Oracle Streams Advanced Queuing and Real Application Clusters: Scalability and Performance Guidelines

An Oracle White Paper
January, 2008

Maximum Availability Architecture

Oracle Best Practices For High Availability

Oracle Streams AQ and RAC Scalability & Performance Guidelines

INTRODUCTION

The following paper describes the best practices for creating, accessing and managing multiple Oracle Streams Advanced Queuing (Oracle AQ) queues in a Real Application Clusters environment.

Oracle Streams AQ is an advanced messaging and queuing system that operates inside the Oracle Database. Typically, inbound messages will be deposited (or enqueued) by one set of processes and then retrieved (or dequeued) by another set of processes. This message database might be, for example, the hub of an email or a cellular phone SMS (Short Message Service) based application.

Maximizing the throughput of all the messages typically involves optimizing the queuing rates, tuning the Oracle Database, and making the most efficient use of the available system resources. However, even in an optimal configuration, system resources will eventually limit throughput.

When the resource limitations of a single machine have been reached, database throughput may be improved by using Oracle Real Application Clusters (RAC), which provides the ability for separate instances on separate nodes to operate on the same database and coordinate their operations. However, moving the messaging system from one node to multiple nodes provides further opportunity for tuning of the application and the database.

This paper presents both the considerations in how to architect an AQ-based messaging system in a Real Application Clusters environment as well as the best practices in this regard. These best practices are the result of both internal and customer deployments where the emphasis has been on maximizing the throughput of AQ messages in both single node and multiple node environments.

The first part of this paper describes the concepts involved in deploying a multiple-queue solution that takes the greatest advantage of the scalability inherent in a RAC environment. This includes the applications in which the recommendations are most applicable, as well as guidelines for managing multiple independent queues.

The second part of this paper delves into additional performance tuning considerations for both maximizing messaging throughput, and minimizing contention, both in a single node environment as well as a RAC environment.

Finally, the third part of this paper presents a sample set of steps for deploying an example application with the above considerations.

The guidelines presented here are applicable to Oracle Database 10gR1 or higher, unless otherwise noted.

GENERAL AQ TUNING

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. This is a general tuning exercise and will involve:

- AQ Tuning

Managing potential contention of the enqueue/dequeue processes that access the queue and tuning how queue objects are accessed.

- Tuning Application Code and Data Objects

The greatest performance increases will come from tuning the customer application itself. This includes tuning the overall efficiency of the system as well as SQL tuning in the case of database access.

- Instance Tuning

Instance tuning takes the form of tuning Database initialization parameters and making the most efficient use of the available system resources

- System Tuning

System tuning involves discovering bottlenecks and addressing them. This may take the form of resolving memory or CPU limitations or identifying I/O issues. It's a process often done in conjunction with instance tuning.

AQ ON RAC: A MULTIPLE QUEUE SOLUTION

On a well-tuned system, that has nevertheless reached its resource limits, additional system throughput may be obtained by adding more nodes in a RAC configuration.

Under certain conditions, outlined later in this paper, queue access can be reconfigured so as to be distributed across all the available nodes. This can provide a significant increase in performance but will require some additional management considerations.

The multiple queue solution described in this paper should only be implemented if a single queue, shared across the RAC instances, cannot satisfy the throughput requirements.

The following section outlines some additional requirements.

Queue Access Requirements

There are certain types of queues that are more amenable to the queue partitioning solution described herein. The queue to be partitioned should at least meet the following two requirements:

1) All of the clients that are enqueuing or dequeuing to that particular queue or queue partition should be able to perform all their access through one specific RAC instance.
2) Another limitation is that strict global serialization not be absolutely required.

Both of these requirements are described in more detail below.

Access Criteria Requirement

The primary requirement for partitioning queues can be stated succinctly as follows:

Enqueing and Dequeing from a particular Queue should take place within the same DB Instance.

This places some requirements on the type of queue access that can be partitioned in this manner.

