Oracle RAC Design Techniques

The proper design of a Real Application Clusters environment is critical for all Oracle professionals. This book excerpt (http://www.rampant-books.com/book_2004_1_10g_grid.htm) show expert tips and techniques used by real-world RAC professionals for designing a robust and scalable RAC system.

This is an excerpt from the top RAC book Oracle 10g Grid & Real Application Clusters (http://www.rampant-books.com/book_2004_1_10g_grid.htm), by Mike Ault and Madhu Tumma.

Introduction to Grid Design

This chapter focuses on the issues that must be considered when designing for RAC. The reasons for utilizing RAC must be well understood before a proper implementation can be achieved.

Essentially, there are only a couple of reasons to use RAC. RAC spreads the load across multiple servers, provides high availability, and allows larger SGA sizes than can be accommodated by a single Oracle10g instance, on Windows2000 or Linux implementations, for example.

The most stringent design requirements come from the implementation of high availability. A high availability RAC design must have no single point of failure, a transparent application failover, and reliability, even in the face of disaster at the primary site. A high availability design requires attention to equipment, software, and the network. This three-tier approach can be quite daunting to design.

The following sections provide a look into two key design considerations. The first is the design of the equipment needed to support a high availability (HA) RAC configuration. Next, the methods of configuring RAC instances in a RAC cluster to meet performance and HA requirements must also be considered.

Designing Equipment for Real Application Clusters

The most important design feature of the equipment used in HA RAC clusters is a layout that eliminates any single point of failure. The diagram in Figure 12.1 provides an opportunity to look at some design features and their impact on the potential success of HA RAC clusters in the environment.

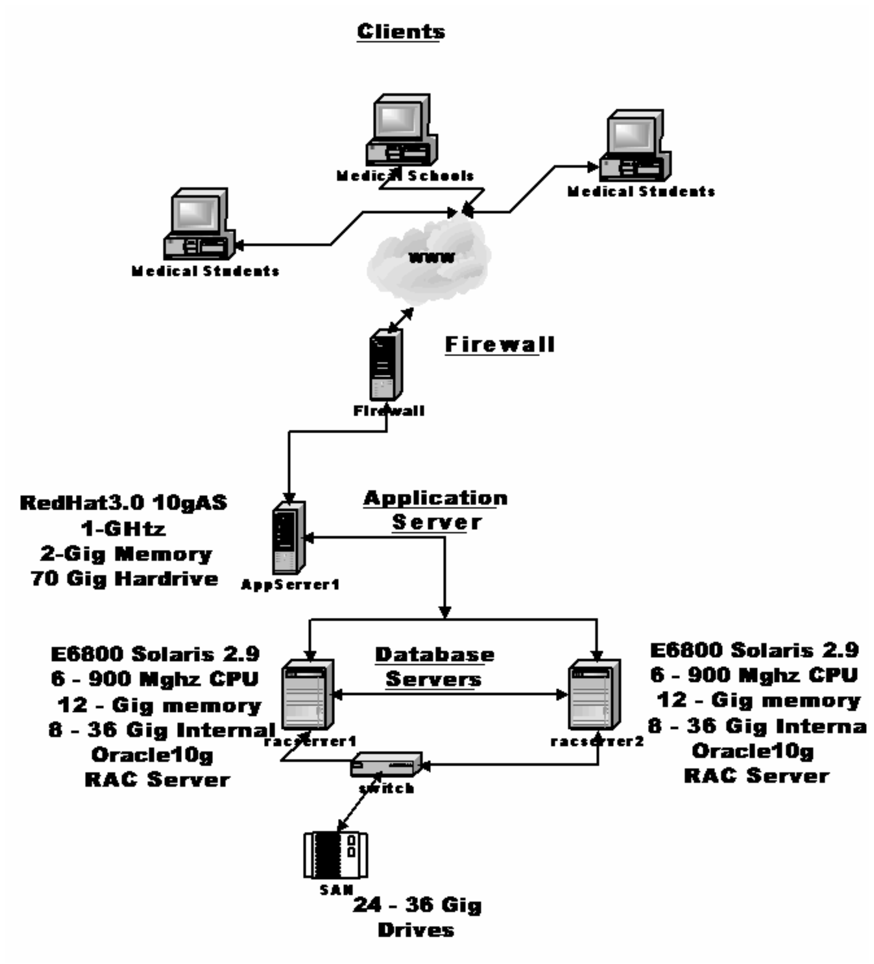

Figure 12.1: *Non-Redundant Configuration*

Figure 12.1 shows a typical RAC configuration. However, this configuration, other than the RAC cluster itself, has no redundancy and many single points of failure. The single points of failure are:

- Firewall
- Application Server
- Fabric Switch
- SAN array

A failure of any one of these single points will result in system unavailability, no matter how well the RAC cluster itself is laid out, designed and tuned.

