PERFORMANCE TIPS FOR BATCH JOBS

Here is a list of effective ways to improve performance of batch jobs.

Use Set Processing

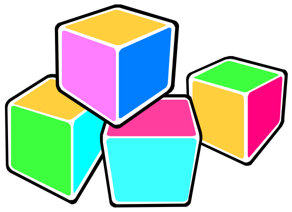
This is probably the most common performance lapse I see. The point is to avoid looping through millions of rows, issuing SQL calls as you go. Instead, try to process all the rows *as a group*. For example, looping 1 million times, running one SQL statement each time, is much slower than issuing a single SQL statement that retrieves all 1 million rows.

There are several reasons why set processing is so much faster. First of all, you can take advantage of multi-block reads (discussed next). Multi-block processing can literally give you a 100x speedup. In contrast, performing a huge number of single-block reads is rarely a good choice. The multi-block method is especially attractive if your tables are partitioned such that Oracle only scans a few partitions.

Set processing also avoids the degradation due to issuing thousands (or even millions) of separate SQL statements. This degradation is not really eliminated by the use of bind variables. Even if you use bind variables, Oracle must still process each statement, and send back the result set. In extreme cases, the time to send the SQL statement repeatedly over the network actually becomes a bottleneck.

SQL scripts, rather than a procedural language, are oftentimes the better approach. Of course, with SQL scripts, you are pretty much forced to handle the data as a group. Set processing means you may need staging tables to hold intermediate results. This is common practice.

Take Advantage of Multi-Block Reads



On most servers, Oracle can read up to 1 Mb (typically 64-128 blocks) at one time. That is why a full table scan can be performed so quickly. Keep in mind, however, that 2 conditions must be met:

1) The database parameter ***Db_File_Multiblock_Read_Count*** must be set correctly; and
2) The table or index being scanned must have extent sizes of at least 1Mb.

If the multiblock parameter is set too low at the database level, you can easily alter your session to set the parameter higher. The second point above recognizes that Oracle will not continue a multi-block scan across extent boundaries. If most of the extents are great than 1 megabyte, it's probably okay. (This is one of those cases where extent sizing really does matter.)

It is especially important to optimize your reads if you are using Oracle parallelism. Why launch many extra processes if you don't first optimize what a *single* process can do?

Optimize Queries *before* using Oracle Parallelism

As long as there is spare capacity on your server, you can usually improve your full table scans by using the PARALLEL hint. But first, make sure the query runs as efficiently as possible. If the query runs badly with one process, it won't be much better with six or eight--plus, you will be consuming server resources unnecessarily.

As a general rule of thumb, a parallel degree of 6 typically gives excellent performance. Of course, this assumes that you have adequate reserves (both CPU and disk throughput) on your server.

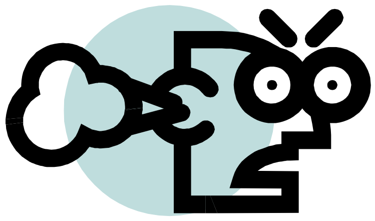
4) Avoid Massive Deletes

Oracle is simply not very fast at deleting millions of rows. Instead, copy and temporarily store the rows you DO want, truncate the table, and then put the rows back in. This can easily be 10x faster. If this method is not feasible due to business considerations, consider multi-threading (discussed later.)

5) Use Summary Tables and Materialized Views

If you repeatedly access a large table, consider building an aggregate table, or materialized view. A materialized view used in this manner is just like a table, but you can rebuild it with a single command. Remember to use Oracle parallelism to speed-up the refresh.

Whether you use a materialized view, or actual table, the idea is to create a "pre-digested" form of the data. You include only the columns and rows that meet your conditions. Once you build the summary table, you can also build custom indexes that optimize your query.



Avoid Excessive Commits

A commit after every row will usually wreck performance. Most jobs should commit no sooner than every 1,000 rows. Committing every 10,000 rows would be a good rule of thumb.

Unless you are working with many millions of rows, further increasing the commit interval doesn't really offer much improvement.

