Maximize Data Warehouse Performance with Parallel Queries

Christo Kutrovsky
Pythian
Ottawa, ON, Canada

Keywords:

Parallel query, performance, partition join, partition table

Introduction

The goal of this presentation is to provide practical knowledge on using Oracle Parallel Query on large amounts of data for aggregation and processing. Fully understanding Oracle Parallel Query is critical to achieving high query performance

Oracle Parallel Query Overview

The Oracle Data warehousing manual has plenty of concepts and examples on how Oracle Parallel query works. But a lot of the details and combinations are either left out or scattered through the document. Using the correct parallelization method is just as critical as using the correct access path (ex. Index vs Full Table Scan) for a performant parallel query. The analysis and decision is made by the CBO Optimizer. And just like the optimizer can someone pick a sub-par execution plan for non-parallel queries, the same can happen with parallel queries as well. Understanding the parallel execution paths is just as important as understanding regular execution plans.
The Oracle manual talks about producer consumer model and how a parallel query slave is either a producer or consumer. Although entirely correct, this is not always the case and depends on the point of view. In respect to other parallel slaves, an Oracle parallel slave is either producer or consumer. However in respect to dataflow, a slave can be both producer and consumer as well as flow trough. For example the Query Coordinator, although been a parallel query slave, is always a flow-through slave. From data perspective, it's receiving data from other slaves, and sending it back to client. From query slave's perspective, it's a consumer. Slides 9 to 14 with this paper illustrate the concepts behind this.
This is a key aspect to keep in mind when understanding parallel query operations

Parallel Query Operations Explained - The Big Group By (Aggregation)

Aggregation is perhaps the most classical case of data analysis. Aggregating sales data by region, by country, by product, by department and etc. Endless possibilities.
Oracle Implements various optimizations for group by operations in order to reduce the amount of data that needs to move around nodes. For example on Slide 41 the "group by push down" optimization is demonstrated. As a query slave receives data, it doesn't immediately send it to the correct slave for "grouping". It firsts performs an "in place" grouping which reduces the amount of data that needs to be send to another node. Depending on the data, this can be extremely efficient.

Where it can hurt is when the query in question will have very few rows per "group" and thus the grouping operation will be performed twice. Once on the slave that reads the data, and once again on the slave that is responsible for grouping that section of the data.
Performing this optimization also requires twice the amount of memory than without it, as both producer slaves and consumer slaves need to perform the grouping.
Oracle has also two methods of grouping data. They are known as SORT GROUP BY and HASH GROUP BY. It's actually two methods of grouping data. In the SORT method data is grouped by RANGE while in the HASH method it is grouped based on hashing function.
The range method can be very difficult to get to scale properly. The end points for each slave need to be pre-defined at the start of the operation. Let's imagine we are grouping data by last name, in that case Oracle may decide to split last names starting with A to D on slave 1, E to I on slave 2 and etc. However last names are not uniformly distributed in the alphabet. Thus some slaves will get more data than others. This will cause dis-balance of the operation and increased overall execution time.
The hash method usually does not suffer from such issues, however if the final result set is to be sorted, the hash method requires another reshuffling while the range (SORT GROUP BY) does not.
Slides 41 to 55 have examples of queries and their response time, and details on why the response time was affected.
Note that Oracle offers a very extensive summary management capabilities knows as "materialized view" and "query rewrite". These features allow Oracle to pickup summary tables if available to transparently optimize queries. This is the single most efficient way of analyze summary data, as the benefits are tremendous.

Parallel Query Operations Explained - The Big Join

With classical Star Schema design, dimension keys allow separating metadata from data, and such "data" becomes smaller in size. In addition when summarized, these same dimension tables with metadata are utilized to apply all required aspects of data.
Oracle offers 4 types of joins:

- Nested Loops joins – usually used in conjunction with indexes
- Hash Joins – usually used in conjunction with full table or partition scans
- Sort Merge joins – rarely used as hash joins are more efficient. Some edge cases when sorting is required or when input data is already sorted
- Cartesian joins – these are mostly used for joins with no join conditions

When dealing with parallel queries, hash joins are predominant, but sometimes nested loops are also interesting options.
Hash joins in Parallel queries are usually executed in three steps. Step 1 reads the table resulting in a smaller result set, and hashes into memory on the join key(s). The second step involves each parallel query slave reading a section of the table expecting to return the larger result set, and sending the record to the appropriate query slave based on join key. The receiving slave then produces and stores a joined record if matched.
The final 3th steps is sending the result sets to the next step (aggregation for example) or if none, directly to query coordinator.
This method is called HASH JOIN BUFFERED with PX SEND HASH distribution method. This method requires as much memory (or temp space) as the resulting join from both tables. If the join result is a lot of records, which is often the case, the requirements for temporary storage are very large.

