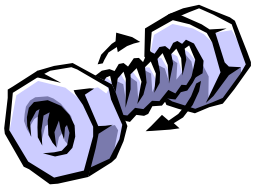
RAC PERFORMANCE TUNING
Part 1: Is RAC Like *Bolt-on Power*?



"BOLT-ON" POWER?

It's tempting to think that "bolting-on" RAC automatically improves scalability and alleviates performance problems. Alas, that simply isn't the case. Just like many Oracle features, you need to understand the strengths and limitations of the particular technology so that you can optimize the advantages and avoid potential pitfalls. RAC can indeed provide better scalability—as long as you address the key issues that may obstruct these gains. If you ignore the limitations, you could get *worse* performance after migrating to RAC.

CAPITALIZE ON RAC ADVANTAGES

Let's take a look and see why RAC can lead to better performance (or at least help maintain existing performance.)

RAC Expands CPU Potential

A key point to keep in mind is this: Installing RAC does not cause things to run faster; rather, RAC provides the *potential* for faster performance by allowing potentially more CPUs to be employed. For OLTP applications, this simply means having more CPUs available. For batch jobs, data warehouses, and large reporting systems, this likely means increasing the number of threads (or alternatively, increasing Oracle parallelism, which is not the same thing.)

OLTP

Let's first consider OLTP applications, disk reads are often minimal—that is, the application is *cpu-limited.* The SQL is usually very efficient, with each query accessing data via a very selective index. Many blocks are cached at either the Oracle or SAN level. Thus, it is very common for simple OLTP queries to average less than a single disk read per execution.

For example, I analyzed the most-run queries on a very large OLTP application recently. The most frequent query consumed just 1 disk read every *3,000 executions*. Clearly, this application is cpu-bound. Using RAC and adding more cpus will indeed increase the *capacity* of the system. Of course, individual queries will not actually run faster (unless the new nodes have faster CPU speeds.)

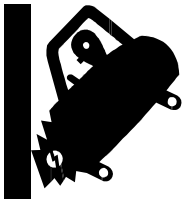
RAC PERFORMANCE TUNING
Part 1: Is RAC Like *Bolt-on Power*?

Batch Jobs

Batch jobs are trickier to analyze, but will likely also benefit from RAC. Most batch jobs tend to be disk i/o heavy, or *i/o limited*. For example, in a major financial system application I analyzed recently, 2/3 of the elapsed time was spent waiting for disk i/o. At first glance, it may appear that RAC would be of little value, since RAC doesn't directly change the disk system. After all, if the batch job is not waiting for CPU cycles in the first place, how can adding more CPUs help?

Nevertheless, expanding the CPU capacity can improve these batch jobs as well. Here's why: Overall disk throughput for a given job is directly tied to the number of threads (or degrees of parallelism) that are causing the disk retrievals[1]. So, getting blocks off disk at a faster rate likely means adding more threads—*even though little CPU will actually be consumed from the new threads!*

An interesting corollary of this is that the increased performance for these disk i/o limited jobs may be achievable *even without RAC*. We could simply add the extra threads, knowing that most of the new resources consumed will be from the disk system, not CPU cycles.



BE AWARE OF POSSIBLE ROADBLOCKS

Although RAC offers significant advantages, distributing processing across multiple nodes does not happen without a cost. Let's take a look at that cost. Later, we'll also explain good ways to avoid performance pitfalls.

Queries Of Objects Spanning Nodes May Degrade A *Little*

The nature of RAC is multi-node, with each node contributing cpu power. Various sessions, whether OLTP, reports, or batch jobs can potentially run on any node. That's the beauty of RAC. However, this strength of RAC also contains the seed for performance degradation. Here's why: Every time a session accesses an Oracle block, RAC must ensure that the session is provided the *correct version* of the block—which may actually reside on a different node. When this occurs, there is a slight, *but not negligible* delay, as the block is transferred via the cache fusion technology. Additionally, if the sending node has modified the block, but not yet written the transaction to the redo log, there is a further delay whilst the redo block is written.

[1] I am assuming, of course, that we have not reached the overall throughput of the disk system.

RAC PERFORMANCE TUNING
Part 1: Is RAC Like *Bolt-on Power*?

Undo Considerations

RAC adds additional complication when nodes are variously updating, inserting, or querying the same block. A typical scenario would be programs that insert rapidly, but perform infrequent commits. Any node doing a query of these active blocks must undo the un-committed changes—wherever they originated from, local or remote. Thus, the requesting node must request, wait for, combine, and apply undo changes from multiple nodes. Shipping of undo blocks can thus potentially add significant delays.

Full Table Scans Issue

Delays due to full table scans can be exacerbated in RAC. When a block is not found cached in the local node, Oracle first attempts to find the block in a remote node (before resorting to a disk read.) Potentially, these extra calls lead to longer delays.

Updates Of Objects Spanning Nodes May Degrade A *Lot*

There are special performance concerns when performing large inserts. In a recent RAC study I analyzed, this was the #1 performance drag. When multiple nodes are performing massive inserts, there may be lots of high-water mark adjustments. Before a node can adjust the high water mark, however, it must acquire the HWM enqueue. This leads to delays if a node must wait for the enqueue. Of course, avoiding small extent sizes is a good solution to this issue.