If message access is by a particular condition (such as messages with particular message IDs) then access may not be possible since the messages may be on another instance. That is, there should be no selector criteria for dequeing messages.

The guidelines presented here work best for a multiple-producer, multiple-consumer queue where messages can be enqueued to anywhere in the logical queue and dequeued from anywhere in the logical queue.

Global and local Serialization

The process of splitting up one queue into many parallel queues will have an effect on global serialization requirements.

In a partitioned queue, different parts of the original queue are now being accessed independently. Messages processed on different nodes may take more or less time to be processed. This means that messages placed in the larger, logical queue may not be dequeued in the order they were enqueued. The solution outlined here is a best-effort attempt at serialization. Explicitly,

- Early messages will in general get processed before later messages, and
- The time taken for any specific message to travel through the system will not show a great deviation from the system average.

In cases where serialization is a requirement, a group of messages that need to be serialized can still be sent to the same node. This specific set of messages will be dequeued in the order they were enqueued. The message ordering features of Streams AQ will support this functionality for a specific queue.

An example may be helpful here: if the messages are text/sms messages from cell phone users, it may not be critical that any particular message be received before/after another as long as earlier messages do generally get delivered before later messages.

However, it may be necessary to ensure that a particular group of messages are delivered in exact order. An example may be a multi-part message meant for the same receiver. In that case, the application interface should allow a group of messages to be sent to one specific instance and thus one specific physical queue.

The Parallel Queues Concept

As stated in previous sections, the throughput of AQ in a RAC multi-node environment can be increased considerably by partitioning one queue into multiple queues, with each partition assigned to a specific RAC instance. Or, beginning with many highly active queues, distribute the queues among the available RAC instances. In both cases, the objective is the same: to distribute queue access among the available RAC instances.

From hereon we will use the following language to distinguish the larger logical queue from the queue partitions:

Logical Queue - This is the original, larger logical queue that will be divided into smaller queues, all accessed equally. It is assumed that any client needing access to the Logical Queue can go to any of the queue-partitions to post or retrieve messages.

Queue/queue-partitions - One of many queues that together form a larger logical queue. We use the term 'partition' here only in the logical sense. Each queue-partition is bound to a specific instance in a RAC environment.

Queue creation

Since each queue-partition will be assigned to only one RAC instance, the number of queues in the system will be at least equal to the number of RAC instances.

In the simplest case of one logical queue that is being split up among several RAC instances, we can create one queue-partition per instance. For example, on a 3-node RAC we would create three distinct queues and associated queue tables which will be enqueued and dequeued from completely separately. The naming of these tables should be identifiable and distinct but not tied to a RAC instance id since the Instance-Queue association may change.

However, we want to establish affinity between the queues by adding information about the associated instances to the AQ queue tables.

The Queue table and queue affinity

AQ allows specifying a mapping between queues and their associated instance. The dbms_aqadm.create_queue_table command may be used to specify this mapping at queue table creation. dbms_aqadm.alter_queue_table may be used to alter this mapping at any time. Oracle AQ provides a dictionary view dba_queue_tables that can be used to query this mapping. AQ affinity is automatically re-assigned in cases of instance failure.

Each queue has a primary and secondary instance assignment. During normal operations, the queue is accessed via the primary instance. In the event of failover, the queue can be accessed through the secondary instance.

AQ best-practices recommend setting instance affinity for the queue table and restricting access to the queue table to the specified instance. AQ uses the instance affinity to select the instance in which the queue buffer for buffered messages is implemented. Background processing of messages, like propagating messages and implementing timing properties like delay and expiration, run in the primary or failover instance of the queue.

The application may create its own view on top of the queue table dictionary view to monitor the queues specific to the application .The advantage of creating a user-maintained queue table view is that the user application can add additional information that it may need to manage the application queues for itself. The user created view can hold metadata about the queues such as the purpose of the queue.