It is critical to ensure that there is no single point of failure in a high availability configuration. Figure 12.2 illustrates exactly what eliminating single points of failure means.

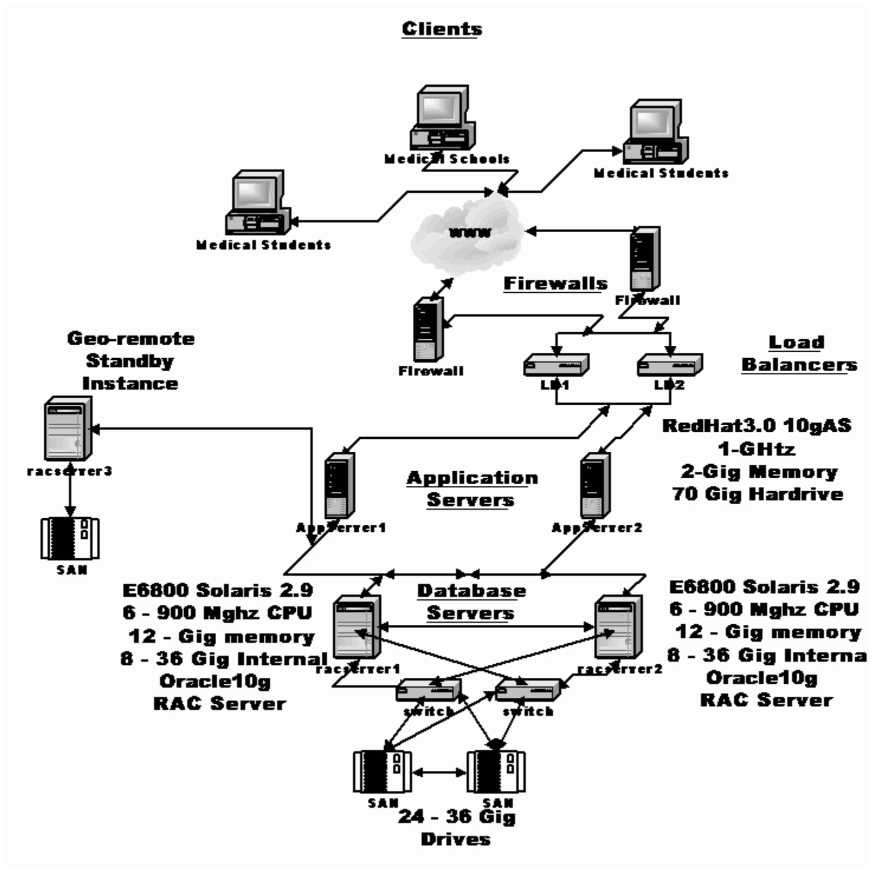

Figure 12.2: *Example of a Redundant RAC Configuration*

The system shown in Figure 12.2 has had the following redundancies added:

- A second firewall with an independent connection to the web.
- A second application server.
- A second fabric switch with redundant pathways.
- A second SAN array.
- A set of load balancers.
- A geo-remote RAC Guard configuration.

Now, the single points of failure in Figure 12.1 have been eliminated. A third server has also been added, as well as a SAN array in a geographically remote location. This third server and SAN ensure that not even a disaster at the primary location will bring the application down. The application server and firewall for this third server are not shown and may not be required if the firewalls and application servers are in a different location from the database servers.

For highly active websites, the application servers may include web caches as well as OC4J servers. The servers that run OC4J are the building blocks of the application tier and provide the application service to the clients. OC4J is J2EE compliant and includes:

- A JSP translator.
- A servlet container.
- An EJB container.
- Several other Java specifications.

OC4J clustering connects servers that run individual containers so that they act in tandem. This provides more availability and scalability than a single Oracle instance can provide. Additional OC4J instances can be added or removed transparently to increase scalability or do system maintenance. The OC4J cluster at the primary site has identical hardware, OS, application server software, and application-specific software to the secondary site. They are configured similarly in all respects.

By having Oracle10gAS Web Cache Cluster and load balancers in front of the OC4J clusters, performance, scalability, and availability of a web site are increased. Web Cache eliminates the need to repeatedly process requests on the application web server by storing frequently accessed URLs in memory. Multiple instances of Web Cache can be configured to run as members of a cache cluster. This provides:

- Higher scalability of the website by allowing more content to be cached and more client connections.
- Higher availability by detecting failures and failing over of caches.

Because each OC4J instance and cache cluster member is redundant on multiple sets of machines, problems and outages within a site are transparent to the client. The OPMN monitoring infrastructure ensures nearly uninterrupted availability of an application's services by automatically detecting and restarting failed components of the application tier or the Web Cache.

