash clustering is only effective if you know the partition key for every access. If you do not always know the partition key and no suitable indexes exist, then Oracle will resort to a full table scan of every partition. Although there is a hash partition range scan operation, it is rarely invoked by the optimizer.

From a performance point of view, hash partitions represent a good way of distributing rows between a number of partitions, thereby reducing or eliminating potential contention. However, only a limited number of access paths are available for a hash partition, so they may not be suitable for all applications.

In Oracle 8.1.5 and above, you can also use *range-hash composite partitioned tables*. The table is initially partitioned by range, and then subpartitioned by hash keys. This partitioning presents the best of both worlds as the table is logically divided into range partitions, enabling more efficient administration, and then it is physically divided into a small number of hash partitions (usually eight or fewer), reducing contention.

In a *composite partitioned table*, a segment exists for each of the subpartitions. Data is only stored at the subpartition level. At the partition level, no segment exists, and no data is stored.

The partition key can be different at partition level to the subpartition level, meaning that if the partition key is known, but the subpartition key is not, partitions can still be eliminated, and the worst case is that each of the hash subpartitions within that partition must be scanned.

In the example of the orders table, you might create the following four subpartitions for each partition:

```
CREATE TABLE orders
(
       orderkey NUMBER,
       orderdate DATE,
       custkey NUMBER,
       quantity NUMBER,
       value NUMBER
)
PARTITION BY RANGE (orderdate)
SUBPARTITION BY HASH (custkey)
(
        PARTITION 200605
        VALUES LESS THAN (TO_DATE ('31-MAY-2006','DD-MON-YYYY' ))
        (
              SUBPARTITION 200605_1,
              SUBPARTITION 200605_2,
              SUBPARTITION 200605_3,
              SUBPARTITION 200605_4
       ),
       PARTITION 200606
       VALUES LESS THAN (TO_DATE ('30-JUN-2006','DD-MON-YYYY' ))
       (
              SUBPARTITION 200606_1,
              SUBPARTITION 200606_2,
              SUBPARTITION 200606_3,
              SUBPARTITION 200606_4
       ),
       PARTITION 200607
       VALUES LESS THAN (TO_DATE ('31-JUL-2006','DD-MON-YYYY' ))
       (
              SUBPARTITION 200607_1,
              SUBPARTITION 200607_2,
              SUBPARTITION 200607_3,
              SUBPARTITION 200607_4
       )
);
```

In Oracle 8.1.5 and above, you can also create partitioned IOTs. These have all the advantages of nonpartitioned IOTs and offer the administrative advantages of partitioned tables.

Oracle 9.0.1 saw the introduction of *list partitions*. These are useful for tables for which neither range nor hash partitioning is appropriate. For each partition, a list or set of partition keys is specified. This list of keys is useful to create partitions based on logical entities or groups of entities.

In Oracle 9.2 and above, list partitioned tables can contain hash subpartitions. The behavior of list-hash partitioned tables is similar to that of range-hash partitioned tables.

Partitioned tables offer the same administrative and performance benefits in a RAC environment as they do in single-instance databases. In an optimal configuration, each partition in a table would have an affinity with a specific instance to guarantee that blocks from a specific partition will only appear in a single buffer cache, to minimize interinstance block traffic, and to maximize the effectiveness of the buffer cache.

Prior to Oracle 10.1, if you wanted to achieve affinity between the partitions of a table and specific instances, then the simplest way was to modify the application to connect to the appropriate instance. If the required instance was not known before the connection was made, then you needed to maintain a connection to each instance. However, in the event of a node failure, you would need

to manually handle the redistribution of connections to other instances with the application. Manually redistributing connections requires a fair amount of intelligence within the application, and the resultant code can still be quite inefficient following a node failure.

In Oracle 10.1 and above, the mechanism just described has been formalized by Oracle into database services. You may still need to maintain multiple connections to each instance, but you can now define a database service for each instance and specify a backup instance in the event of the failure of the primary instance.

Therefore, in Oracle 10.1 and above, you can spread large partitioned tables evenly across all available instances. Node failures can be handled by Oracle Services rather than by application-specific code, which leads to the most efficient use of the buffer cache as blocks subject to update within the partitioned table will normally only appear in the buffer cache of one instance.

Summary

In this chapter, we reviewed the factors that should influence the design stage of your RAC implementation. We considered the business requirements and architectural decisions that you should take into account and some of the design decisions to make for storage and hardware selection. Finally, we looked at the areas to note when designing your application and database for RAC.