

Tuning the Performance of the Oracle7 Parallel Query Option

Gary Hallmark
Oracle Kernel Development

The parallel query option (PQO) is at the heart of the Oracle Warehouse strategy. The performance of PQO has risen steadily from Release 7.1 to Release 7.2 to Release 7.3.

The first part of this paper highlights the parallel query features in Release 7.1, 7.2, and 7.3, and shows how performance has improved in each release.

The second part of the paper presents a step-by-step tuning method for obtaining the best possible performance from the parallel query option in Oracle7 Release 7.2 and Release 7.3.¹

Release to Release Performance Comparison

In order to demonstrate the steadily increasing performance of successive Oracle7 PQO releases, we chose three kinds of decision support operations:

- index creation,
- complex queries on a conventional order-entry schema, and
- “drill down” support, using parallel *create table as select* to summarize order entry revenue along five dimensions—date, brand, shipping container, supplier country, and customer country.

Figure 2: Oracle7 PQO Performance shows relative performance improvement. We omit absolute

¹ Good performance is also achievable in Release 7.1, but in some cases different tuning steps are required. See an earlier version of this paper in the Proceedings of the 1995 European Oracle User Group for information on tuning Release 7.1.

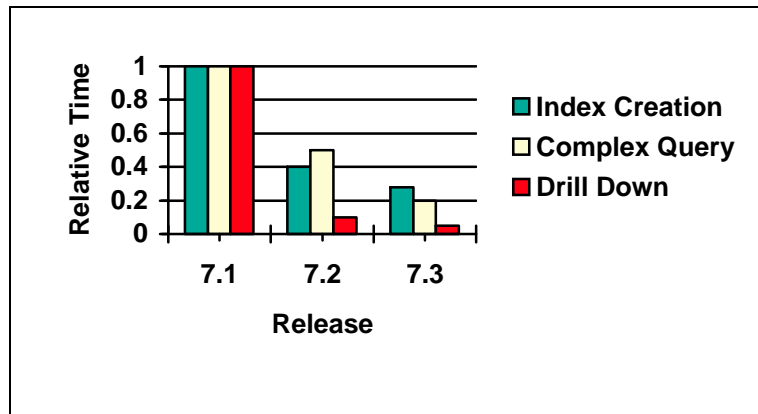


Figure 1: Release to Release Performance Comparison

numbers to avoid making claims about specific hardware platforms, which is not the intent of this paper. As can be seen, Release 7.3 creates indexes in parallel four times faster than Release 7.1, executes complex queries five times faster, and prepares summary tables for drill-down analysis 20 times faster.

Table 1 lists new features in each release that contribute to the performance gains. Also listed are the initial parallel features in Release 7.1.

Tuning PQO Performance

Database performance tuning can be broken down into 3 areas

- tuning the system parameters (init.ora and operating system),
- tuning the physical database layout, and
- tuning the SQL.

We examine each tuning area in turn.

Tuning the System Parameters

Start with some key system initialization parameters in the **init.ora** file before starting Oracle.

compatible = 7.3.1

This parameter enables all features that may prevent fallback to earlier releases. To be sure that you are getting full benefit of all performance features, set this equal to the current release. For example, direct read for table scans and sorts was introduced in 7.1.5. This feature greatly speeds up table scans on SMP platforms. If not set, all reads will go through the buffer cache. Reading through the buffer cache may result in greater pathlength, excessive memory bus utilization, and LRU latch contention on SMPs. Another example is temporary tablespaces introduced in 7.3. Temporary tablespaces improve efficiency of sort and hash joins.

Be sure to try this on a test database first before subjecting your production database to disk format changes!

db_block_size=8k

The database block size must be set when the database is created. If you are creating a new database, use a large block size.

db_block_multiblock_read_count = <high>

This parameter determines how many database blocks are read with a single operating system READ call. Many platforms limit the number of bytes read to 64K, limiting the effective maximum for an 8K block size to 8. Other platforms have a higher limit. Use the maximum value supported on your platform.