In Figure 12.2, the use of a RAC server platform capable of CPU partitioning is shown. This means that multiple CPUs, in systems like the E6800 from SUN, can be separated into virtual servers that act like independent servers. This partitions malfunctioning CPUs or memory, which is on the CPU cards in the E6800, and the rest of the server continues to work.

In addition, the SAN, perhaps a SUN T3, Hitachi or EMC Clariion or Symmetrix, can be configured using redundant disk configurations such as RAID-1, RAID5, or a combination, sometimes called plaid, of RAID1 and RAID5 that virtually eliminate downtime from the loss of a single disk. It should be stressed that application performance can suffer horribly from a disk failure during either a disk rebuild with installed spares or a rebuild of the information using parity information from the other disks in a RAID5 set.

What Are the Effects of Component Failure?

This section will provide a quick look at the effects of component failure.

Failure of the Internet or Intranet

While not a component that a DBA usually has control over, failure of the Internet connection, usually due to the provider having a failure, means no one outside the company can access the application. Failure of the intranet or internal networks means no one inside the company can access the application. These components, usually comprised of multiple components, should also have built in redundancy.

Failure of the Firewall

The firewall acts as the gatekeeper between the company's assets and the rest of the world. If the database is strictly internal with no connection to the web, a firewall is not needed. If there is only one firewall, a failure will prevent anyone outside the firewall, such as the rest of the universe, from contacting the database. Internal users, those inside the firewall, may still have access and some limited processing can occur.

Failure of the Application Server

The application server usually serves the web pages, reports, forms, or other interfaces to the users of the system. If there is only a single application server and it goes down, even if the database is fully functional, there is no application to run against it. A failed application server without redundancy means no one can use the database, even if all other components are still functional. This also applies to single web cache servers or OC4J servers.

Failure of the Database Server

The failure of the database server is the one failure that is taken care of in a normal RAC configuration. Failure of a single database server leads to failover of the connections to the surviving node. While not a critical failure that will result in loss of the system, a single server failure means a reduction in performance and capacity. Of course, a catastrophic failure of both servers will result in total loss of service.

The servers will have disk controllers or interfaces that connect through the switches to the SAN arrays. These controllers or interfaces should also be made redundant and have multiple channels per controller or interface. In addition, multiple network interface cards (NICs) should also be redundant, with at least a single spare to take the place of either the network connection card or the cluster interconnect should a failure occur.

Failure of the Fabric Switch

The fabric switch allows multiple hosts to access the SAN array. Failure of the fabric switch or other interconnect equipment can result in loss of performance or total loss of the application. If the SAN cannot be accessed, the database will crash and no one will be able to access it, even if all other equipment is functional.

SAN Failure

SAN failure can come in many forms. Catastrophic failure will, of course, result in total loss of the database. Failure of a single drive, if there is no hot spare or if the hot spare has been utilized, will result in severe performance degradation, by as much as 400-1000 percent, in a RAID5 situation where the data on the failed drive has to be rebuilt on the fly from parity information stored on the other drives in the RAID5 set. Even if there is an available hot spare, it still takes time to rebuild this hot spare from the parity data on the other drives. During this rebuild, performance will suffer.

Usually, SANs are configured with disk trays or bricks of a specific number of drives. This is usually comprised of eight active and a single spare in each tray or brick. A single tray becomes an array, in the case of a RAID0+1 setup, the array will be striped across the eight drives and would be mirrored to another tray in the array. Failure of a RAID0+1 drive has little effect on performance, as its mirror drive takes over while the hot spare is rebuilt on an "on available" basis. In a RAID5 array, the eight drives are usually set up in a 7+1 configuration, meaning seven drives in a stripe set and one parity drive.

When a drive fails, there must be an immediate spare available to replace it, even if the hot spare is available. If the hot spare has already activated and a second drive is lost, the entire array is in jeopardy. Most of these arrays use hot pluggable drives, meaning they can, in time of failure, be replaced with the system running.

NICs and HBAs

While not shown in the diagram, every component requires a connection to the others. This connection is usually via a network interface card (NIC) or host bus adapter (HBA) interface. These NIC or HBA interfaces should be the fastest possible, especially in the case of the cluster interconnect and disk connect. Failed NIC interfaces result in the loss of that component, unless a second NIC card is immediately failed over to. A failure of the HBA results in loss of connection to the disk array. At a minimum, a spare NIC and HBA for each and every component must be available. Wherever possible, use interchangeable NIC and HBA interfaces.

Provide Redundancy at Each Level

It is easy to see that redundancy at the hardware level is vital. At each level of the hardware layout an alternate access path must be available. Duplicating all equipment and configuring the automatic failover capabilities of the hardware reduce the chances of failure to virtually nil. It is also critical to have spares on hand for non-redundant equipment such as NIC and HBA cards and interface cables.

