An Oracle White Paper
April 2013

Oracle RAC Database aware Applications - A Developer's Checklist

Contents

Introduction

This document is intended to be a checklist for Developers who expect their applications to run on Oracle Real Application Clusters (RAC) database. The goal is to provide a set of guidelines to ensure that the application will work functionally in a RAC environment, be performant and scalable and leverage RAC's HA capability. The primary focus is on application design/development and it is assumed that the RAC backend configuration will follow general best practice guidelines in this regard. The primary version scope of this document is Oracle database 11.x and Clients using libraries that come along with the above version.

The intended audience is Application Developers, Database Administrators.

Terms and Abbreviations

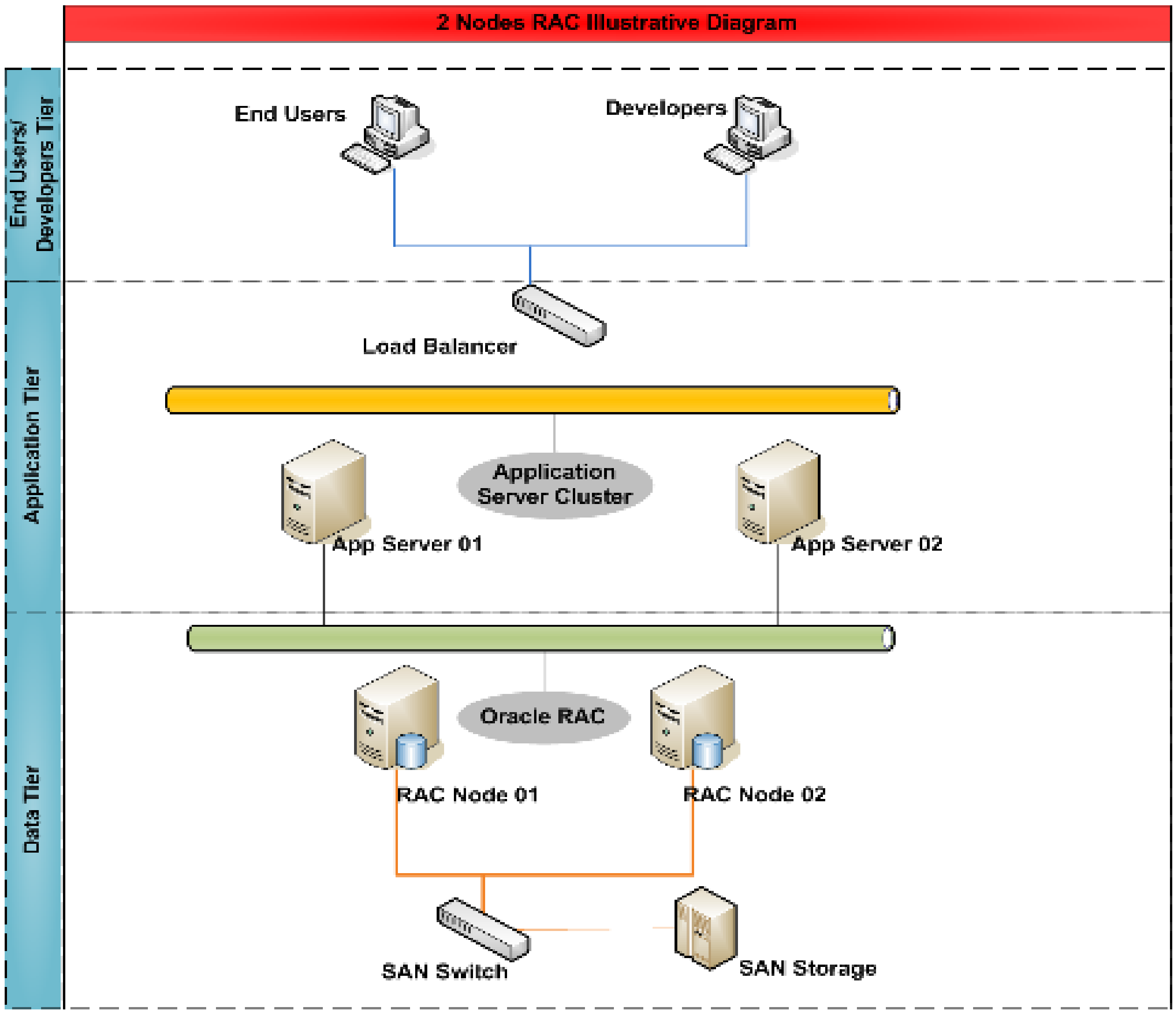
Abbreviation	Description
RAC	Real Application Cluster
AQ	Advanced Queuing
SGA	System Global Area
Oracle XA	Oracle XA is Oracle's implementation of the X/Open Distributed Transaction Processing (DTP) interfaces
FAN	Fast Application Notification
FCF	Fast Connection Failover
UCP	Universal Connection Pooling
API	Application Programming Interface
GCS	Global Cache Service
JDBC	Java Database Connectivity
OCI	Oracle Call Interface
HA	High Availability
FTS	Full Table Scans
TAF	Transparent Application Failover
Cache Fusion	Cache Fusion of Oracle RAC provides the ability to fuse the in-memory data cached physically on separate cluster nodes into a single global cache, where in it guarantees the coordination of data changes on different nodes such that whenever a node queries data it receives the current version, even if another node recently modified that data.