Feature	Index Create	Complex Query	Drill Down
<i>Release 7.1</i>			
parallel scan, join, sort, distinct, group by		✓	✓
parallel index create	✓		
direct read (7.1.5)	✓	✓	✓
<i>Release 7.2</i>			
sort direct write	✓	✓	✓
parallel create table as select	✓		✓
index direct write	✓		
unrecoverable create option	✓		✓
<i>Release 7.3</i>			
hash join		✓	
async readahead	✓	✓	✓
parallel union, not in		✓	
Cartesian product		✓	✓
histograms		✓	
MPP device affinity	✓	✓	✓

Table 1: Performance Features by Release

parallel_max_servers = <2 * max_degree * number_of_concurrent_users>

Most queries need at most twice the number of query server processes as the maximum degree of parallelism attributed to any table in the query. To support concurrent users, add more query servers. Note that if Oracle runs out of query servers when attempting to execute a query in parallel, it will either execute the query sequentially, or give an error if

parallel_min_percent is set. See “Determining the degree of parallelism” on page 4 for information on

determining the value of **max_degree**.

parallel_min_percent = 50

This parameter is new for Release 7.3. If fewer than this percentage of parallel query servers can be allocated for a query, the query gives an error rather than running with less parallelism. With a value of 50%, the variability of query response times is limited to a factor of 2 (given a constant average system load). The default value of this parameter is 0, meaning that the query may run with any degree of parallelism, or even run sequentially, based on available resources.

shared_pool_size = <4M + (2K * 1.5 * (max_degree+2) * parallel_max_servers)>

This parameter dimensions the size of the shared pool in global (SGA) memory. Message buffers for parallel query servers to communicate with each other are allocated from the shared pool. The 4M is a reasonable default in the absence of parallelism. The number of 2K communication buffers increases quadratically with high degrees of parallelism because each server process performing a first parallel operation, e.g., a table scan, may have to communicate with each server process performing a second parallel operation, e.g., a sort.

For example, 10 users each running a parallel degree 10 query need an 11 Meg shared pool. One user running a parallel degree 100 query needs a 64 Meg shared pool. In both cases, 200 query servers are used.

sort_area_size = 1M

This parameter controls the maximum amount of memory allocated for each sort.

If the sort area is too small, an excessive amount of I/O will be required to manage a large number of runs. If the sort area is too high the OS paging rate will be excessive. The cumulative sort area adds up fast because each parallel query server can allocate this amount of memory for each sort. Monitor the OS paging rate to see if too much memory is being requested.

Measurements have shown that large sort areas (over 10 Meg) do not perform much better than 1 Meg sort areas unless a disk sort is avoided altogether. In parallel sorts, large sort areas can actually degrade performance.

sort_area_retained_size = 1M

For decision support, this should always be set to the same value as **sort_area_size**. We assume that multithreaded servers are not used in a decision support environment.

sort_read_fac = 16

This parameter lets sort tradeoff multiblock reads for wider merges. The default is usually too low. It should not be greater than

db_block_multiblock_read_count..

sort_direct_writes = true

This parameter is new for Release 7.2. It bypasses the buffer cache for the writing of sort runs. Performance improvement can be a factor of 3 or more. It also removes the need to tune buffer cache and DBWR parameters.

A new value, **auto**, is available for this parameter in 7.3. It is the default, and is recommended.

sort_write_buffer_size = 32768

This parameter is new for Release 7.2. A bug in the initial release (7.2.2) requires the explicit setting of

this parameter. This parameter will be set automatically and correctly in subsequent releases.

hash_join_enabled = true

This parameter is new for Release 7.3. Use it to enable hash joins, which can be much faster than sort merge joins when one input is much smaller than the other.

hash_area_size = <as much as possible>

This parameter is new for Release 7.3. Unlike sort, which benefits from a large sort area only when a disk sort is completely avoided, hash joins always benefit from more memory. The same cautions about too much memory causing OS paging apply. A parallel hash join uses **hash_area_size** in each process.

hash_multiblock_io_count = ?

