|  |
| --- |
|  |
|  |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| REVISION HISTORY | | | | | |
| Ver. | Description of Change | Author | Date | Approved | |
| Name | Effective Date |
| 1.0 | Initial status | [Kiryl Bucha](mailto:Kiryl_Bucha@epam.com) | 12-JAN-2012 |  |  |
| 2.0 | Updated in accordance with renewed content | [Elias Nema](mailto:Elias_Nema@epam.com) | 20-JAN-2014 |  |  |

Contents

[1. Parallel Execution Concepts 3](#_Toc383292545)

[1.1. Parallel Execution of SQL Statements 4](#_Toc383292546)

[1.2. Dividing Work Among Parallel Execution Servers 4](#_Toc383292547)

[1.3. Producer or Consumer Operations 6](#_Toc383292548)

[1.4. How Parallel Execution Servers Communicate 7](#_Toc383292549)

[1.5. Granules 7](#_Toc383292550)

[1.6. Parallel Partition-Wise Joins 8](#_Toc383292551)

[2. Parallel Programming 8](#_Toc383292552)

[3. Summary 9](#_Toc383292553)

[3.1. Block-range parallelism. 9](#_Toc383292554)

[3.2. Partition-based parallelism. 9](#_Toc383292555)

[4. Source Books and Articles 10](#_Toc383292556)

# Parallel Execution Concepts

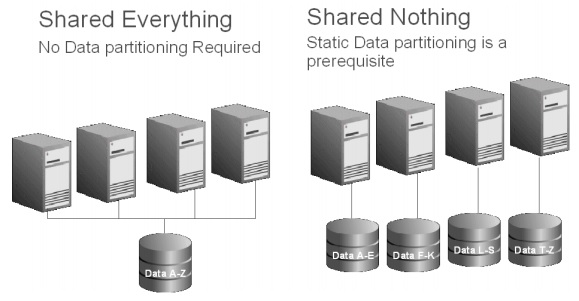
Parallel execution is a commonly used method of speeding up operations by splitting a task in smaller sub tasks. In this section we will discuss the basic reasoning around parallel execution and the basic concepts. Furthermore we will discuss the Oracle Parallels Execution concepts in detail.

Imagine that your task is to count the number of cars in a street. There are two ways to do this, one, you can go through the street by yourself and count the number of cars or you can enlist a friend and then the two of you can start on opposite ends of the street, count cars until you meet each other and add the results of both counts to complete the task. Assuming your friend counts equally fast as you do, you expect to complete the task of counting all cars in a street in roughly half the time compared to when you perform the job all by yourself. If this is the case then your operations scales linearly; 2x the number of resources halves the total processing time. The database is not very different from the counting cars example. If you allocate twice the number of resources and achieve a processing time that is half of what it was with the original amount of resources, then the operation scales linearly. Scaling linearly is the ultimate goal of parallel processing, both in counting cars as well as in delivering answers from a database query.

In the counting car example we made some basic assumptions to get to linear scalability. These assumptions reflect some of the theory behind parallel processing. First of all we chose to use just the two of us to do the counting. Here we decided the parallel degree as it is called in a database. In other words, how many of us would be ideal to solve the problem fastest. The bigger the workload, the more people we could use and of course, if there is a short street with 4 cars, we should avoid any parallelism as it would take longer to decide who starts where than it takes to just count the cars.

So we decided that the overhead of having the two of us count and coordinate is worth the effort. In a database this choice is made by the query optimizer based on the cost of the operation. Secondly, in the car example we divided the work in 2 equal parts as each of us started on one end of the street and we assumed each counted with the same speed. The same goes for parallel processing in a database. The first step is to divide the data work on in chunks of similar size allowing them to be processed in the same amount of time. Some form of hashing algorithm is often used to divide the data.

This partitioning of data is implemented in two basic ways. The main differentiation is whether or not physical data partitioning is used as a foundation – and therefore a static pre-requisite – for parallelizing the work. These fundamental approaches are known as shared everything architecture and shared nothing architecture respectively.



