

Parallel Execution with Oracle Database

WHITE PAPER / FEBRUARY 20, 2019

PURPOSE STATEMENT

This document provides an overview of features and enhancements included in the Oracle Database. It is intended solely to help you assess the business benefits of upgrading and to plan your I.T. projects.

DISCLAIMER

This document in any form, software or printed matter, contains proprietary information that is the exclusive property of Oracle. Your access to and use of this confidential material is subject to the terms and conditions of your Oracle software license and service agreement, which has been executed and with which you agree to comply. This document and information contained herein may not be disclosed, copied, reproduced or distributed to anyone outside Oracle without prior written consent of Oracle. This document is not part of your license agreement nor can it be incorporated into any contractual agreement with Oracle or its subsidiaries or affiliates.

This document is for informational purposes only and is intended solely to assist you in planning for the implementation and upgrade of the product features described. It is not a commitment to deliver any material, code, or functionality, and should not be relied upon in making purchasing decisions. The development, release, and timing of any features or functionality described in this document remains at the sole discretion of Oracle.

Due to the nature of the product architecture, it may not be possible to safely include all features described in this document without risking significant destabilization of the code.

TABLE OF CONTENTS

Purpose Statement	2
Introduction	4
Parallel Execution Concepts	5
Why use parallel execution?	5
The theory of parallel execution	5
Parallel Execution in Oracle	7
Processing parallel SQL statements	7
In-Memory Parallel Execution	19
Controlling Parallel Execution	21
Enabling parallel execution	21
Managing the degree of parallelism	21
Managing the concurrency of parallel operations.....	24
Conclusion	29

INTRODUCTION

The amount of data stored in databases have been growing exponentially over the recent years in both transactional and data warehouse environments. In addition to the enormous data growth users require faster processing of the data to meet business requirements.

Parallel execution is key for large scale data processing. Using parallelism, hundreds of terabytes of data can be processed in minutes, not hours or days. Parallel execution uses multiple processes to accomplish a single task. The more effectively the database can leverage all hardware resources - multiple CPUs, multiple IO channels, multiple storage units, multiple nodes in a cluster - the more efficiently queries and other database operations will be processed.

Large data warehouses should always use parallel execution to achieve good performance. Specific operations in OLTP applications, such as batch operations, can also significantly benefit from parallel execution. This paper covers three main topics:

- **Fundamental concepts of parallel execution** – why should you use parallel execution and what are the fundamental principles behind it.
- **Oracle's parallel execution implementation and enhancements** – here you will become familiar with Oracle's parallel architecture, learn Oracle-specific terminology around parallel execution, and understand the basics of how to control and identify parallel SQL processing.
- **Controlling parallel execution in the Oracle Database** – this last section shows how to enable and control parallelism within the Oracle environment, giving you an overview of what a DBA needs to think about.

PARALLEL EXECUTION CONCEPTS

Parallel execution is a commonly used method of speeding up operations by splitting a task into 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 parallel execution concepts in detail.

Why use parallel execution?

Imagine that your task is to count the number of cars in a street. There are two ways to do this, 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, your car counting operation 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 operation.

The theory of parallel execution

In the counting cars 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 so-called 'degree of parallelism' (DOP) as it is called in a database. But 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 only, we should avoid any parallelism as it would take longer to decide who starts where than it takes to just count the cars.

We decided that the "overhead" of having the two of us count and coordinate is worth the effort. In a database the database engine, based on the total cost of the operation, should make this decision.

Secondly, in the car example we divided the work in two 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 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 evenly divide the data.

This "partitioning of data" for parallel processing is commonly done in two basic, but fundamentally different ways. The main differentiation is whether or not physical data partitioning (placement) is used as a foundation – and therefore as static prerequisite – for parallelizing the work.

These fundamental conceptually different approaches are known as **shared everything** architecture and **shared nothing** architecture respectively.