By providing the required levels of redundancy, the system becomes highly available. Once there is an HA configuration, it is up to the manager to plan any software or application upgrades to further reduce application downtime. In Oracle Database 10g using grid control, rolling upgrades are supported, further increasing reliability. At the SAN level, appropriate duplication software such as Veritas must be used to ensure the SAN arrays are kept synchronous. Oracle Database 10g allows for use of the Oracle Automatic Storage Management or ASM. ASM allows for automated striping, backup and database flashback capability.

Designing for High Performance

Designing for high performance means that every tier of the design must eliminate contention for resources. If there is no contention, each process gets the resources it needs to perform optimally. Resources fall into multiple categories: physical, as in disk and network; and internal, such as CPU speed and memory capacity. Designing for performance means utilizing these resources properly and not relying on memory or disk resources to make up for poor application design.

As with normal databases, the application design will drive performance. The finest equipment will not make up for a poor application design. In the days of Oracle Parallel Server (OPS), Oracle recommended partitioning the application data and the servers to make OPS perform. Now, Oracle salesmen insist that any application can be run with RAC, with no need for changes.

To add capacity, the solution is to just add a server and bingo, more capacity is provided and more users, no matter what they do with the application, can connect. This all sounds wonderful until the manual, Oracle9i Real Application Clusters Deployment and Performance, Release 1 (9.0.1)", Part No. A96598-01, Chapter 3, Scaling Applications for Real Application Clusters, and Chapter 4, Scaling Applications for Real Application Clusters, is consulted. There, Oracle recommends partitioning both the application data and the users to optimize performance. This unfortunate fact is omitted from the 9.2 and 10g versions of the manual which seems to indicate the automated tuning features of 9.2 and 10g relieve the DBA of this arduous task.

Of course, the difference is that now partitioning is based on reducing intra-node block pinging between instances over the cluster interconnects, instead of reducing disk pinging. At most, a factor of 40 improvements can be expected for the same application running on a RAC server versus an OPS-based server. Once the various system latencies are added, the speed difference between memory operations and disk operations falls to approximately a factor of 40 between a disk ping and an intra-node memory ping. The speed is dependent upon the speed of the interconnect. Still, a factor of 4000% (40) is nothing to sneeze at.

Designing applications for better performance on RAC involves:

- Assigning transactions with similar data access characteristics to specific nodes.
- Creating data objects with parameters that enable more efficient access when globally shared.
- Automating free space allocation and de-allocation through the use of locally managed tablespaces.
- Automating block management through local freelist management.
- Using sequences properly by using sequence ranging triggers for each instance.
- Optimizing all SQL in the application.
- Understanding the workload on the system and planning the application to properly utilize resources.

These factors should be considered for proper application design one-by-one in order to see how they can be utilized to make RAC applications perform optimally.

Compartmenting Transactions to Specific Nodes

One obvious method of reducing pings between instances is to isolate the transactions that use a specific data set to a specific server in the RAC cluster. For example, in a multiuse instance that contains multiple applications, isolate each application's user logins to a specific node. For example, sales users use the sales node, accounting uses the accounting node, and so on. In the case of a node failure, the users switch to one of the other nodes.

This compartmenting of transactions is difficult to implement for a large, multiuse RAC database where many different groups of users use each of the applications on the RAC cluster. In the case of a multi-use instance that is used by almost all corporate users, other techniques must be employed to optimize performance.

Creating Efficient RAC Data Objects

The creation of efficient RAC data objects entails using storage wisely in accordance with good RAC practices. Some of the good RAC practices, to put it quite frankly, waste disk and memory space to improve data sharing and dispersal characteristics.

An example of an efficient RAC object is one that is used by only a single instance at a time. To achieve this singularity of use, the rows-per-block (RPB) of the data object must be reduced. By reducing the RPB, the chances that multiple instances in the RAC cluster will require data in the same block are decreased. The following are several techniques that can be used to reduce the RPB in a RAC data object:

- Use a smaller block size for objects that will be used across several instances.
- Adjust *pctfree* to restrict the RPB since a higher *pctfree* will reduce the RPB.
- Use data fillers, for example, CHAR data types can be used to artificially extend the size of a data row, thus reducing RPB.

Other techniques to improve the efficiency of data objects include:

- Use as few indexes as possible to optimize data retrieval from the RAC data objects. Index node contention is the largest source of intra-node block pings, or as Oracle calls them, intra-instance block transfers. Index maintenance causes a great deal of intra-node pinging.
- Use automated freelist management.
- For high insert objects, pre-allocate extents to avoid dynamic space management. Assign allocated extents to specific instances. This avoids intra-instance block transfers during insert activity from multiple nodes. For example:

  ```
  ALTER TABLE ad_proofs
  ALLOCATE EXTENT ( SIZE 200K
  DATAFILE '/usr/u01/oradata/addb/ad_file3.dbf'
  INSTANCE 2);
  ```

- Use locally managed tablespaces to avoid *uet$* and *fet$* block pinging between instances.
- Use reverse-key indexes for indexes that may become right-hand indexes due to high insert rates. This removes the capability to use index scans. Use only when required.
- Design indexes such that the clustering factor is as close to the number of used blocks as is possible. Testing various column orders in concatenated indexes does this. In single column indexes required for SQL optimization, consider re-ordering the table in index order to reduce clustering factor. This technique can result in hot blocks and is the reverse of the previous suggestion to use reverse-key indexes, which actually increases the clustering factor.