Later, we'll discuss how the user created view is created and managed.

How Clients Access the Parallel Queues

Here, we present two solutions to the problem of distributing access to queues on RAC which both rely on multiple queues. In each case, a queue is assigned to a specific instance. The solutions below restrict access of specific clients so that requests for a particular queue partition are made through only one associated instance.

The general solution can be summarized as follows:

1. Partition one queue or distribute multiple queues so that the greatest use is made of all RAC instances in the system.
2. Logically bind each queue to a specific instance.
3. Manage all access to each queue so that it occurs through the one instance associated with that queue.

We'll describe each of these steps in more detail. First, we'll expand on two different ways to address #3 above. There are two configurations to choose from when deciding how to configure clients to access the multiple queues.

In Configuration 1: Fixed Client Solution, a queue is accessed by a particular client. Each client accesses the queue through only one specific instance.

In Configuration 2: Server side solution, a queue is accessed by *any* client but is only accessed via a specific instance.

It might seem at first that Configuration 2 has all the advantages. But Configuration 1 allows the client to control the load balancing across the nodes by attaching to a different instance. In Configuration 2, application may take advantage of RAC load balancing as well as distribute the queue tables across instances by specifying instance affinities.

Configuration 1: Fixed Client Solution

In one implementation, clients can be configured to directly access a specific RAC node.

Depending on the type of client this will be in the form of a specific database connect string contained in a JVM configuration, a JMS client or others. Each client also accesses specific queue tables, confining access to the queue table to one RAC instance.

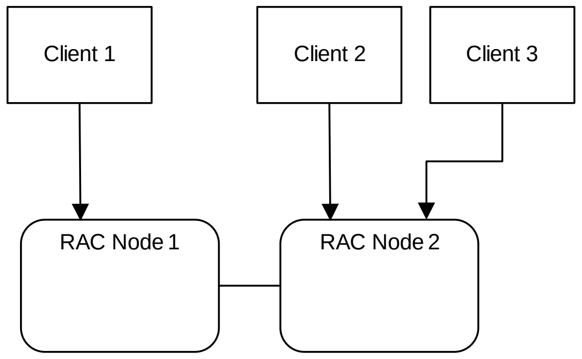


Fig 1. Fixed Client Connections to RAC instances

Each individual client is configured with the fixed location of one node. In the example above this would be:

	Node
Client 1	1
Client 2	2
Client 3	2

In the event of node failure, a client can connect to any other active node. For example, if Node 1 failed above, Client 1 would continue to access its associated queues via Node 2. The client would be responsible for failing over to a new node.

In this configuration, queues are assigned to nodes and also accessed by specific clients. And so a queue assignment (or queue partition) in this example might be:

Queue	Primary Node	Secondary Node
AQ1	1	2
AQ2	2	1
AQ3	2	1

The queue assignments are specified as a queue table property and accessed from the user created view or queue table dictionary view. In the event of a node failover, a queue continues to be accessed via its assignment to a secondary node. The failover node of the queue table would match the client failover node.

More detail on setting up and managing the above configuration, as well as caveats and tuning considerations are in the sections that follow.

Client Connection guidelines

As shown in Figure 1, individual clients should be configured to connect to a specific RAC instance. The exact procedure for doing so will depend on the type of client. Here we briefly discuss the configuration for JMS Servers as well as the process for how a client determines the appropriate queue to enqueue or dequeue from.

OJMS

In many applications, JMS is used as an interface to AQ. In this case, the JMS resource providers should each be configured to connect to specific RAC instances. This can be done for example in the application.xml file with a configuration of the form:

```
<resource-provider name="OJMS" class="oracle.jms.OjmsContext">
   <property name="url" value="jdbc:oracle:thin:jmsserv/jmsserv@node1:1521:aqapp"/>
</resource-provider>
```

or, alternatively, set up the JDBC connection string to access a RAC service with only one preferred instance. The Failover section, later in this paper, discusses how the failover can occur.