**Figure 1 Shared Everything vs. Shared Nothing**

In a shared nothing system, the system is divided into individual parallel processing units. Each processing unit has its own processing power (cores) and its own storage component (disk) and its CPU cores are solely responsible for its individual data set on its own disks. The only way to access a specific piece of data is to use the processing unit that owns this subset of data. Such systems are also commonly known as Massively Parallel Processing (MPP) systems. In order to achieve a good workload distribution shared nothing systems have to use a hash algorithm to partition data evenly across all available processing units. The partitioning strategy has to be decided upon initial creation of the system.

As a result, shared nothing systems introduce mandatory, fixed parallelism in their systems in order to perform operations that involve table scans; the fixed parallelism completely relies on a fixed static data partitioning at database or object creation time. Most non-Oracle data warehouse systems are shared nothing systems.

Oracle Database relies on a shared everything architecture. This architecture does not require any pre-defined data partitioning to enable parallelism; however by using Oracle Partitioning, Oracle Database can deliver the exact same parallel processing capabilities as a shared nothing system. It does so however without the restrictions of the fixed parallel access encompassed in the data layout. Consequently Oracle can parallelize almost all operations in various ways and degrees, independent of the underlying data layout.

By using a shared everything architecture Oracle allows flexible parallel execution and high concurrency without overloading the system, using a superset of parallel execution capabilities over shared nothing vendors.

If you execute a statement in parallel, the Oracle Database will parallelize as many of the individual steps as possible and reflects this in the execution plan. If we were to re-execute the two statements above in parallel we could get the following execution plans.

SQL parallel execution in the Oracle database is based on a few fundamental concepts. The following section discusses these concepts that help you understand the parallel execution setup in your database and read the basics of parallel SQL execution plans.

## Parallel Execution of SQL Statements

Each SQL statement undergoes an optimization and parallelization process when it is parsed. If parallel execution is chosen, then the following steps occur:

1. The user session or shadow process takes on the role of a coordinator, often called the query coordinator.
2. The query coordinator obtains the necessary number of parallel servers.
3. The SQL statement is executed as a sequence of operations (a full table scan to perform a join on a nonindexed column, an ORDER BY clause, and so on). The parallel execution servers performs each operation in parallel if possible.
4. When the parallel servers are finished executing the statement, the query coordinator performs any portion of the work that cannot be executed in parallel. For example, a parallel query with a SUM() operation requires adding the individual subtotals calculated by each parallel server.
5. Finally, the query coordinator returns any results to the user.

After the optimizer determines the execution plan of a statement, the parallel execution coordinator determines the parallel execution method for each operation in the plan. For example, the parallel execution method might be to perform a parallel full table scan by block range or a parallel index range scan by partition. The coordinator must decide whether an operation can be performed in parallel and, if so, how many parallel execution servers to enlist. The number of parallel execution servers in one set is the degree of parallelism (DOP).

## Dividing Work among Parallel Execution Servers

SQL parallel execution in the Oracle Database is based on the principles of a coordinator (often called the Query Coordinator – QC for short) and parallel execution (PX) server processes. The QC is the session that initiates the parallel SQL statement and the PX servers are the individual sessions that perform work in parallel. The QC distributes the work to the PX servers and may have to perform a minimal – mostly logistical – portion of the work that cannot be executed in parallel. For example a parallel query with a SUM() operation requires a final adding up of all individual sub-totals calculated by each PX server.

The **QC**is easily identified in the parallel execution plans below as 'PX COORDINATOR'. The process acting as the QC of a parallel SQL operation is the actual user session process itself.

The **PX servers** are taken from a pool of globally available PX server processes and assigned to a given operation. All the work shown below the QC entry in our sample parallel plan is done by the PX servers. PX server processes can be easily identified on the OS level, for example on Linux they are the oracle processes ORA\_P\*\*\*.