This may seem perplexing, since some of the suggestions are contradictory. The correct approach depends on the specific tuning situation. In a situation where hot blocking is occurring in that multiple instances want the same index block because all of the current data entries are indexed there, randomizing the index nodes will reduce the possibility of intra-node index block pinging.

This is demonstrated in Figure 12.3 below. In the case where the data referenced in a single index block is needed by a single instance, the number of data blocks required is reduced by concentrating the data into as small a number of data and index nodes as possible. This reduces the intra-node pinging of data blocks. This is demonstrated in Figure 12.4. So in the first case shown in Figure 12.3, intra-node index block pinging is reduced, and in the second shown in Figure 12.4, the intra-node pinging of data blocks is reduced. The appropriate technique will have to be determined for each tuning situation.

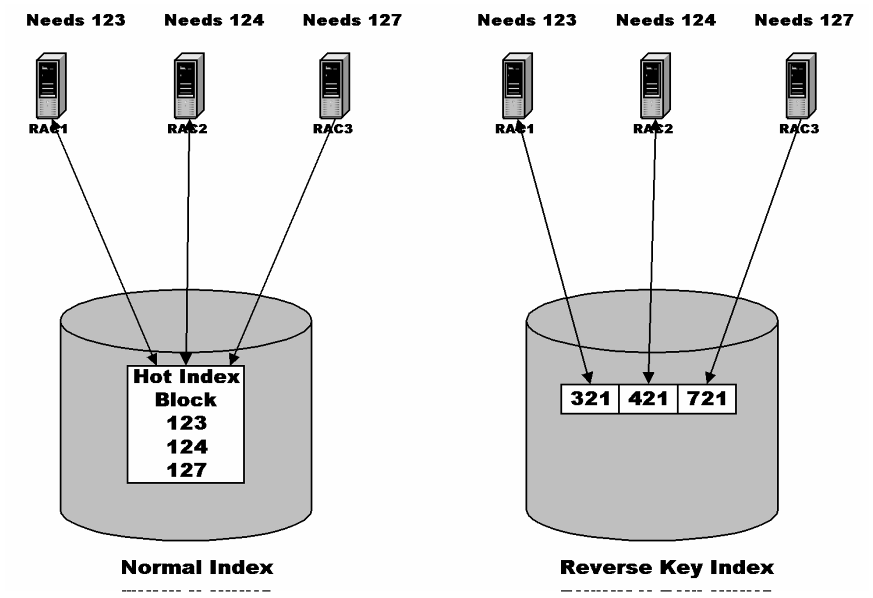

Figure 12.3: *Example of a Reverse Key Index*

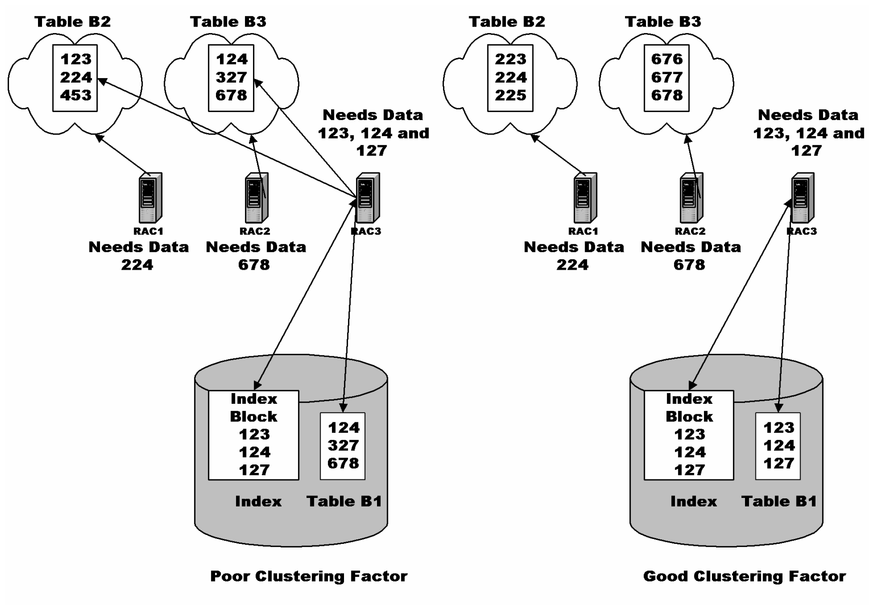

Figure 12.4: *Clustering Factor Affects*

Figure 12.3 illustrates that a hot index block, such as an index containing sequential data, by date or number, that is required by multiple nodes, increases the intra-node index block pinging, as the nodes transfer the index block to perform data lookups.