RAC Overview

Here is a super quick over view of RAC to set the context - Oracle Real Application Clusters (RAC) is the feature of the Oracle Database that enables multiple clustered instances of Oracle to simultaneously access a single shared database. Oracle RAC uses Oracle Clusterware for the infrastructure to bind the interconnected servers so they appear as a single system to end users and applications, and a dedicated, high-speed, low-latency, private network known as the cluster interconnect to synchronize activity and share information between instances.

Each RAC instance runs on a separate cluster node, with its own thread of redo and undo tablespace, and maintains its own SGA and system information. All the instances buffer caches are shared: if one instance has an item of data in its cache that is required by another instance, that data is shipped across the interconnect to the requesting node. All data files, control files, redo log files reside in the cluster-aware shared disks and the technical stack provides support for the concurrent access to the shared storage from all the nodes of the cluster.

Following diagram shows the typical deployment architecture of a two Node RAC in an end-to-end application deployment.



Recommendations

Running applications with a backend Oracle database as RAC can benefit the application developers in three major areas; Scalability, High Availability and Functionality. However, in order to fully utilize the benefits of Oracle RAC, one should take few important considerations in each area as listed below.

Scalability

Scalability is one of the typical reasons a deployment chooses to run an application with Oracle RAC as the backend database. Scalability implies that the system is expected to grow in terms of workload demand and adding more resources is expected to keep up with this demand. In this case the resources added are the extra instances of RAC. For an application to scale well with RAC database, the application designer/developer needs to understand the following.

Applications should scale well on single instance for it to scale on RAC

Description:

When the backend database is RAC enabled, it opens up additional computing resources becoming available to take on the database access load. Naturally, this is expected to scale an already performant application. The scaling factor (for example, does the throughput double when one moves from 1 node to 2 nodes?) is dependent on a variety of factors. Lesser the work required of RAC to provide the application a single consistent database irrespective of the number of nodes/instances accessing the same set of physical files (the database), closer is the application to being linearly scalable with RAC.

For existing applications moving to using RAC backend, it is important to ensure that the application does not have problems scaling on single instance database. On single instance DB environment, this would imply addition of resources (more CPUs allocated, larger SGA, etc) to the system does show the overall system being able to handle greater workload in terms of increased throughput and response times. If this is not the case, moving to RAC is unlikely to help. Before moving to RAC, it is important to ensure that scalability aspects are addressed by looking at the database logical and physical design, SQL, application code, etc. and removing any bottle necks.

To ensure that the application further scales on RAC, the DBA needs to ensure that the physical design of the database conforms to best practices for RAC. From the application design perspective, the developer will benefit more than what RAC automatically provides, if the work is inherently partitionable such that individual instances of RAC process transactions without incurring any of the demands on RAC to maintain global cache and locks.