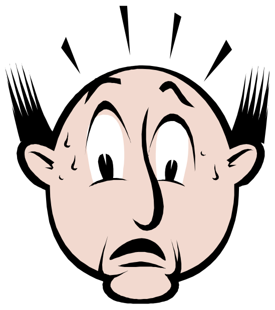
7) Don't Forget to Analyze New Tables

If you build and populate a new table, don't forget to gather statistics. It's very common to see a performance bottleneck caused by incomplete or inaccurate table statistics. It's not necessary to sample all the rows when gathering statistics; the "estimate" option is usually fine.

8) Turn off Logging When Feasible

Oracle allows you to turn off transaction logging for a handful of cases: creating or rebuilding indexes, inserting rows, and Creating Table As Select. To do this, run Alter Table [name] Nologging, then use the hint ***Nologging***. Remember that deactivating logging means the data cannot be recovered after a disk crash and database recovery. So this is not appropriate in every date.

Speed Up Inserts



In cases where you don't really need transaction logging, you can speed up inserts a bit by using the ***Nologging*** feature. For inserting rows, set the table NOLOGGING (using Alter Table …) , then use this syntax: INSERT /*+APPEND */ .

Remember, however, that all the new rows will be placed at the end of the table--above the "high water" mark. This means that if you are performing deletes from the table, these "holes" will never be filled. The table will grow rapidly. Eventually, you will need to rebuild the table to crunch it down.

10) Avoid Building Flat Files

When transferring data to another Oracle database, it's far simpler (and usually faster) to transfer data over a database link. In other words, don't take the data out of the database if you're just going to put it back into another Oracle database.

11) Use Oracle Parallelism

For scans of huge tables or indexes, it is often good practice to invoke Oracle Parallel Query Option (PQO.) On most systems, using a parallel degree of 6 gives excellent performance. Of course, this assumes that you have the spare resources (disk and cpu). For example:

```
Select /*+Parallel (T 6) */ Emp, Name from Giant_Table T
```

Remember that Oracle will actually start a total of 2x the degree specified (one set for reading, another set to process/sort). So, the example above will actually result in 12 processes.

It is usually best to invoke parallelism via a SQL hint, rather than setting the table to a degree higher than one. If you set the table to invoke parallelism, Oracle will tend to start up the slave processes for many queries, whether you actually want parallelism or not.

12) Use Bind Variables

If you will be repeating a massive number of SQL statements, it is very important to properly use bind variables. In this way, the database engine avoids a re-parse of each statement. Also, a large number of unique SQL statements tends to flood the shared pool, causing other SQL statements to be released as the engine makes room for the new SQL. This will annoy the DBA.

13) Allow the Database Engine to Crunch the Statistics

It is rarely necessary to transfer the actual raw data from the database to the report server. If you mistakenly transfer huge amounts of data out of the database, you will pay the price of a large network delay, plus the time for the report server to crunch the data. The database engine will very easily retrieve the data, and then throw it at the report server. In fact, it is very common for the report server to be overwhelmed when you ask it to process a million rows of data. It is much better to leave the raw data in the database, and allow the database engine to crunch the statistics. This can easily be 100x faster.

14) Consider Parallel DML

Similar to parallel queries, you can invoke Oracle parallelism to perform transactions in parallel. There are many restrictions, however, that limit the situations where you can do this. Probably the most significant limitation is that you cannot have an active trigger on the table being manipulated.

My tests show that the gain from executing parallel transactions is not nearly as great as that for parallel queries.

Consider Multithreading

For cases where you need to perform lots of near-identical transactions against a massive table, consider a “multi-threading” strategy. Although some program re-design will be necessary, the gain is typically very large. At first, this technique might seem crude, this practice has proven very successful.

Let’s try an example to see how this works. Suppose you want to update the description (say, remove the leading blanks) for a set of products at many stores. We will need to update the giant ***Product_Line*** table, which contains 100 million rows. A typical transaction might look like this:

```
Update Product_Line
Set Description = Ltrim(Description)
Where Item_Id = 900 and Location#  100;
```

Assume that we need to repeat this transaction for a total of 100,000 item/store pairs. Since the table is extremely large, most of the blocks will not be cached. This means that each update will require at least 1 disk read to retrieve the data block. This will be true no matter how we setup the indexes--you simply need to read the data block off disk.

At a typical disk i/o rate of 100 reads per second, this means a delay of 1,000 seconds--just to get the blocks to change. To this time we need to add the delay for executing 100,000 roundtrips across Sql*Net. We’ll assume a fairly quick time for each round-trip of only 5 ms, or 500 seconds more.