Another popular method is HASH JOIN with PX SEND BROADCAST. This join is executed in two steps. First step is reading the table which will produce the smaller result set, and sending the result set to each parallel query slave. The second step is each of the parallel query slaves that has received a full copy of the "smaller" table reads a section of the table with the larger result set and searches for matches. If one is found a record is produced and sent to the next step or query coordinator.
In this method, the amount of total memory required is the size of the "smaller" result set that is sent to every slave, times the number of slaves.
The requested degree of parallelism can make Oracle choose one or the other method depending on expected result set sizes.

Parallel Query Operations Explained – Joining Two Large Tables Efficiently

In some circumstances, two large tables need to be joined on a regular basis. Using large amounts of temp space can be very inefficient. For such cases, partitioning can be very useful.
Oracle has an optimization when joining tables when at least one of the tables is partitioned on at least one of the join keys. Oracle can do a semi-partition join (one partitioned table) or a full partition join (both tables partitioned). These are two very different type of joins.
Semi Partitioned Join – in this case one of the tables is partitioned on at least one column of the join conditions. Depending on the column's data distribution, great efficiencies can be achieved. Ideally the column will be some kind of relatively unique number (large number of distinct values) and the partitioning method will be hash.
In this case, the join will be HASH JOIN with a distribution method of PX SEND PARTITION (KEY) and will be executed in 2 steps. Oracle will start by having each slave read a section of the non-partitioned table, and distribute the data based on the partition key of the partitioned table. In the second step, each slave that received a portion of the non-partitioned table, will read it's assigned partition from the partition-data. If a matching record is found, it will be "produced" directly to next step or query-coordinator.
In this case, the amount of memory required for performing the join is the size of the result set from the non-partitioned table. Presumably this means a much smaller number, thus a more efficient join.
A full partitioned join is when both tables are partitioned on at least one join key. In this case the join is performed in two steps. A slave is assigned to a matching partition pair from both tables. It will read and hash into memory the table resulting in smaller result set, and then it will read the "larger" table and join in memory.
The amount of memory required is size of the result set of "smaller" table times the parallelism degree.
In both cases additional optimizations can be achieved. In semi-partition join a JOIN FILTER is used to determine which partitions should be read. In some cases partitions can be skipped if they are known to produce no data based on a bloom filter build on the smaller table..

Parallel Query and Indexes

Dealing with large amounts of data does not exclude indexes from been used. Bitmap indexes are often overlooked as the right solution in some cases. Bitmap indexes can be combined at very high speeds to reduce the amount of data that needs to be accessed.
However there are some aspects of parallel query and indexes that need to be in place in order to be able to leverage index use in parallel.

DIRECT INDEX ACCESS

Direct Index access via key='value' cannot be parallelized, unless the table is partitioned. When the table is not partitioned, there is only one index structure, thus only one index tree to traverse. Index trees cannot be traversed in parallel, unless reading the entire index end to end (known as FAST FULL SCAN).

When a table is partitioned and the index is created with "local" value, then each partition's index can be accessed in parallel and values can be streamed from each partition at the same time. Examples are on slide 80.

JOINS WITH INDEX

Joins with indexes are a little bit different. When joining, there's a list of values that need to be scanned from the index, so the requirement for partitioning does not exist.

When joining with indexes, nested loops are required. The join is performed in just one step. Each parallel query server will read a partition of the smaller table and for each value found it will probe the index for the other table and immediately produce a row if one is found.

The difference between this method and the other hash join methods are that data been read will go through the Oracle buffer cache. Depending on the values, index structure and etc, some level of contention may happen, if multiple processes are trying to read the same data blocks. Also in RAC environments, same data may be requested from different nodes, thus sent around as required. Example of this on Slide 81.

Final Thoughts

Using Oracle to analyze large amounts of data requires to use a platform that has the data flow capabilities to do so. Exadata is such platform and has additional optimizations that allow you to further maximize the hardware available.

However with all complex machinery, the optimum path may not be instantly visible and some analysis and work is required to adjust some of the most frequently used reports or the ones that require the most amount of resources to complete. Oracle has a huge amount of features available to increase the efficiency of such operations, especially sumarisations.

Contact address:

Name
Pythian
1200 St.Laurent Blvd
Unit 261
Ottawa, ON Canada
K1K 3B8

Phone: +1-613-565-8696 x1233
Email kutrovsky@pythian.com