Clients of JMS can connect to any JMS Server. The configuration above guarantees that work on a specific queue will occur on the same instance.

Queue table lookup

After a connection to a specific RAC instance, the client must then consult the queue table dictionary view in order to retrieve a list of available queues to work on (either enqueue or dequeue.) Since one or more queues may be assigned to a particular instance, the queue can be chosen randomly among the available queues in order to equally distribute access to all the queues on that instance.

Choosing the specific queue table to work on should precede any enqueue or dequeue work by the client process. That is, the queue table is passed as a parameter to enqueue/dequeue operations that occur within the same database session.

Client Failover

In the event of node failure, connected clients will have to failover to a surviving node. Failover events are considered (hopefully rare) exceptions. The discussion below is limited to JDBC clients although the general principles apply to other clients such as OCI.

The recommendation for client failover is to use RAC's service failover mechanisms in conjunction with Fast Connection Failover for notification to connected clients.

The failover of database services will allow new clients to establish a connection on a surviving node. For clients in the Server-side solution [Configuration 2 below] this will be in the form of an address list. Clients in the client-side solution framework will have one fixed service to which a connection is established

For connected JDBC clients, connection URLs should use service names for the primary host and other hosts in the RAC cluster. If a service is used, only one preferred node should be set. Also, FastConnectionFailover should be enabled and ONS notification to the middle tiers should be configured. This will allow current clients to failover to receive notification of a down node and re-establish connections to a surviving node.

Instructions for doing this configuration are available in the Oracle whitepaper 'Oracle Application Server 10g Fast Connection Failover Configuration Guide' which also contains instructions applicable to generic JDBC clients.

Configuration 2: Server Solution

In addition to the fixed client configuration outlined in Figure 1, there is an alternate solution that allows clients to connect to any RAC instance. The additional requirement is that each client must then "discover" which instance it is on and then select the queues accordingly. The solution is illustrated visually below:

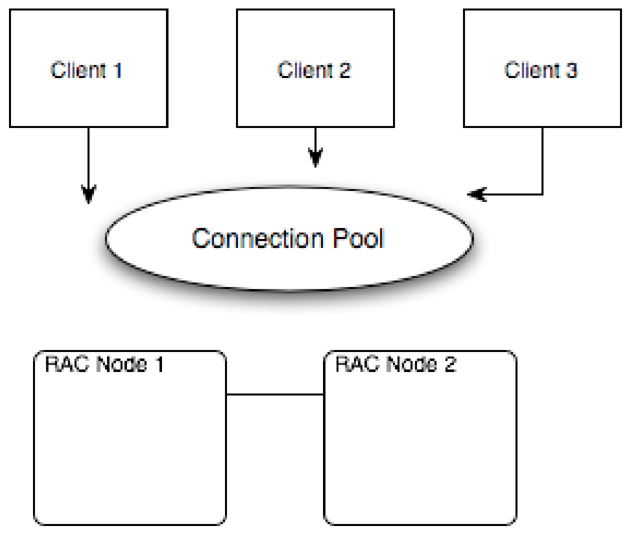


Fig. 2 Server side configuration

Clients may connect to the database in any manner, including via a connection pool or through normal RAC load-balancing mechanisms. The steps a client must undergo after establishing a session are:

1. Discover which instance the session is connected to. (Addressed in the 'Node Discovery' section later in this paper)
2. Determine which queues belong to this instance, using the queue table (An example of this lookup is provided later in this paper)
3. Randomly choose a queue to work on

The additional step required in the Server-side solution is Step 1. Since the client is not connecting to a fixed instance, the instance must be discovered. The step for doing this is explained below.

Node Discovery

At the initiation of the session in the Server-side solution (Fig. 2) the client needs to determine which instance it has connected to. The following is one method of achieving this:

At the beginning of the application session, the client can issue a query to determine the instance:

```
SQL> select sys_context('USERENV','INSTANCE') from dual;
```

This will return the instance number (e.g. '1') that the client has connected to. This can be fed into the subsequent queries outlined above to determine the queue table to work on. This query and its subsequent use must both take place within the context of the same database session.