Applications using Advance Queuing should partition load

Description:

While the context in this section is Oracle Advanced Queuing, in general this applies to any user implemented queues or any other object that has access pattern similar to persistent queues. The entry and exit points of a queue, commonly called its tail and head respectively, can be extreme hot spots since either producers or consumers are actively working this part of the queue. To avoid the implications of hot spot several measures can be taken. As usual, before implementing AQ in a RAC environment, the AQ based application should be well tuned to run on a single node. That is, before new nodes are added to the system, all effort should be directed at maximizing throughput on a single node. To ensure the application will also scale will RAC, follow design patterns like

- Design such that it is possible to enqueue and dequeue a message from the same instance
- Consider making application design such that strict serialization of processing is not a requirement i.e. need for queue entries to be processed in exactly the same order as they were received is not truly required.
- If such partitioning is not possible, because Oracle RAC may not scale well in the presence of hot spots, limit usual access to a queue from one instance only. If an instance failure occurs, then messages managed by the failed instance can be processed immediately by one of the surviving instances. Use singleton service feature of Oracle to manage this failover seanless from the client perspective.

Application and Schema Design Considerations

Description:

The notion of hot spot was introduced in the previous section for AQ. This applies in general as well. To maximize the scaling of applications with RAC it is important to minimize scenarios where multiple active sessions, specifically from separate instances are likely to access a set of blocks frequently. Tables with high row density per database block and frequent updates and reads can become "globally hot" with serialization.

- At application design time, factor this consideration in the tables design and database access layers. Where feasible, simultaneous access requirements to tables and indexes (and other objects) and more specifically same blocks of these objects should be localized to individual RAC instances.
- To improve the overall throughput of the system it may be worth isolating such specific access pattern to a single instance at a time while rest of the application uses all instances of RAC. Using a specific RAC service for such work within the application and configure this service as a singleton service will help achieve this.
- Decrease row density - Decreasing row density in database blocks results in reducing the chances of collision for the hot spots and thus helps improve system. While space usage may go up, this makes particular sense for hot tables which are small in size since the performance gain is well worth the extra space required. This can be achieved in a number of ways such as
  - ✓ Higher PCTFREE for table reduces # of rows per block
  - ✓ Including columns in table design by the applications with no real semantics but which will help decrease row density by taking up more space per table row.
- Use Hash range partitioning where feasible
- Design Indexes smartly – To avoid contention for index blocks (and the resulting hot block effect) , Index values should be such that they are well spread out. With indexes on normal business data , this happens naturally in most cases. In some cases, index insertions happen only at the right edge of the index. This situation occurs when applications use artifacts such as sequences (or monotonically increasing numbers) or some such similar construct. In such situations, the right edge of the index becomes a hotspot because of contention for index pages, buffers, latches for update, and additional index maintenance activity, which results in performance degradation.

  - ✓ Avoid using generated numbers for columns that will be indexed
  - ✓ It is preferable to use randomly generated number instead of monotonically increasing numbers.
  - ✓ If using monotonically increasing numbers, use reverse key indexes. Here Oracle physically stores the index entries with bytes reversed (123451 indexed as 154321 and 123452 as 254321).These

implicitly spread out the indexes and reduce index block contention. Revers key also implies SQL select that require range scan (a > 2 and a < 10) is not done using these indexes

- ✓ With hash partitioned global indexes index entries are hashed to different partitions based on partitioning key and the number of partitions. This spreads out contention over number of defined partitions, resulting in increased throughput. A global hash partitioning index can be created on a regular non-partitioned table as well. Details can be found from the references mentioned below.
- ✓ If using Partioning feature of Oracle , use local indexes at the partition level when possible to reduce overhead during index maintenance

- Avoid Full table Scans (FTS) in the application and optimize index based access – This also results in lowering the need for blocks to be accessed by multiple instances. When a query does full table scan it eventually hits all the blocks of the table and this also implies currently "active" blocks being accessed in other instances of RAC need to be accessed by the instance running the query. Going back to the idea of optimizing all simultaneous access to blocks to be local to an instance when possible, FTS should be avoided to minimize this. With index based access , most queries are likely to be satisfied without running into possibilities of requiring to use the cache coherency features of RAC.