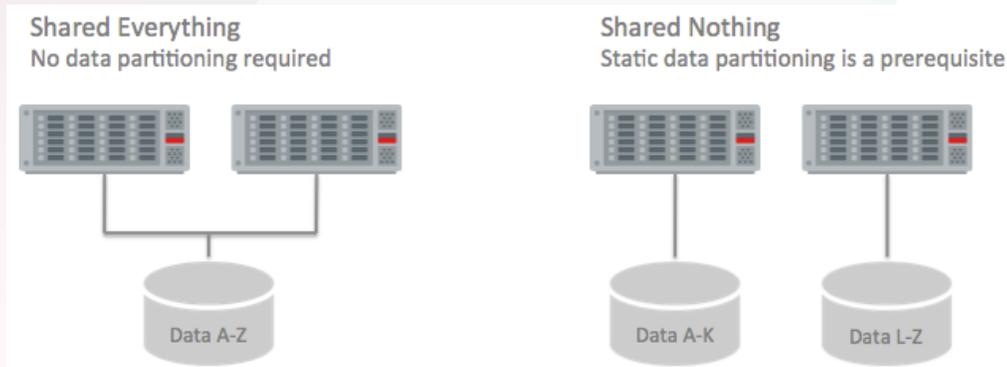


Figure 1: Shared everything versus shared nothing

In a **shared nothing system**, the system is physically divided into individual parallel processing units. Each processing unit has its own processing power (CPU cores) and its own storage component; its CPU cores are solely responsible for its individual data set on its own storage. 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. Data partitioning is a fundamental prerequisite for these systems. In order to achieve a good workload distribution shared nothing systems have to use a hash algorithm to statically partition data evenly across all available processing units. The data partitioning strategy that controls the data placement has to be decided upon initial creation of the system.

As a result, shared nothing systems introduce mandatory, fixed minimal parallelism in their systems in order to perform operations that involve table scans; the fixed parallelism completely relies on the fixed static data partitioning at database or object creation time: the number of parallel processing units determines the minimal degree of parallelism to access all partitions of the data. 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; all of the data is accessible from all processing units without limitations; the degree of parallelism for an operation is decoupled from the actual data storage. However, by using Oracle Partitioning, Oracle Database can operate on the same processing paradigm, offering the exact same parallel processing capabilities as a shared nothing system. It is worth noting that it does so 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, in addition to the parallel capabilities of a shared nothing system. 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.

PARALLEL EXECUTION IN ORACLE

The Oracle Database provides functionality to perform complex tasks in parallel, without manual intervention. Operations that can be executed in parallel include but are not limited to:

- Data loads
- Queries
- DML statements
- RMAN backups
- Object creation, e.g. index or table creation
- Optimizer statistics collection

This paper focuses on SQL parallel execution only, which consists of parallel query, parallel DML (Data Manipulation Language) and parallel DDL (Data Definition Language).

Processing parallel SQL statements

When you execute a SQL statement in the Oracle Database it is decomposed into individual steps or row sources, which are identified as separate lines in an execution plan. Below is an example of a simple SQL statement that touches just one table and its execution plan. The statement returns the total number of customers in the `CUSTOMERS` table:

```
SELECT COUNT(*) FROM customers c;
```

Line ID	Operation	Name
0	⊞ SELECT STATEMENT	
1	⊞ SORT AGGREGATE	
2	└─ TABLE ACCESS FULL	CUSTOMERS

Figure 2: Serial execution plan of a `COUNT(*)` on the `CUSTOMERS` table

A more complex serial execution plan would be one that includes a join between multiple tables. In the example below, information about purchases made by customers is requested. This requires a join between the `CUSTOMERS` and `SALES` tables.

```
SELECT c.cust_first_name, c.cust_last_name, s.amount_sold  
FROM customers c, sales s  
WHERE c.cust_id=s.cust_id;
```

Line ID	Operation	Name
0	⊞ SELECT STATEMENT	
1	⊞ HASH JOIN	
2	└─ TABLE ACCESS FULL	CUSTOMERS
3	└─ TABLE ACCESS FULL	SALES

Figure 3: More complex serial execution plan showing a two table join

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:

Line ID	Operation	Name
0	SELECT STATEMENT	
1	SORT AGGREGATE	
2	PX COORDINATOR	
3	PX SEND QC (RANDOM)	:TQ10000
4	SORT AGGREGATE	
5	PX BLOCK ITERATOR	
6	TABLE ACCESS FULL	CUSTOMERS