Managing the queues

Since queues operate independently, the load on the queues must be managed so as to have an even balance of workload across the queues. This involves two separate activities: The first involves methods of monitoring the size and load on the queues. The second is making adjustments so as to re-distribute that load if necessary.

Finally, managing the queues also involves re-distributing the queues in the event of a node failure – either temporary or permanent.

Monitoring queues

Queue sizes can grow, even in the case of a single queue, if the dequeing is not keeping up with the enqueuing. In the case of multiple, parallel queues it's also a good practice to keep the queues approximately the same size. This is to help ensure that the average time a message spends on the global queue is independent of which node it is enqueued to or dequeued from.

Monitoring queue size [Applicable only to Database 11g]

In a Real Application Clusters environment, each instance keeps its own Oracle Streams AQ statistics information in its own System Global Area (SGA), and does not have knowledge of the statistics gathered by other instances. When any GV$ view is queried by an instance, all other instances funnel their Oracle Streams AQ statistics information to the instance issuing the query. So the following dynamic views could be queried to measure the performance of various AQ components across the cluster:

```
V$BUFFERED_QUEUES
V$PERSISTENT_QUEUES
```

The view GV$BUFFERED_QUEUES displays information about all the buffered queues in the database. The view GV$PERSISTENT_QUEUES displays information about all active persistent queues in the database. For example, to discover which queue on a particular instance has the least number of messages, one can issue the query:

```
Select queue_name from v$persistent_queues where enqueued_msgs = (select min(enqueued_msgs)
from v$persistent_queues);
```

The above query will return the queue name where the minimum number of messages have been enqueued since the instance startup time. One thing to note here is that the performance views are an estimate and don't reflect the exact number of messages that are in the queue on disk. Still, these views should be helpful in monitoring the load across the queues in the system.

Managing queue load

The best way to avoid imbalanced queues in the first place is to construct the system so as to have balanced traffic across all the queues. What follows are some general load-balancing guidelines for working with distributed queues:

In the case of the fixed client solution (Configuration 1), the number of clients attaching to each instance should be kept the same. This assumes that all other things are equal: that is, each Client is producing the same amount of work and the servers hosting the instances are equal in terms of system resources.

In addition, load balancing can occur to the clients themselves. In the case of fixed JMS clients, for example, each client can be configured to connect to a specific RAC instance. Then, load balancing the traffic can take place by balancing the connection to each of the JMS Servers. This architecture is illustrated below:

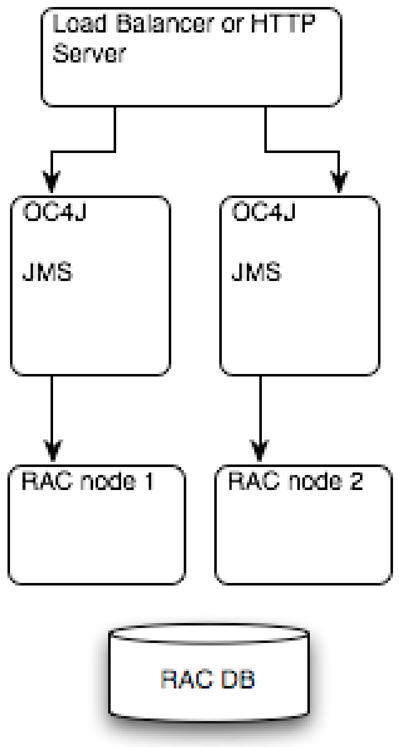


Fig 3. Load Balancing across the Clients

That is, traffic can be managed at the front-end if necessary depending on the load profile at the back-end of the system.