The parallel execution coordinator examines each operation in a SQL statement's execution plan then determines the way in which the rows operated on by the operation must be divided or redistributed among the parallel execution servers. As an example of parallel query with intra- and inter-operation parallelism, consider next:

EXPLAIN PLAN FOR

SELECT /\*+ PARALLEL(4) \*/ customers.cust\_first\_name, customers.cust\_last\_name,

MAX(QUANTITY\_SOLD), AVG(QUANTITY\_SOLD)

FROM sales, customers

WHERE sales.cust\_id=customers.cust\_id

GROUP BY customers.cust\_first\_name, customers.cust\_last\_name;

PLAN\_TABLE\_OUTPUT

---------------------------------------------------------------------------------------

Plan hash value: 4060011603

Query Coordinator

---------------------------------------------------------------------------------------

| Id | Operation | Name | Rows | Bytes | TQ |IN-OUT|PQ Distr|

---------------------------------------------------------------------------------------

| 0 | SELECT STATEMENT | | 925| 25900 | | | |

| 1 | PX COORDINATOR | | | | | | |

| 2 | PX SEND QC (RANDOM)| :TQ10003 | 925| 25900 | Q1,03 | P->S |QC(RAND)|

| 3 | HASH GROUP BY | | 925| 25900 | Q1,03 | PCWP | |

| 4 | PX RECEIVE | | 925| 25900 | Q1,03 | PCWP | |

| 5 | PX SEND HASH | :TQ10002 | 925| 25900 | Q1,02 | P->P | HASH|

Parallel Servers do majority of the work

|\* 6 | HASH JOIN BUFFERED | | 925| 25900 | Q1,02 | PCWP | |

| 7 | PX RECEIVE | | 630| 12600 | Q1,02 | PCWP | |

| 8 | PX SEND HASH | :TQ10000 | 630| 12600 | Q1,00 | P->P | HASH |

| 9 | PX BLOCK ITERATOR | | 630| 12600 | Q1,00 | PCWC | |

Producers

| 10 | TABLE ACCESS FULL| CUSTOMERS| 630| 12600 | Q1,00 | PCWP | |

| 11 | PX RECEIVE | | 960| 7680 | Q1,02 | PCWP | |

| 12 | PX SEND HASH | :TQ10001 | 960| 7680 | Q1,01 | P->P | HASH |

| 13 | PX BLOCK ITERATOR | | 960| 7680 | Q1,01 | PCWC | |

| 14 | TABLE ACCESS FULL| SALES | 960| 7680 | Q1,01 | PCWP | |

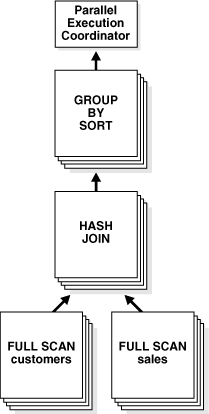
---------------------------------------------------------------------------------------

Predicate Information (identified by operation id):

---------------------------------------------------

6 - access("SALES"."CUST\_ID"="CUSTOMERS"."CUST\_ID")

26 rows selected.



**Figure 2 Dataflow of Parallel Execution**

Given two sets of parallel execution servers SS1 and SS2 for the query plan illustrated in Figure 2, the execution proceeds as follows: each server set (SS1 and SS2) has four execution processes because of the PARALLEL hint in the query that specifies the DOP.

Child set SS1 first scans the table customers and sends rows to SS2, which builds a hash table on the rows. In other words, the consumers in SS2 and the producers in SS1 work concurrently: one in scanning customers in parallel, the other is consuming rows and building the hash table to enable the hash join in parallel. This is an example of inter-operation parallelism.

After SS1 has finished scanning the entire customers table, it scans the sales table in parallel. It sends its rows to servers in SS2, which then perform the probes to finish the hash-join in parallel. After SS1 has scanned the sales table in parallel and sent the rows to SS2, it switches to performing the GROUP BY operation in parallel. This is how two server sets run concurrently to achieve inter-operation parallelism across various operators in the query tree.

Another important aspect of parallel execution is the redistribution of rows when they are sent from servers in one server set to servers in another. For the query plan in next example, after a server process in SS1 scans a row from the customers table, which server process in SS2 should it send it to? The operator into which the rows are flowing decides the redistribution. In this case, the redistribution of rows flowing up from SS1 performing the parallel scan of customers into SS2 performing the parallel hash-join is done by hash partitioning on the join column.