What alternatives do we have? We can:

- Option 1 (our baseline option): Run the DML 100,000 times, accessing each row via an index.
- Option 2: Store the list of rows to be updated in a temp table, then rewrite the code to perform a full scan of the *Product_Line* table, using the temp table to identify the rows to be updated.
- Option 3: Same as option 2, but invoke Oracle Parallel DML to launch multiple processes.
- Option 4: Run the DML once, using a bulk collect to identify a group of rows to be updated. Each row is accessed via an index. [note: trying to combine with Parallel DML was disaster--runaway parallel processes.]
- Option 5: Run 10 threads, each of which processes 10,000 rows via index.

Let's analyze each option in our thought experiment:

Option 1: This is our base option. The 100,000 physical reads mean a runtime of about 16 minutes just to access the relevant blocks in the table. To this we must add 8 minutes for the cost of making 100,000 roundtrips over Sql*Net. Total runtime: 24 minutes.

Option 2: We perform one full scan instead of 100,000 index lookups. A table this large typically has about 3 million Oracle blocks (and the vast majority will not be cached). Assuming an average read of 2,000 Oracle blocks per second, this means a delay of about 1,500 seconds. Total runtime 25 minutes.

Option 3: Actual testing with parallel DML shows a performance of 2.5x. This means a runtime of about 25 minutes/2.5 = 10 minutes. (Note that future versions of Oracle might show a better performance gain.)

Option 4: The bulk collect eliminates the numerous Sql*Net round-trips. However, it does not eliminate the 24-minute disk-access time. Total runtime: 16 minutes.

Option 5: We eliminate the Sql*Net roundtrips, but we still must account for the disk access time of 16 minutes. The key here is that we spread this time over the 10 threads. Of course, the time will not go down linearly to 1.6 minutes, but many systems will show an excellent trend. We assume a 20% penalty. Total runtime: 2 minutes.

Note that all multi-threading schemes must account for the possibility of locking amongst the various threads. So each thread must be assigned a group of keys that cannot possibly conflict with any other group. This is really not very difficult in practice. Also, many threads operating simultaneously on a table bring up the possibility of ITL locking. This can be avoided by building a table with INI_TRANS set to more than one.

PARTITIONING ISSUES

Design Partitions to Match the Queries

Partitioning a table does not automatically improve performance. The objective of partitioning is to reduce (or "prune") most of the data to be searched, leaving a small group that still meets your conditions. In order for this to happen, the partition key must match the condition used in your queries. For example, if your queries contain a date restriction, then you should partition by date.

There are two main ways that partitioning can speed up queries:

1) If we need to perform a full table scan, we can substitute a full *partition* scan. In this case, having lots of partitions really helps.
2) If we perform an index scan, the index will be much smaller, and the scan will be slightly more efficient. This boost may not too noticeable unless you are doing a huge number of index scans.

Indexes on Partitioned Tables typically should be LOCAL

A local index has partitions that are aligned one-for-one with the table partitions. So, if we have 256 partitions in a table, our local index will likewise have 256 partitions. It makes sense to give the indexes the same benefit as the table. Since we presumably partitioned the table in a way to reduce the data set under consideration, we normally want our indexes to achieve the same benefit.

Be Aware of Global Index Tradeoffs

A global index does NOT match the table partitioning. You partition it in a way *different* than the table. For example, you can have a different number of partitions or even partition the index using a different key. In most cases, however, a global index is non-partitioned--it is one big segment that spans all the partitions of the table. When most people say "global index" that's what they mean--a non-partitioned index.

We need a global index when our query cannot benefit from the way the table is partitioned. For example, assume we want to retrieve inventory information as of a certain sale_date. (Assume we have an index on sale_date.) The table has 256 partitions, but the partitions are keyed on STORE, not date. If our index is LOCAL, a query based on date will require combining the scans of all 256-index segments to get the result. If our index had been created global, Oracle could have just performed one read of the single (larger) index.

There is a further disadvantage to global indexes that makes them less desirable. If you modify any partition in the table, the global index becomes *Invalid*, and must be rebuilt. The DBAs do not like this feature.

When you create an index, it will be *global* unless you explicitly specify local! Similarly, if you add a primary key and specify the Using Index clause, the index will be global. So, it is easy to mistakenly create a bunch of global Indexes.