This parameter is new for Release 7.3. At press time, we have mixed input on a good value for this parameter. Preliminary data suggests that for joining large tables with moderate **hash_area_size**, the value of this parameter should be 1. With larger memory and/or smaller tables, the value of this parameter should be made larger (try a range of 2..16).

always_anti_join = hash

This parameter is new for Release 7.3. The NOT IN operator can be evaluated in parallel using a parallel hash anti-join. Without this parameter, NOT IN is evaluated as a (sequential) correlated subquery.

Tuning the Database Layout

Most of the remainder of this paper is concerned with tuning the physical database layout. To this end we discuss how to prepare a simple database for parallel query. We create, load, index, and analyze a 10 GB table called "tbl". We will spread it evenly over 10 devices.

Striping

In order to avoid I/O bottlenecks, all tablespaces accessed by multiple processes must be striped over at least as many devices as the degree of parallelism. This includes tablespaces for tables, tablespaces for indexes, and temporary tablespaces.

The operating system or volume manager can perform striping (OS striping), or Oracle can perform striping for parallel operations. For OS striping, use a stripe width at least as wide as the

db_block_multiblock_read_count factor. For Oracle striping, add multiple files, each on a separate device, to each tablespace. To stripe data during load, use the **file=** clause of parallel loader to load data from multiple load sessions into different files in the tablespace.

OS striping is usually more flexible and easier to manage than Oracle striping. It supports multiple users running sequentially as well as single users running in parallel. On the other hand, Oracle striping is a generic solution that works on all platforms and it performs a few percent better, especially on single-user parallel benchmarks. For any striping to be effective, you must also insure that there are enough controllers and other I/O components to support the bandwidth of parallel data movement into and out of the striped tablespaces.

Determining the degree of parallelism

Release 7.3 automatically computes the default degree of a table as the minimum of the number of devices storing the table and the number of CPUs available. We recommend that you explicitly set the degree of parallelism for tables in Release 7.2 to this value.

Parallel load

We now describe the steps to load a decision support database. We have chosen to use SQL loader to explicitly stripe data over a number of raw devices in a tablespace.

Create the tablespace

Below is the command to create a tablespace named “TStbl”. We specify a single datafile with the create command. Later we will specify nine more datafiles to add to TStbl. Oracle initializes each block in the datafile, so we want to add the datafiles in parallel to speedup up datafile initialization.

We make the initial extent size small because the first extent, allocated when each object in the

```
create tablespace TStbl
  datafile '/dev/D1P1' size 1024M reuse
  default storage (initial 1K next 100M pctincrease 0);
```

tablespace is created, is not used by the parallel loader.

We make next extent size large enough so that you do not run out of extents. The maximum number of extents is about 400 with 8k database blocks. In Release 7.3 objects can have an unlimited number of extents.

Always use a pctincrease value of 0.

The default storage parameters can be customized for each object created in the tablespace, if needed.

Add datafiles in parallel

In order to speed datafile initialization, add the

```
alter tablespace TStbl
  add datafile '/dev/D2P1' size 1024M reuse;
...
alter tablespace TStbl
  add datafile '/dev/D10P1' size 1024M reuse;
```

datafiles in parallel. Use a scripting language like *csh* or *perl* to fork multiple instances of *sqldba*, *server manager*, or *sql plus* in line mode

You could instead use extendible datafiles to avoid having to initialize all the data blocks at the time you add the datafiles, but this works only for OS files, not for raw partitions.

Create as many datafiles as the degree of parallelism you will use for creating objects in the tablespace. This will reduce fragmentation and consequent waste of space in the tablespace. Create multiple datafiles even if you are using OS stripes.

Create the table

```
create table tbl (c number, t char(100))
  tablespace TStbl
  parallel (degree 10);
```

We create a table named “tbl” in the TStbl tablespace.

We define a parallel degree of 10 because there are 10 devices in the tablespace TStbl, and we assume we are running on a machine with 10 or more CPUs. Override the default storage of TStbl if needed.