As a final note, it is important to keep in mind that most applications can scale fairly well with RAC and do not require one to go overboard with partitioning. RAC features ensures that a lot of the scalability is automatic and further effort in this direction is mainly to exploit other Oracle features with sound application design practices to further maximize the benefit.

Importance:

High

References:

Appendix A.1

Test Applications for Scalability

Description:

Before moving the applications to run on RAC in production, it is very important to test those applications on RAC for scalability and this certification should be done on production like system using similar data and a representative load. This will give the developer and DBA the an earlier and less expensive opportunity to identify the issues that can be easily be fixed either from a database configuration stand point or thru simple application changes. Extrapolation from a smaller test cases or test bed that do not simulate the exact pattern of access may not give an accurate enough picture to apply production scenarios

Importance**:**

High

Considerations for XA Applications

Description:

If the application is using XA transactions in a RAC environment where two or more branches of the same global transaction are accessing the same RAC database and in this case it is desirable for them to be on the same RAC instance. With 11.2 Oracle has introduced several optimizations to help cases where the two branches are on separate instances of RAC but it is still desirable from performance perspective to direct these to be using the same RAC instance. To achieve this use Distributed Transaction Processing (DTP) services.

When using XA, also consider using Connection pools that provide RAC XA affinity and automatically ensure the above when the two branches originate from the same pool. For example, with Oracle Web logic, both Multi Pool Data Source and Grid Link work well with RAC. These XA affinity optimizations do not require use of Distributed Transaction Processing (DTP) services.

Importance:

High

References:

Appendix A.2

Automatic Space Segment Management (ASSM) for RAC Database

Description:

ASSM removes the need to alter the number of FREELISTS and FREELIST GROUPS when new instances are brought online thereby saving the downtime associated with such table reorganizations.

Importance:

Medium

References:

Appendix A.3

Considerations while usage of Oracle Sequences in a RAC database.

Description:

If application uses Sequences, there are several things for considerations. Regarding usage of sequences for indexes, please refer to the section above on indexes. While RAC has multiple instances, the sequence object still is for the database as whole. To ensure performance and scalability for the sequenced object, it becomes desirable for each instance to cache a certain range of sequence numbers. This implies that there is always a possibility that certain sequence numbers will never be used (example a RAC instance crashed or was shutdown and the sequence numbers it cached are now gone forever).

The other aspect of this is that since multiple instances cache a range of sequence numbers from a given sequence object, there is no guarantee that usage of the sequences by the application (as part of sequence.nextval calls) will be ordered. Some instance may be ahead of another and depending on which instance a thread of the application was connected to, the value it received will not be the next highest value. Given this, it is important to design applications so they are resilient to lost sequences and not-in-order sequences. For performance, Create sequences with large instance specific cache. Avoid dependency on sequence number and be comfortable with lost sequences

Importance:

High

References:

Appendix A.4

High Availability

The second big reason for using RAC is High Availability. Since there are multiple instances on multiple machines in the cluster offering a given database service, the deployment is not automatically resilient to a bunch of planned and unplanned outages. A given database service is thus likely to be up in some part of the cluster. Oracle RAC does not protect in flight transaction on a given RAC instance when an instance is lost but will ensure service availability either by allowing/ensuring the service is already running on some other instance or through the service failing over to a surviving RAC instance. Given this, the application can now assume that the database is always accessible and instance/service outages are transient. In some cases, it may take a few seconds for the service to resume or it may already be available. Applications can take advantage of this service availability to provide a seamless failover behavior and mask database outages completely (or at least handle it gracefully) when deployed against a RAC database. To do so, some things to keep in mind are as follows.

Leverage service features of RAC

Description:

This recommendation is not specific to Oracle RAC or it High Availability aspect. All applications should use wire to the database using specific service names and a given deployment should be setup to use services as well. All advanced features of RAC are based on *services*. The key recommendations are

- DBAs should create specific service on the RAC backend for a given application
- Externalize the configuraion used to wire an Application to the database. This also allows it tobe customized .
- Database connection string or URLs should always use *service*. Please check the reference for the recommended format.
  - ✓ As a special mention, never use SID in JDBC connection string for the java applications.