That is, a server process scanning customers computes a hash function of the value of the column customers.cust\_id to decide the number of the server process in SS2 to send it to. The redistribution method used in parallel queries explicitly shows in the Distrib column in the EXPLAIN PLAN of the query. This can be seen on lines 5, 8, and 12 of the EXPLAIN PLAN.

## Producer or Consumer Operations

Operations that require the output of other operations are known as consumer operations. In Figure2, the GROUP BY SORT operation is the consumer of the HASH JOIN operation because GROUP BY SORT requires the HASH JOIN output.

Consumer operations can begin consuming rows as soon as the producer operations have produced rows. In last example, while the parallel execution servers are producing rows in the FULL SCAN of the sales table, another set of parallel execution servers can begin to perform the HASH JOIN operation to consume the rows.

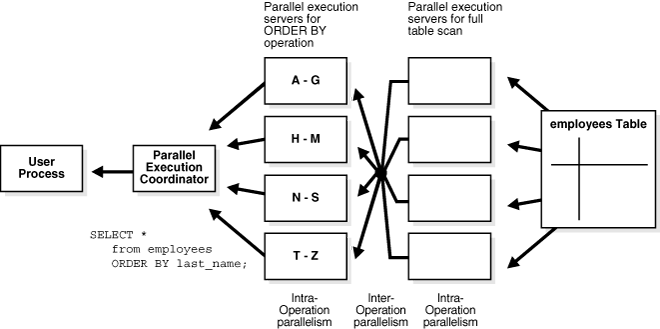
Operations (row-sources) that are processed by the same set of PX servers can be identified in an execution plan by looking in the TQ column. The first slave set (Q1,00) is reading table CUSTOMERS in parallel and producing rows that are sent to slave set 2 (Q1,02) that consumes these records and joins then to record coming from the SALES table (Q1,01). Whenever data is distributed from producers to consumers you will also see an entry of the form :TQxxxxx (Table Queue x) in the NAME column.

Each of the two operations performed concurrently is given its own set of parallel execution servers. Therefore, both query operations and the data flow tree itself have parallelism. The parallelism of an individual operation is called intra-operation parallelism and the parallelism between operations in a data flow tree is called inter-operation parallelism. Due to the producer-consumer nature of the Oracle Database operations, only two operations in a given tree must be performed simultaneously to minimize execution time. To illustrate intra- and inter-operation parallelism, consider the following statement:

SELECT \* FROM employees ORDER BY last\_name;

The execution plan implements a full scan of the employees table. This operation is followed by a sorting of the retrieved rows, based on the value of the last\_name column. For the sake of this example, assume the last\_name column is not indexed. Also, assume that the DOP for the query is set to 4, which means that four parallel execution servers can be active for any given operation.

Figure 3 illustrates the parallel execution of the example query.



**Figure 3 Inter-operation Parallelism and Dynamic Partitioning**

As illustrated in Figure 3, there are actually eight parallel execution servers involved in the query even though the DOP is 4. This is because a producer and consumer operator can be performed at the same time (inter-operation parallelism).

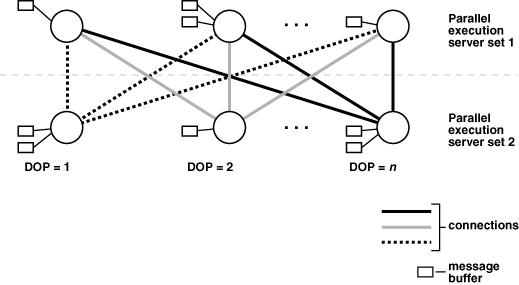
Also, note that all of the parallel execution servers involved in the scan operation send rows to the appropriate parallel execution server performing the SORT operation. If a row scanned by a parallel execution server contains a value for the last\_name column between A and G, that row is sent to the first ORDER BY parallel execution server. When the scan operation is complete, the sorting processes can return the sorted results to the query coordinator, which, in turn, returns the complete query results to the user.