This demonstrates the effects of a poor clustering factor when the data in a table is poorly ordered, thus spreading data blocks containing index-sequential data across multiple instances. This random scattering of data into data blocks results in the possibility of multiple data blocks being required by a single instance to satisfy a range type query against a single index node. The result is the false pinging of data blocks between instances.

Neither situation is desirable. Use the reverse-key index method when multiple instances will require access to current data referenced in ascending numeric key or date key indexes. Use method two when data is usually accessed by an index range scan for specific periods from the individual instances. Method two is also good for concatenated indexes.

What about character-based indexes? By design, character-based indexes will always be random unless the table is sorted in character index order. This is because a character-based index organizes the data reference pointers in alphabetical order, while the data will usually be in natural insert order, based on the mean frequency charts for the specific language the database uses. Character-based indexes will always be skewed towards those letters that appear most frequently in the leading portions of the words for a given language. In English the letters N, R, S, L, and T lead the frequency charts.

Proper Sequence Usage

If not used properly, sequences can be a major headache in RAC. Generally speaking, sequences should be either CACHED or ORDERED, but not both. The preferred sequence is a CACHED, non-ordered sequence. If the ordering of sequences is forced, performance in a RAC environment will suffer unless ordering the sequence to a single node in the RAC cluster isolates insert activity.

Another method to optimize the use of sequences is to use a staggered sequence insert trigger. A staggered sequence insert trigger is a specific constant added to the sequence value based on the instance number. This isolates each set of inserts and prevents inadvertent attempts to use the same sequence number. An example of a staggered sequence insert trigger is shown in the following script:

```
CREATE TRIGGER insert_EMP_PK
 BEFORE insert ON EMP
 FOR EACH ROW
DECLARE
 INST_ID NUMBER;
 SEQ_NUM NUMBER;
 INST_SEQ_ID NUMBER;
BEGIN
 select
    INSTANCE_NUMBER INTO INST_ID
  FROM
    V$INSTANCE;
  select
    EMP_ID_SEQ.NEXTVAL INTO SEQ_NUM
  FROM
    DUAL;
  INST_SEQ_ID:=(INST_ID-1)*100000 + SEQ_NUM;
  :NEW.EMP_ID:=INST_SEQ_ID;
END;.
```

A staggered sequence trigger will insert the values into indexes such that each instance's values are staggered to prevent index node intra-node transfers. The formula to allow this is:

```
index key = (instance_number -1)* 100000+ Sequence number
```

Generally, sequences can be cached with cache values as high as 200 in RAC. This is much higher than for a regular Oracle instance. If there is insufficient caching, contention can result and will show up as an increase in service times. If there are performance problems due to sequences, the *row cache locks* statistics in the *v$system_event* view should be examined to determine whether the problem is due to the use of Oracle sequences. This is discussed in the following points:

- A problem with sequences is indicated in a *v$system_event* as an extended average wait time for *row cache locks* in the range of a few hundred milliseconds.

The proportion of time waiting for *row cache locks* to the total time waiting for non-idle events will be relatively high.

- In the *v$rowcache* view, for the *dc_sequences* parameter, the ratio of *dlm_conflicts* to *dlm_requests* will be high. If this ratio exceeds 10 to 15% and the *row cache lock* wait time is a significant portion of the total wait time, it is likely that the service time deterioration is due to insufficiently cached sequences. For example:

```
SQL> col parameter format a15

SQL> select parameter, dlm_conflicts, dlm_requests, dlm_conflicts/dlm_requests
ratio
   2 from v$rowcache
   3 where dlm_requests>0;

PARAMETER        DLM_CONFLICTS   DLM_REQUESTS         RATIO
--------------- -------------- --------------   -----------
dc_sequences                 0              2             0
```

In some applications, the sequence numbers used must be sequential. An example would be the line numbers for a purchase order or perhaps check numbers for an automatic check printer. In this case, a sequence table may have to be used to store the highest sequence number. The value is read from the sequence table, increased by one, and then updated. While all of this occurs, the row for the sequence being used is locked, thus no one else can use it. If this type of logic must be used, the table should be placed in a tablespace with a 2048 block size and the *pctfree* should be set high enough that only one row per block is allowed. This will reduce the chances of intra-instance pinging, assuming there is more than one sequence stored in the table.

Tablespace Design in Real Application Clusters

The goal in tablespace design is to group database objects according to their data access distribution patterns. The dependency analysis and transaction profiles of the database must be considered, and then tablespaces are divided into containers for the following objects:

- Frequently and randomly modified tables and indexes belonging to particular functional areas.
- Frequently and randomly modified tables and indexes with a lower probability of having affinity to any functional area.
- Tables and indexes that are mostly READ or READ-ONLY and infrequently modified.