In the Server side solution (Configuration 2) any client can connect to any instance. In that case load balancing can be managed via RAC load balancing schemes.

External clients

Since load across the system is carefully managed by the process of queue assignments and fixed client connections, care should be taken when considering other external clients that access the system.

Any type of access, especially among clients that access the AQ queues directly, should be managed carefully:

> *Any external clients must use the queue table dictionary view to determine which queue to work on.*

The assignment between instances and queues is critical.

Any external clients should, after accessing a specific RAC instance, use the dba_queue_tables view to retrieve the list of available queues.

Queue failover

In the event of instance failover, the queue table needs to be rebalanced. This requires:

- Re-assigning queues

All queues previously assigned to the failed instance now need to be assigned to the secondary instance. The previous secondary instance becomes the new primary instance. The AQ background process will automatically re-balance the queues based on the chosen primary and secondary instances.

- Failing over client connections if necessary

In the case of the fixed-client solution, clients will have to failover to a surviving instance. If the connections are to RAC services then this failover should occur automatically. The failover instance of the client should match the failover instance of the queue.

For current connections, Fast Connection Failover (FCF) should be configured to failover existing sessions as described previously.

Also note that the above is only applicable to persistent queues. Buffered queues are not recommended in these configurations.

Queue table recreation

In the case of long-term or permanent instance reconfiguration the application queue table should be reconstructed to reflect new, balanced mappings of instances and queue tables

PERFORMANCE TUNING

In these sections we discuss some performance tuning considerations that are specifically applicable in this environment.

Tuning Queue access

If the goal is to process messages on a queue as fast as possible, then multiple threads may be necessary to both enqueue and dequeue in parallel. Tuning the degree of parallelism is appropriate in both a single node and RAC environment.

Also, it will depend on a variety of factors and so is very environment-dependent. A tuning exercise will involve increasing the amount of parallelism on the system until either system resources are exhausted or the increasing contention between the parallel processes overwhelms any other gain.

Tuning queue access

Since the queue described in this document is a FIFO queue, dequeuers will contend with each other in their attempts to grab the next message off the queue.

A typical tuning exercise will involve increasing the number of dequeuing processes while monitoring the system throughput. Also system statistics will show a dramatic rise in buffer_busy_waits when too many processes are contending for the same blocks.

The number of both enqueuers and dequeuers should be managed in order to minimize contention on individual blocks.

Next Message queue navigation

One navigation mode in AQ is DBMS_AQ.FIRST_MESSAGE. This will instruct processes to grab the first message off the queue. Parallelization can be increased however (and potential contention minimized) by having dequeue processes operate on the second message in the queue.

To enable this, the navigation mode should be DBMS_AQ.NEXT_MESSAGE. This will be the default in PL/SQL and OCI applications.

Ensuring that NEXT_MESSAGE is enabled may provide a marginal increase in performance, especially in environments where contention is high. However, doing so has the drawback that serialization is further compromised since the first message on the queue may not be the first one dequeued. However, this may be acceptable especially in an environment where strict serialization is not a requirement on any specific queue.

Note: JMS clients will automatically implement NEXT_MESSAGE within a transaction but switch to FIRST_MESSAGE after a commit.

Tuning Data structures

Certain aspects of how data objects are created, stored and managed may provide additional opportunities for performance improvements.

ASSM Tablespaces

Because of contention for the same blocks across RAC instances, it is usually recommended to use freelists and freelist groups for segment space management.

Using Automated Segment Space Management (ASSM) separates the data structures associated with the free space management of a table into disjoint sets that are available for individual instances. With ASSM, the performance issues among processes working on different instances are reduced because data blocks with sufficient free space for inserts are managed separately for each instance

We strongly recommend the use of ASSM tablespaces when creating not only data objects described in this document but in almost any RAC environment.

Sequence Caches

A well-sized sequence cache is particularly important here in an environment where queue messages may be tied to an internally generated sequence number.