## How Parallel Execution Servers Communicate

To execute a query in parallel, Oracle Database generally creates a set of producer parallel execution servers and a set of consumer parallel execution servers. The producer server retrieves rows from tables and the consumer server performs operations such as join, sort, DML, and DDL on these rows. Each server in the producer set has a connection to each server in the consumer set. The number of virtual connections between parallel execution servers increases as the square of the degree of parallelism.

Each communication channel has at least one, and sometimes up to four memory buffers, which are allocated from the shared pool. Multiple memory buffers facilitate asynchronous communication among the parallel execution servers.

A single-instance environment uses at most three buffers for each communication channel. An Oracle Real Application Clusters environment uses at most four buffers for each channel. Figure 4 illustrates message buffers and how producer parallel execution servers connect to consumer parallel execution servers.



**Figure 4 Parallel Execution Server Connections and Buffers**

When a connection is between two processes on the same instance, the servers communicate by passing the buffers back and forth in memory (in the shared pool). When the connection is between processes in different instances, the messages are sent using external high-speed network protocols over the interconnect. In Figure 4, the DOP equals the number of parallel execution servers, which in this case is n. Figure 4 does not show the parallel execution coordinator. Each parallel execution server actually has an additional connection to the parallel execution coordinator. It is important to size the shared pool adequately when using parallel execution. If there is not enough free space in the shared pool to allocate the necessary memory buffers for a parallel server, it fails to start.

## Granules

A granule is the smallest unit of work when accessing data. Oracle database uses a shared everything architecture, which from a storage perspective means that any CPU core in a configuration can access any piece of data; this is the most fundamental architectural difference between Oracle and all other database vendors on the market. Unlike all other systems, Oracle can – and will - choose this smallest unit of work solely dependent on a query's requirements.

The basic mechanism the Oracle Database uses to distribute work for parallel execution is block ranges on disk – so-called block-based granules. This methodology is unique to Oracle and is independent of whether the underlying objects have been partitioned. Access to the underlying objects is divided into a large number of granules, which are given out to PX servers to work on (and when a PX server finishes the work for one granule the next one is given out).

The number of granules is always much higher than the requested DOP in order to get an even distribution of work among parallel server processes. The operation 'PX BLOCK ITERATOR ' shown in execution plan, literally is the iteration over all generated block range granules.

Although block-based granules are the basis to enable parallel execution for most operation, there are some operations that can benefit from the underlying data structure and leverage individual partitions as granules of work. With partition-based granules only one PX server performs the work for all data in a single partition. The Oracle Optimizer considers partition-based granules if the number of (sub)partitions accessed in the operation is at least equal to the DOP (and ideally much higher if there may be skew in the sizes of the individual (sub)partitions). The most common operations that use partition-based granules are partition-wise joins, which will be discussed later.

Based on the SQL statement and the degree of parallelism, the Oracle Database decides whether block-based or partition-based granules lead to a more optimal execution; you cannot influence this behavior.

## Parallel Partition-Wise Joins

If at least one of the tables accessed in the join has been partitioned on the join key the database may decide to use a partition-wise join. If both tables are equi-partitioned on the join key the database may use a full partition-wise join. Otherwise a partial partition-wise join may be used in which one of the tables is dynamically partitioned in memory followed by a full partition-wise join.

A partition-wise join does not require any data redistribution because individual PX servers will work on the equivalent partitions of both joined tables.

*Note that partition-wise joins use partition-based granules rather than block-based granules.*

The partition-wise join is the fundamental enabler for shared nothing systems. Shared nothing systems typically scale well as long as they can take advantage of partition-wise joins. As a result, the choice of partitioning (distribution) in a shared nothing system is key as well as the access path to the tables. Operations that do not use partition-wise operations in an MPP system often do not scale well.

# Parallel Programming

While not a new concept in Oracle Database 11g Release 2, table functions are an important aspect of parallelism. The goal of a set of table functions is to build a parallel processing pipeline leveraging the parallel processing framework in the database.