Importance:

Medium

References:

Appendix B.1

Use Standard Oracle Pools

Description**:**

To help application take advantage of RAC availability features, RAC has built in features like FAN (Fast application Notification). This allows RAC to send out notifications on outage events. Oracle Client side connection pools are integrated with FAN and together they ensure that applications can take maximum advantage of RAC HA capabilities. These pools provide immunity from TCP timeouts for in flight calls and eagerly clean up dead connection from connections pools to minimize application exposure to a backend database outage. In order to leverage this, it is recommended not to use home grown connection pools but use Oracle provided connection pools. This implies using

UCP (Oracle Universal Connection Pool) for Java (J2SE based applications)
Grid Link Data sources for Weblogic based applications
FAN/OCI OCI pools for OCI connections integrated with TAF.

Importance:

High

References:

Appendix B.2

Understand Connection Time and Run time Load balancing

Description**:**

Though the typical application developer may not need to worry about this explicitly it is good to understand this feature. As we know, a RAC configuration has multiple instances spread over different machines in a cluster accessing the same physical database. When a client (the application in this case) connects to RAC, it ends ups opening connection to all the instances offering the *service* the application connects to. Oracle connect time load balancing ensures that at the time a new connection is requested by the client , Oracle will decided based on a combination of metrics and heuristics , the best instance to direct the client connection to . The physical TCP/IP connection (the Oracle Net connection) is thus established based on the situation at the time of connection. Oracle connection pool (UCP and Grid Link particularly) also have a feature call Runtime Load balancing. This feature further ensures that all backend instances are evenly loaded in terms of work. With these connection pools, the physical pool has a bunch of connections open and available to use at any given point in time. When an application requests a connection, it is normally handed over a pre-established connection from the pool. Oracle pools enabled with the runtime load balancing feature essentially use load metrics information sent as notifications from the RAC instances to determine which among these pre-established connections should be the one handed over to an application thread requesting a connection. This ensures that even at runtime, applications use a connection that is least loaded and hence also achieves balancing workload on the backend database servers. As a logical outcome of this, it makes sense for applications to follow a pattern of using a connection only as long as they need it and not hold on to it. This ensures that at every request for connection, they get directed to a RAC instance that is most likely to provide the most optimal performance.

Build Retries in the Applications

Description:

For applications requiring seamless failover behavior to RAC instance outages, ensure that best practices are followed in the application code. Typically this requires doing a retry for an ongoing transaction. For example for JDBC, this implies catches the SQL_RECOVERABLE_EXCEPTION and retry the ongoing transaction after fetching a new connection from the surviving database instances.

Once the application has a connection, it may retain it for a while and use it for multiple transactions, or it may use the connection for a single transaction before closing/returning it to the pool. In either case, the application will get exceptions when a database outage occurs. When this happens, the application may be about to begin a transaction using the database connection or it may have already begun a transaction and may be in the middle of a transaction. In the former case, it is easier to retry the transaction since the state of the system is most likely the same to make the transaction retry a valid one. When a DB exception is received in the middle of a transaction, the retry decision is more difficult. If a retry were designed for a transaction, the typical application logic in pseudo code is:

```
Do Pre-Transaction Processing (if any)
WHILE number of specified retries > 0
TRY
      Decrement number of specified retries
      Do Main Transaction processing (perhaps branching out to
      a separate method)
      IF transaction completed successfully THEN
            Do Post-Transaction Processing
      ENDIF
CATCH SQLException
      IF connection is still valid and usable THEN
            As exception is not due to a DB outage, process as
            appropriate or return exception back to caller
      ELSE
            Close the connection
            Get a new good connection
            IF good connection received THEN
```

```
                        Retry back to while loop to begin transaction
                  again
                  ELSE
                        Return exception back to caller
                  ENDIF
            ENDIF
      ENDTRY
      ENDWHILE
```

Integrate Applications using FAN API.

Description:

Oracle has published the interfaces to consume the FAN notifications in applications. These can be used by the application to react to RAC outages more proactively. Applications can use these to manage its own connection pools or incorporate more advanced behavior. Consider integrating the applications with FAN using FAN API in case application needs tighter integration in reacting to RAC instance outages.