A general recommendation is creating sequence caches of at least 5,000 to 10,000 for each instance. Higher values may provide more benefit at the cost of increased memory requirements.

Tools such as Oracle's Automatic Database Diagnostic Monitor (ADDM) can be used to obtain more specific recommendations on appropriate sequence cache sizes.

Sequence caches also come with application prerequisites. For example, strict sequenced message ordering should not be required.

Tuning RAC Instances

There are specific RAC instance parameters that are relevant to a partitioned queue environment. We discuss some of these here.

Gcs_server_processes

This can be set to 1, at least initially. One Lock Manager Service (LMS) should be sufficient for most purposes. If the LMS process is consuming too many resources, then an additional one can be started.

Gcdefer_time

This parameter sets the wait time from when a block is requested and when it is released to another node.

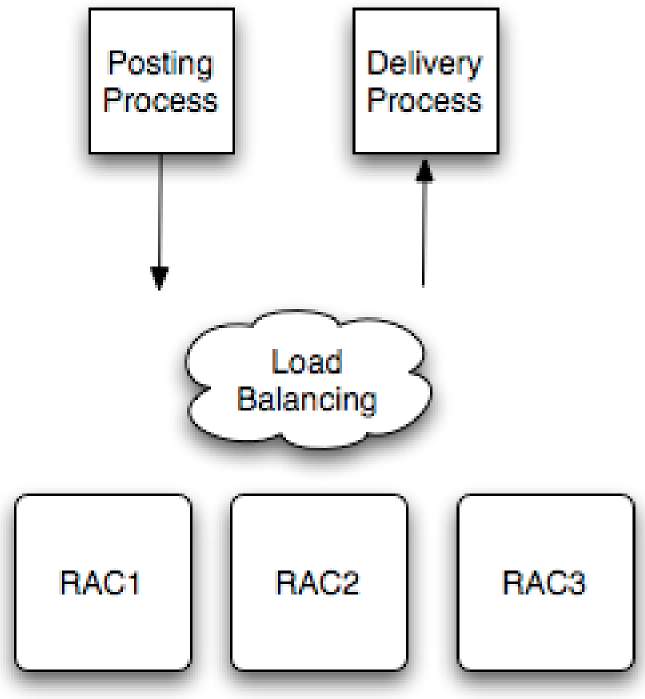
If an active block, such as that associated with a queue, is requested on another node, either the application has not been fully well-partitioned or a transfer is taking place – that is, a new instance is becoming the owner of a queue. In the latter case, it is best to set this parameter to 0 so that the transfer can take place quickly and efficiently.

IMPLEMENTING THE SOLUTION

In this section, we specifically walk through an example application.

In this case, we have a messaging system that receives requests and places them in the queue. Sending processes then pick up messages off the queue to deliver them to their destination.

We'll use the Server-based solution for managing client access. That is, a client can connect to any instance and will then post or retrieve to the queues associated with that instance.



The steps involve creating the data objects, modifying the application and then tuning and monitoring the application.

Create the new data objects

We are assuming that we have one Logical Queue where all messages are posted and then retrieved. Since we have three RAC nodes, for load-balancing reasons we will want to split this one queue up into 3N distinct queues. For this example, we will split it up into 3 queues.

Step 1: Replace the one logical queue with three new queues

For example, on a 3-node RAC, we would create three new queues, one per node:

```
dbms_aqadm.create_queue_table(
     queue1_table, payload, …);
dbms_aqadm.create_queue(queue1, queue1_table,…);

dbms_aqadm.create_queue_table(
     queue2, payload, …);
dbms_aqadm.create_queue(queue2, queue2_table,…);


dbms_aqadm.create_queue_table(
     queue3, payload, …);
dbms_aqadm.create_queue(queue3, queue3_table,…);
```

and then specify primary and secondary instances for each of the queue tables:

```
dbms_aqadm.alter_queue_table( queue_table => queue1,
       primary_instance => '1',
       secondary_instance => '2');
dbms_aqadm.alter_queue_table( queue_table => queue2,
       primary_instance => '2',
       secondary_instance => '3');
dbms_aqadm.alter_queue_table( queue_table => queue1,
       primary_instance => '3',
       secondary_instance => '1');
```