A table function encapsulates complex logic in a PL/SQL construct while allowing you to process your data in parallel. A table function also allows you to stream data to the next consumer, building up the pipeline with providers and consumers. This behavior can be best viewed when looking at a small SQL example:

select \*

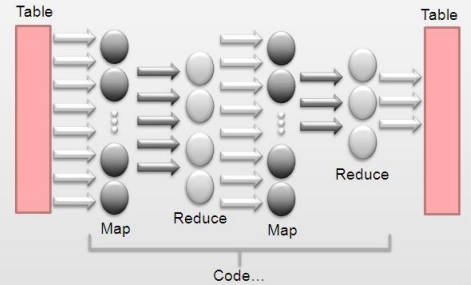
from table(oracle\_map\_reduce.reducer(cursor(

select \* from table(oracle\_map\_reduce.mapper(cursor(

select \* from sls))) map\_result)));

As can be seen, the inner most action is a select from a table, this data then streams into a parallel table function which, after manipulating the data streams it into the next table function. That table function then acts as the data source for the ultimate select.

As the naming above implies, this is very similar to the current popular MapReduce framework used on top of Hadoop. This mechanism provides an alternative for SQL developers to perform very complex processing in a procedural way, not easily expressed with SQL. It also follows the MapReduce paradigm, enabling massively parallel processing within the realm of the database.



**Figure 5 MapReduce Approach**

# Summary

The objective of parallel execution is to reduce the total execution time of an operation by using multiple resources concurrently. Resource availability is the most important prerequisite for scalable parallel execution.

The Oracle database provides a powerful SQL parallel execution engine that can run almost any SQL-based operation – DDL, DML and queries – in the Oracle Database in parallel. This paper provides a detailed explanation of how parallel execution is implemented in the Oracle Database and provides a step by step guide to enabling and using SQL parallel execution successful.

Data warehouses should always leverage parallel execution to achieve good performance. Specific operations in OLTP applications, such as batch operations, can also significantly benefit from parallel execution.

Two types of parallelism are possible within an Oracle Database:

## Block-range parallelism.

Block-range parallelism is implemented by dynamically breaking a table into pieces, each of which is a range of blocks, and then using multiple processes to work on these pieces in parallel. Oracle’s implementation of block-range parallelism is unique in that it doesn’t require physically partitioned tables to achieve parallelism.

With block-range parallelism, the client session that issues the SQL statement transparently becomes the parallel execution coordinator, dynamically determining block ranges and assigning them to a set of parallel execution (PE) processes. Once a PE process has completed an assigned block range, it returns to the coordinator for more work. Not all I/O occurs at the same rate, so some PE processes may process more blocks than others. This notion of “stealing work” allows all processes to participate fully in the task, providing maximum leverage of the machine resources.

A useful analogy for dynamic parallelism is eating a pie. The pie is the set of blocks to be read for the operation, and the goal is to eat the pie as quickly as possible using a certain number of people. Oracle serves the pie in helpings, and when a person finishes his first helping, they can come back for more. Not everyone eats at the same rate, so some people will consume more pie than others. While this approach in the real world is somewhat unfair, it’s a good model for parallelism because if everyone is eating all the time, the pie will be consumed more quickly. The alternative is to give each person an equal serving and wait for the slower eaters to finish.

## Partition-based parallelism.

Since partitioned tables were introduced in Oracle8, an operation may involve one, some, or all of the partitions of a partitioned table. There is essentially no difference in how block-range parallelism dynamically splits the set of blocks to be read for a regular table as opposed to a partitioned table. Once the optimizer has determined which partitions should be accessed for the operation, all the blocks of all partitions involved are treated as a pool to be broken into ranges. A small subset of Oracle’s parallel functionality is based on the number of partitions or subpartitions accessed by the statement to be parallelized. For block-range parallelism, the piece of data each PE process works on is a range of blocks. For partition-based parallelism, the pieces of data that drive parallelism are partitions or subpartitions of a table.

# Source Books and Articles

1. Oracle Essentials: Oracle Database 12c; Rick Greenwald, Robert Stackowiak, Jonathan Stern, 2013.
2. Leveraging Massively Parallel Processing in an Oracle Environment for Big Data Analytics; An Oracle Whitepaper, 2010.
3. Oracle® Database VLDB and Partitioning Guide; Oracle Corporation, 2008.