Load the table in parallel

We will use 10 processes loading into 10 devices.
Use a script language like *perl* or *cs*h to fork the 10 SQL loader (*sqlldr*) processes.

```
sqlldr gary/secret control.f log.f error.f data1.f
      direct=true parallel=true file=/dev/D1P1;
...
sqlldr gary/secret control.f log.f error.f data10.f
      direct=true parallel=true file=/dev/D10P1;
```

Setup for parallel sort and hash join

For better space management performance we will use the new temporary tablespaces in Release 7.3. For Release 7.2, omit the **temporary** key word. In this example we will use an OS striped file over as many devices as the degree of parallelism of tables to sort (10 in our example). Call this */dev/stripe10*.

```
create tablespace TStemp temporary
      default storage (initial 4M next 4M pctincrease 0)
      datafile '/dev/stripe10' size 10240M reuse;
```

Alternatively, you can add multiple files from different non-striped devices to TStemp (like we did for TStbl) and Oracle will stripe temporary results over the multiple files. If you use a massively parallel platform, each file in the tablespace should be local to a different node in the MPP.

Also use the *init.ora* parameter **sort_direct_writes = true**. This will bypass the buffer cache for writing intermediate runs, resulting in a factor of 3 to 10 speedup for tasks like index creation, analyze, and sort-merge joins.

Parallel sort and hash join, unlike parallel load, use the initial extent of every segment, so the initial extent size should be the same as the next. In 7.3, the combination of unlimited extents per segment and temporary tablespaces reduce the need to worry about the size of the extents, other than wasted space if the extent size is very large. For 7.2, making the extent size small can result in exceeding the maximum number of extents per segment (about 400 with 8k blocks), and can cause excessive contention for the global lock *ST00*, effectively serializing a parallel sort. For this example, an extent size of 40 Meg is better than the 4 Meg shown. Making the extents too large can use an excessive amount of disk space, because on average half of the last extent allocated per parallel query process will be unused.

Finally, do not forget to make your temporary tablespace available for use:

```
alter user gary
temporary tablespace
TStemp;
```

Parallel create index

After loading data and setting up for sorts, the next step is to create indexes. The considerations for creating the index tablespace are similar to those for creating the base data tablespace.

You can choose to either use OS striping or you can add multiple files from different devices to TSidx (like we did for TStbl) and Oracle will stripe temporary results over the multiple files. Create as many files in the tablespace as the degree of parallelism used to create indexes in the tablespace. This will reduce fragmentation.

Both initial and next extent sizes are used, and there is only 1 segment for the index so the extent sizes should be chosen so that the number of extents does not exceed the limit (about 400 with an 8k block size). Choose pctincrease 0. The default storage parameters can be customized for each object created in the tablespace, if needed.

After the tablespace is created, we create an index "I".

```
create index I on tbl(c)
      tablespace TSidx
      parallel (degree 10)
      unrecoverable;
```

The *parallel* clause specifies the number of query server processes to scan "tbl" and the number of query server processes to sort and build the index. Twenty query servers processes will be used in total.

The *unrecoverable* clause specifies that no redo log records are to be written when building the index. Although this speeds up index creation significantly, a disk failure after the index is created but before it is backed up will cause index "I" to be marked corrupt.

Index creation (and create table as select) use the direct load path to bypass the buffer cache when building indexes. This improves performance and eliminates the need to tune buffer cache and DBWR parameters.

Tuning the SQL

Analyze

After the data is loaded and indexed, analyze your data. The `analyze` command does not execute in parallel. For large databases, it is not practical to use the `compute` option of `analyze`. You should use the `estimate` option instead. The default number of samples is about 1000. For more accurate statistics, try 10,000 or more samples. Accuracy does not usually improve when the sample size is a few percent of the data.

Most real-world data is not uniformly distributed. In this case, use histograms in Release 7.3.

When you analyze a table, the indexes that are defined on that table are also analyzed. If you have several tables to analyze, you can use a scripting language to submit several `analyze` commands in parallel. Do not attempt to analyze a table and an index on that table in parallel.

Execute the SQL

After analyzing your tables and indexes you should be able to run queries and see speedup that scales linearly with the degree of parallelism used. The following operations should scale:

- table scans,
- nested loop join (with parallel index probes),

- sort merge join,
- hash join,
- group by,
- select distinct,
- aggregation,
- order by, and
- create table as select.