Step 2: Create user-managed queue table view

A simple user-maintained queue table view might include the following columns:

```
queue_id NUMBER PRIMARY KEY,
queue_name VARCHAR2(24) NOT NULL,
pri_instance_id varchar2(4), /* the primary instance id */
sec_instance_id varchar2(4), /* the secondary instance id */
active_instance_id varchar2(4) /* the active instance id */
```

This maps each of the queues to both the primary instance through which it will be accessed as well as a failover instance. In all cases, access will be through the primary instance. If the primary instance becomes unavailable then the failover process will involve setting the new primary instance to be what was previously the secondary instance.

The views dba_queue_tables, all_queue_tables, user_queue_tables contain the following columns of interest:

Owner – schema of the queue table

Queue_table – the name of the queue table

Primary_instance – the primary instance specified during create_queue_table or alter_queue_table

Secondary_instance - the secondary instance specified during create_queue_table or alter_queue_table

Owner_instance - The instance to which the queue table is currently mapped. If the primary instance fails, the secondary instance automatically becomes the owner. If the primary instance comes back up, it will regain ownership

The mapping between queues and queue tables is defined in the views dba_queues, all_queues, user_queues. These views contain the *owner* of the queue table, the *nam*e of the queue table and the associated *queue_table* name.

Additional columns can be added to this table in order to provide more information about the queues being used by this application.

Modify Application Access

With the new queue assignment, we now distribute client load across the RAC nodes. This can be done in two ways, as summarized previously. The first is to have fixed connections between specific clients and specific RAC instances. In that case, each client also works on a specific set of queues – those that are bound to that particular instance.

Here, we walk through the other case, where a client can connect to any instance and then retrieve a list of available queues, depending on which instance it has connected to.

Step 3: Application made instance-aware

Before selecting a queue to work on, the application must first determine which instance it has connected to. This can be done as previously stated with a query such as the following:

```
select sys_context('USERENV','INSTANCE') from dual;
```

which will retrieve the instance number that the current session has connected to.

Step 4: Look up available queues

Using the instance number obtained from the previous query, now do a lookup of the queue table to get a random queue to work on.

This can be done with an example SQL statement of the form:

```
SELECT queue_name
    FROM queue_table_view
    WHERE
    active_instance_id = instanceid
    ORDER BY dbms_random.value)
   WHERE ROWNUM = 1;
```

In the above query, the queue_table_view is the view we created above. This randomly selected queue is then used for subsequent operations such as enqueuing or dequeuing:

```
Dbms_aq.enqueue(queue_name, op, mp, data, gid);
```

Tune the System

The system should be monitored to determine the greatest source of waits. This can be done with system performance tools such as ADDM or TKPROF.

In addition, the recommendations in the Performance Tuning section of this document should be implemented where appropriate.

Monitor the Application

The guidelines, posted previously, for monitoring the load and size of queues across the system help ensure that work is distributed fairly equally and that the time it takes a message to be processed through the system is fairly independent of which queue or instance it was posted or retrieved from.

CONCLUSION

Oracle Real Application Clusters is a powerful solution that leverages the processing power of multiple nodes to provide access to one physical database. Oracle Streams AQ provides the ability to use the database to queue message traffic between multiple producers and consumers.

The guidelines in this paper provide one form in which the parallel processing power of RAC can best be leveraged to provide fast and scalable access to a logical queue which would ordinarily be shared by all nodes.

Oracle Streams AQ and RAC: Performance and Scalability Guidelines
December 2007
Author: Richard Delval
Contributing Authors: Neerja Bhatt, Michael Zoll, Anil Madan, Pradeep Bhat

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