Functionality

Most applications work seamlessly when deployed to RAC from a functional perspective. However, RAC does bring an element of multiple physical nodes running the database as a whole and if the DB application interacts with the operating system on the database server there are certain things it needs to do to continue to work correctly.

Applications using DBMS_PIPE

Description:

If an application uses the DBMS_PIPE feature, it needs to be ensured that both producer and consumer are connected to same instance. This happens naturally in a single instance database but in the RAC case when there are multiple instances, the application needs to ensure that the two sessions connect to the same RAC instance. Better still, it is recommended the application use a messaging mechanism such as AQ or JMS instead.

Importance:

Medium

References:

Appendix C.1

Applications accessing external files from within the DB

Description:

If application accesses external files from within the database using DB stored procedures, the DB procedure is likely to run from any of the instances. To make it work, ensure that these files are accessible from all instances of the database where the application service runs. This can be achieved by staging these files on a shared storage mounted across all the machines in the cluster and maintain the same mount points. In such case, ensure that shared storage also provides the right level of concurrency control on file access needed by the application

Applications using External Tables feature

Description:

If application is using External Tables feature of the database, the file needs to be accessible on all instances of the RAC where the applications service runs. This is similar to the previous point with external files.

Applications using System Views

Description:

If application reads information from any of the dynamic views (v$ tables), in RAC environment they will need to read the equivalent gv$ table to get a global view .Most applications do not have to do this at all but many scripts to manage the Database need to consider this.

Importance:

Medium

References:

Appendix C.2

Applications using DBMS_JOB

Description:

If application is using DBMS_JOB, consider moving to using the dbms_scheduler feature. If not possible, ensure that the resources required by the job when it starts are available on all instances of the DB (including ones where the service may not run). Use instance affinity feature to bind a job to a specific instance if a job needs to run from a specific instance.

Importance:

Medium

References:

Appendix C.3

Appendix

Sl. No	References
A.1	Oracle Support Article 220970.1 - RAC: Frequently Asked Questions Global Hash Partitioned Indexes Global Partitioned Indexes
A.2	Using Oracle XA with Oracle Real Application Clusters (Oracle RAC)
A.3	Oracle Support Article 62002.1 - Caching Oracle Sequences
A.4	Oracle Support Article 180608.1 - Automatic Space Segment Management in RAC
B.1	Database URLs and Database Specifiers Using Application Tracing Tools Design and Deployment Techniques
B.2	Universal Connection Pool for JDBC Developer's Guide Using GridLink Data Sources OCI Programming Advanced Topics Transparent Application Failover in OCI
C.1	DBMS_PIPE
C.2	About Dynamic Performance Views
C.3	DBMS_JOB DBMS_SCHEDULER

Quick Checklist

Item	Recommendations	Checked
1	**Scalability Checks**	
1.1	Did applications scale well on Oracle Single instance before considering to migrate on to RAC Database?	
1.2	If Applications are using Advanced Queuing, has load partition been considered?	
1.3	Have the applications and schema been designed keeping into consideration of running them on RAC database?	
1.4	Are the applications well tested for scalablity?	
1.5	Considerations for XA Applications?	
1.6	Considerations while using Oracle Sequences on a RAC database?	
1.7	Considered Automatic Space Segment Management for RAC Database?	
2	**High Availability Checks**	
2.1	Leverage service features of RAC by configuring Applications to use services?	
2.2	Using of Standard Oracle Pools?	
2.3	Evaluated Connection Time and Run time Load balancing?	
2.4	Building Retries in the Application?	
2.5	Integrate Applications using FAN API?	
3	**Functionality Considerations Checks**	
3.1	Applications using DBMS_PIPE	
3.2	Applications accessing external files from within the DB	
3.3	Applications using External Tables feature	
3.4	Applications using System Views	
3.5	Applications using DBMS_JOB	

Oracle RAC Database aware Applications – A Developer's Checklist
April 2013
Author: Pradeep Bhat
Contributors: Bob Dunsby, Lingaraj Nayak

Oracle Corporation
World Headquarters
500 Oracle Parkway
Redwood Shores, CA 94065
U.S.A.

Worldwide Inquiries:
Phone: +1.650.506.7000
Fax: +1.650.506.7200

oracle.com