Wait Events

Oracle captures RAC-unique delays in a set of wait events. The wait event names can be bewildering, but they all identify different delays in transmitting the requested block to the requesting node. Here are some typical RAC wait events that you will almost certainly see in your RAC implementation. (There are others, but these are typical):

- gc current block busy:
- gc current grant busy:
- gc current block 2-way:
- gc buffer busy:
- gc cr multi block request.
- gc current grant 2-way.
- gc cr block busy.

We'll look more closely at these wait events in subsequent papers.

RAC PERFORMANCE TUNING
Part 1: Is RAC Like *Bolt-on Power*?

GOOD PRACTICES

We have a variety of options that we can use to moderate this delay. To head-off possible performance degradation, there are a few key steps.

Optimize Single-Node Performance First

If you have performance problems on a single node, you will experience even *worse* performance problems with RAC. Using RAC can't possibly fix a performance problem—at best, it may get only slightly worse. Here's one of the lessons-learned during a recent Siebel-RAC implementation study:

> "RAC is not a silver bullet for your performance/scalability problem. An application must be tuned to perform and scale on a single-node database before scaling on RAC."[2]

Minimize Interaction Between Nodes

Many RAC performance issues relate to minimizing delays due to shipping blocks back and forth between nodes—or worse, some form of inter-node block contention, such as happens during massive inserts occurring simultaneously.

In an ideal setup, we would like the blocks used by one node to rarely be needed by another. This means that we somehow partition users or data to minimize areas of overlap. While 100% node decoupling is probably not achievable, you may be able to partially accomplish this.

One simply way is to partition oft-used tables based on geographical regions. Then, route users to a node based on their region, so that any given partition is only accessed by a single node. This could possibly be as simple as setting up *East-West* partitions.

Minimize Block Contention

You will not be able to completely prevent all block contention, but there are some effective tactics to minimize it.

---

[2] OpenWorld 2007 presentation S290535, *Siebel on RAC: Customer Experience*, presented by James Qiu.

RAC PERFORMANCE TUNING

Part 1: Is RAC Like *Bolt-on Power*?

Multi-Thread Batch Jobs Based On Block

For multi-threaded batch jobs, it's possible to organize the work by block#, so that all of the data in any given block is always accessed by the *same thread*. (This is actually a good idea even if you're *not* using RAC.) In this way, there will be little shipping of blocks back and forth, since no two threads will ever need the same block. Therefore, you reduce buffer busy waits, which in RAC will likely turn into *gc* buffer busy waits.

Use Hash-Partitioning to Moderate Contention

Oftentimes, data access tends to clump around certain hot blocks, which contain oft-referenced data (possibly the most recent data.) This can cause significant performance problems with RAC, as different nodes compete for the same blocks. Hash-partitioning is a good way to distribute the data among different partitions—that is, we *uncorrelate* the data. Now, all of the partitions become equally hot, as the data is distributed roughly evenly across the partitions. This partitioning can be considered for both tables and indexes that experience contention amongst nodes.

Use Large Cache Size For Sequences. This applies to cases where a sequence is used to generate a key, and the application performs rapid inserts using that key. The issue is with the index related to the sequence. Because of the closely-related key values, each node doing the insert will likely be competing for the same index block. By greatly increasing the cache, however, each node will be working with drastically different key values, and thus different index blocks.

Use Reverse Key Indexes. This idea is similar in effect to the previous. The point is, we need to avoid inserting nodes competing for the same index block. That is, we need to spread the values somehow into different index blocks. Instead of changing the sequence numbers, however, we store the values in the index *reversed*. So, even though the keys are *lexically* adjacent, they are not adjacent when stored in the reverse index.

One caution: a reverse index will not support a range scan, because the values are no longer stored next to each other. On the other hand, these types of queries should be very rare. Why would we be doing queries based on a range of artificially-generated keys?

FOCUS ON LIKELY PROBLEM AREAS

RAC PERFORMANCE TUNING

Part 1: Is RAC Like *Bolt-on Power*?

It is important to note that RAC-induced contention will likely correlate with contention that *already occurs.* That is, you can probably project some of the problem areas ahead of time.

For example, a recent analysis showed a particular insert statement, in a non-RAC production environment, with substantial *buffer busy* waits. With RAC, this translated into a *global buffer busy* wait. So RAC simply amplified contention that was *already occurring*. A local buffer busy wait turned into a *gc* buffer busy wait.

For a detailed explanation of using statspack and AWR reports to predict potential RAC problem areas, see my white paper, Using AWR Reports for RAC Performance Analysis.

Chris Lawson is an Oracle Performance Consultant and Oracle *Ace* who lives with his family in Dublin, California. He is the author of *Snappy Interviews: 100 Questions to Ask Oracle DBAs,* and *The Art & Science of Oracle Performance Tuning*, both available on Amazon. Chris' web site is: www.OracleMagician.com. In his spare time, Chris enjoys golf, choral singing (bass), and throwing frisbees to Morgan, the resident border collie.