The following criteria must also be considered for separating database objects into tablespaces:

- Tables should be separated from indexes.

- Assign READ-ONLY tables to READ-ONLY tablespaces.
- Group smaller reference tables in the same tablespace.

Using this strategy to group objects into tablespaces will improve the performance of Oracle's dynamic resource mastering. Oracle's dynamic resource remastering by datafiles algorithm re-distributes GCS resources to the instance where they are needed most. This remastering strategy is designed to improve resource operation's efficiency. Oracle remasters cache blocks to the instance with which the cache blocks are most closely associated based on historical access patterns. As a result, resource operations, after remastering, require minimal communication with remote instances through the GES and GCS.

Extent Management and Locally Managed Tablespaces

Allocating and de-allocating extents are expensive operations that should be minimized. Most of these operations in Real Application Clusters require inter-instance coordination. A high rate of extent management operations can adversely affect performance in Real Application Clusters environments more than in single instance environments. This is especially true for dictionary-managed tablespaces.

Identifying Extent Management Issues

If the *row cache lock* event is a significant contributor to the non-idle wait time in *v$system_event*, there is contention in the data dictionary cache. Extent allocation and de-allocation operations could cause this. *v$rowcache* provides data dictionary cache information for *dc_used_extents* and *dc_free_extents*. This is particularly true when the values for *dlm_conflicts* for those parameters increase significantly over time. This means that excessive extent management activity is occurring.

Minimizing Extent Management Operations

Proper storage parameter configuration for tables, indexes, temporary segments, and rollback segments decreases extent allocation and de-allocation frequency. This is accomplished using the *initial*, *next*, *pctincrease*, *minextents*, and *optimal* parameters.

Using Locally Managed Tablespaces

Extent allocation and de-allocation overhead can be greatly reduced if locally managed tablespaces are used. For optimal performance and the most efficient use of space, segments in locally managed tablespaces should ideally have similar space allocation characteristics. This enables the tablespace to be created with the proper uniform extent size that corresponds to the ideal extent size increment calculated for the segments.

For example, tables with relatively high insert rates can be placed in a tablespace with a 10MB *uniform extent size*. On the other hand, small tables with limited DML activity can be placed in a tablespace with a 100K *uniform extent size*. For an existing system, where tablespaces are not organized by segment size, this type of configuration can require significant reorganization efforts with limited benefits. For that reason, compromise by making most of the tablespaces locally managed, with AUTOALLOCATE instead of UNIFORM extent allocation.

Minimizing Table Locks to Optimize Performance

For RAC, table locks are coordinated through global inter-instance communication. Due to the fact that properly designed applications do not need to lock entire tables, table locks can be disabled to improve locking efficiency with minimal adverse side effects. Essentially, there are two methods for disabling table locks:

- Disabling table locks for individual tables.
- Setting *dml_locks* to zero.

The following sections will examine these methods.

Disabling Table Locks for Individual Tables

To prevent users from acquiring individual table locks, the following statement can be used:

```
ALTER TABLE table_name DISABLE TABLE LOCK
```

When users attempt to lock tables with disabled locks, they will receive an error. To re-enable table locking after a transaction, the following statement can be used:

```
ALTER TABLE table_name ENABLE TABLE LOCK
```

Using this syntax forces all currently executing transactions to commit before enabling the table lock. The statement does not wait for new transactions to start after issuing the ENABLE statement. The disadvantage to this statement is that it must be executed for all tables that may experience improper locking.

To determine whether a table in the schema has its table lock enabled or disabled, the *table_lock* column in the *user_tables* data dictionary table should be queried. If SELECT privilege is on *dba_tables*, the table lock state of other user's tables can be queried as well. The *all_tables* views can be used to see the locking state of tables for which a user has been granted SELECT privileges.

Setting *dml_locks* to Zero

Using the *dml_locks* initialization parameter, table locks can be set for an entire instance. This will disable the DROP TABLE, CREATE INDEX and LOCK TABLE commands. If these commands are not needed, *dml_locks* should be set to zero to minimize lock conversions and achieve maximum performance. DDL statements cannot be executed against tables with disabled locks.

SQL*Loader checks the flag to ensure that there is not a non-parallel direct load running against the same table. The use of direct load forces Oracle to create new extents for each session.

If *dml_locks* are set to zero on one instance, it must be set it to zero on all instances. If non-zero values are used with the *dml_locks* parameter, the values need not be identical on all instanc es.

Performance for Object Creation in Real Application Clusters

In any Oracle database, DDL statements should be used for maintenance tasks, not during normal system operations. For example, global temporary tables or PL/SQL tables should be used rather than permanent tables for reports. If this guideline is followed, in most systems, the frequency of DDL statements should be low.