Observe and evaluate execution. Parallel execution generates large quantities of performance data. Use graphical CPU and I/O meters to visually absorb it.

Start with simple parallel queries. Evaluate total I/O throughput with `select count(*) from tbl`. Evaluate total CPU power by adding a complex `where` clause. I/O imbalance may suggest a better physical database layout. After you understand how simple scans work, add aggregation, joins, and other operations that reflect individual aspects of the overall workload. Again, look for bottlenecks.

Don't just monitor query performance— also monitor parallel load and parallel index creation and look for good utilization of I/O and CPU resources.

Explain plan

You can use the `explain plan` command to view the sequential and parallel query plan. The **other_tag** column of the plan table summarizes how each plan step is parallelized. Optimizer cost information for selected plan steps is given in the **cost**, **bytes**, and

other_tag text	interpretation
(blank)	serial execution
serial_to_parallel	serial execution, output of step is partitioned or broadcast to parallel query servers
parallel_to_parallel	parallel execution, output of step is repartitioned to second set of parallel query servers
parallel_to_serial	parallel execution, output of step is returned to serial “query coordinator” process
parallel_combined_with_parent	parallel execution, output of step goes to next step in same parallel process. No interprocess communication to pass data to next plan step.
parallel_combined_with_child	same as for parallel_combined_with_parent

Table 2: Using EXPLAIN to inspect parallelism

cardinality columns. Table 2 summarizes the meaning of the **other_tag** column:

The SQL script in Figure 5 can be used to extract a compact hierarchical plan from the output of explain:

Figure 3 is a query plan from the complex suite of TPC-D queries.

For complex queries, it is good to draw a picture of the query plan “tree”. Use different arrows as the arcs. For example, use a thin arrow to indicate data flow in the same process, a fat arrow to indicate data flow between parallel processes, and a medium arrow to mean data flow from the parallel processes back to the coordinator.

Figure 4 is a graphical rendering of the above plan. This plan is “good” in the sense that all operations are parallel.

Rewriting the SQL

Much has been written (both correctly and incorrectly) about how to rewrite SQL or use hints in order to gain performance for sequential query execution. We will not address that issue.

The most important issue for parallel query execution is insuring that all parts of the query plan that process a substantial amount of data execute in parallel. Use explain plan and see that all plan steps have an **other_tag** of *parallel_to_parallel*, *parallel_to_serial*, *parallel_combined_with_parent*, or *parallel_combined_with_child*. Any other keyword (or null) indicates serial execution, and a possible bottleneck.

Following are some transformations that can increase the optimizer’s ability to generate parallel plans:

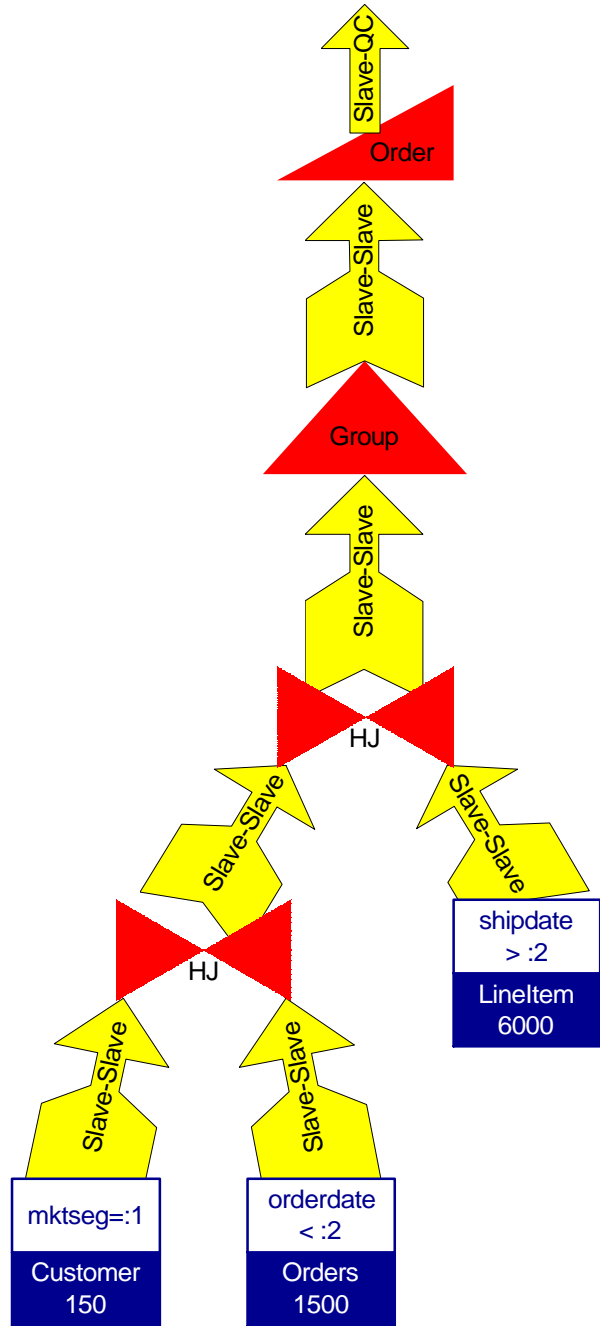


Figure 4: Graphical Explain Output

```

Q3 (32921, 5695333, 7309)
  SORT ORDER BY (,,) PARALLEL_TO_SERIAL
  SORT GROUP BY (,,) PARALLEL_TO_PARALLEL
  HASH JOIN (,,) PARALLEL_TO_PARALLEL
  HASH JOIN (,,) PARALLEL_TO_PARALLEL
    TABLE ACCESS FULL CUSTOMER (3000,60000,) PARALLEL_TO_PARALLEL
    TABLE ACCESS FULL ORDERS (72911,5759969,) PARALLEL_TO_PARALLEL
  TABLE ACCESS FULL LINEITEM (322763,23884462,) PARALLEL_TO_PARALLEL
  
```

Figure 3: Textual Explain Output

- convert subqueries, especially correlated subqueries, into joins,
- use a PL/SQL function instead of a correlated subquery,
- don't create indexes on columns with fewer than several hundred distinct values,
- rewrite queries with distinct aggregates as nested queries. For example, rewrite

```
select count(distinct C) from T
to
select count(*)
from (select distinct C from T)
```

Parallel table creation

A new tool in Release 7.2 for rewriting SQL is the parallel version of the *create table as select* statement. When combined with the *unrecoverable* option, parallel create table provides a very efficient temporary table facility.

```
create table temp
parallel (degree 10) unrecoverable
as select * from tbl;
```

Some advantages of temporary tables are

- Common subqueries can be computed once and referenced many times. This may be much more efficient than referencing a complex view many times.
- Complex queries can be decomposed into simpler steps for ease of debugging.
- Complex star queries that arise in multidimensional databases can be decomposed into simpler star queries that the CBO understands how to execute effectively.
- Index range scans can be used to create a table, and that table can be scanned in parallel. This compensates for lack of parallel index range scan.
- Parallel deletes can be implemented efficiently by creating a new table that omits

```
select
  substr( lpad(' ',2*(level-1)) ||
    decode(id, 0, statement_id, operation) ||
    ' ' || options || ' ' || object_name ||
    ' (' || cardinality || ', ' || bytes ||
    ', ' || cost || ') ' || other_tag ,
    1, 79) "step (card,bytes,cost) par"
from plan_table
start with id = 0
connect by prior id = parent_id
and prior nvl(statement_id,' ') =
nvl(statement_id,' ');
```

Figure 5: SQL for textual EXPLAIN

the unwanted rows from the original table. The original table can then be dropped.

- Partitioned tables can be simulated by creating many tables that are tied together with a union-all view.
- Summary tables for multidimensional drill-down analysis can be created efficiently. For example, a summary table might store the sum of revenue grouped by month, brand, region, and salesperson.

Conclusion

Oracle is committed to improving the performance of the parallel query option in every release. With some care spent tuning the system, the data layout, and the application, users can take full advantage of the new features to build a high performance, scalable data warehouse.