If the application has to create objects frequently, performance degradation to the RAC environment will occur. This is due to the fact that object creation requires inter-instance coordination. A large ratio of *dlm_conflicts* to *dlm_requests* on the *dc_object_ids* row cache in *v$rowcache,* the same SELECT as was used for sequences will work here as well, along with excessive wait times for the *row cache lock* event in *v$system_event*, is indicative that multiple instances in the cluster are issuing excessive amounts of concurrent DDL statements.

About the only method to improve object creation performance in these situations is to set event 10297 so that it caches *object_id* values. This will improve concurrent object creation by eliminating the recursive DDL and some of the intra-instance pinging. To set event 10297, the following line can be added to the initialization parameter file:

```
event="10297 trace name context forever, level 1"
```

If the additional level argument is set to one, the caching behavior is automatically adjustable internally. Otherwise, the level can be set to the desired cache size.

Conclusion

Cache fusion in Oracle10g RAC introduces an improved diskless algorithm that handles cache coherency more efficiently than Oracle's earlier disk-ping-based architectures. This enables simpler database designs to be implemented while achieving optimal performance. However, great care must be taken to select the fastest interface and network components to get optimal performance from the cluster interconnect in order to realize the true benefits of Oracle's improvements.

Designing for true high availability starts with redundant hardware. If there are multiple single-points of failure, the finest RAC implementation in the known Universe will do little to achieve high availability.

The response time and throughput requirements placed on the system by service-level agreements and customer/client expectations ultimately determine whether a data and functional partitioning strategy should be implemented and how stringent the strategy must be. The response time and throughput needs for the application also determine how much effort needs to be invested to achieve an optimal database design.

To determine how to allocate work to particular instances, start with a careful analysis of the system's workload. This analysis must consider:

- System resource consumption by functional area.
- Data access distributions by functional area.
- Functional dependencies between application software components.

Proper implementation of a strategy that considers these points will make the system more robust and scalable.

The old 80/20 rule applies here; 80% or more of the overhead results from 20% or less of the workload. If the 20% is fixed by observing some simple guidelines, tangible benefits can be achieved with minimal effort. Workload problems can be corrected by implementing any or all of the following:

- Use Oracle automated free list management, or define free list groups for partitioned, as well as non-partitioned, data that is frequently modified.
- Use read-only tablespaces wherever data remains constant.
- Use locally managed tablespaces to reduce extent management costs.
- Use Oracle sequences to generate unique numbers and set the CACHE parameter to a high value, if needed.
- Use sequence staggering to help prevent index block contention.

- If possible, reduce concurrent changes to index blocks. However, if index key values are not modified by multiple instances, or if the modification rate is not excessive, the overhead may be acceptable. In extreme cases, techniques like physical table partitioning can be applied.

References

Metalink Note: 226880.1: *Configuration of Load Balancing and Transparent Application Failover*

OOW 2003 Presentation 32035: *Oracle 9i RAC: A Case Study, Steve Teeters, Science Applications International Corporation, Bobby Hampton, Science Applications International Corporation*

OOW 2003 Presentation 32949: *Oracle 9i Real Application Clusters Helping Southwest Airlines Take Off, Kerry Schwab. Director IT, Southwest Airlines*

OOW 2003 Presentation 33433: Oracle RAC and Linux in the real enterprise, Mark Clark, Director, Merrill Lynch Europe PLC, Global Database Technologies

SEOUC 2003 Presentation: *A Bare Bones RAC Installation, Michael R. Ault, Senior Technical Management Consultant, TUSC Inc - The Oracle Experts*

UKOUG 2001 Presentation: *Advantages of Oracle9i Real Application Clusters, Michael R. Ault, Senior Technical Management Consultant, TUSC Inc - The Oracle Experts.*

Oracle Real Application Clusters Deployment and Performance Guide 10g Release 1 (10.1), Part Number B10768-01

Oracle Real Application Clusters Administrator's Guide 10g Release 1 (10.1) Part Number B10765-01

Oracle Real Application Clusters Installation and Configuration Guide 10g Release 1 (10.1) for AIX-Based Systems, hp HP-UX PA-RISC (64-bit), hp Tru64 UNIX, Linux, Solaris Operating System (SPARC 64-bit) Part No. B10766-01

OOW 2003 Presentation 32539: Maximum Availability Architecture
Oracle's Recipe For Building An Unbreakable System, Ashish Prabhu, Douglas Utzig

High Availability Systems Group, Server Technologies, Oracle Corporation

OOW 2003 Presentation 32630: R*eal-World Performance of Oracle9i Release 2, Andrew Holdsworth, Director of Real World Performance, Server Technologies, Oracle Corporation.*

Building Highly Available Database Servers Using Oracle Real Application Clusters, An Oracle White Paper, May, 2002