Figure 4: Parallel execution plan of a COUNT(*) on the CUSTOMERS table

Line ID	Operation	Name
0	SELECT STATEMENT	
1	PX COORDINATOR	
2	PX SEND QC (RANDOM)	:TQ10002
3	HASH JOIN BUFFERED	
4	PX RECEIVE	
5	PX SEND HASH	:TQ10000
6	PX BLOCK ITERATOR	
7	TABLE ACCESS FULL	CUSTOMERS
8	PX RECEIVE	
9	PX SEND HASH	:TQ10001
10	PX BLOCK ITERATOR	
11	TABLE ACCESS FULL	SALES

Figure 5: Customer purchase information, parallel plan

These plans look quite a bit different than before, mainly because we are having additional logistical processing steps due to the parallel processing that we did not have before.

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 in your database and provides the basics of how to read parallel SQL execution plans.

QUERY COORDINATOR (QC) AND PARALLEL EXECUTION (PX) 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 processes that perform work in parallel on behalf of the initiating session. 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 which is done by the QC.

The **QC** is easily identified in the parallel execution plans above as 'PX COORDINATOR' (for example Line ID 1 in Figure 5 shown above). 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 (the setup is discussed in a later section) for the lifetime of the operation. The PX servers are doing all the work shown below the QC entry in our sample parallel plans (Figure 4, Figure 5).

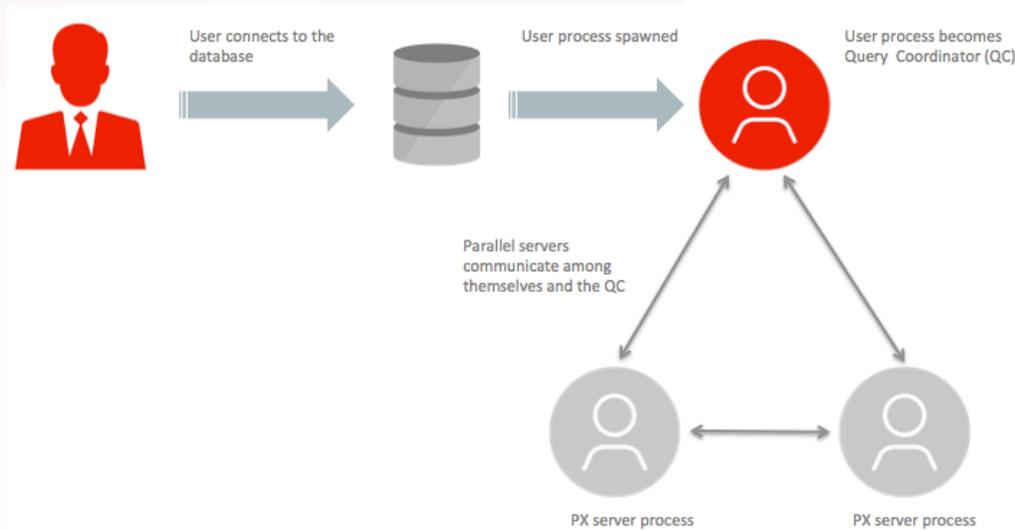


Figure 6: Parallel Execution with the Query Coordinator and a set of PX server processes

PX server processes can be easily identified on the OS level, for example on Linux they are the processes ora_p***:

```
oracle 58203 1 0 Oct30 ? 00:01:25 ora_p000_dbm1
oracle 58205 1 0 Oct30 ? 00:01:02 ora_p001_dbm1
oracle 58207 1 0 Oct30 ? 00:01:02 ora_p002_dbm1
oracle 58209 1 0 Oct30 ? 00:01:03 ora_p003_dbm1
oracle 58211 1 0 Oct30 ? 00:00:33 ora_p004_dbm1
```

Figure 7: PX server processes seen on the Linux OS level using 'ps -ef'

Going back to the example of counting the cars: you and your friend are acting as PX servers and there would be a third person – the QC - telling you and your friend to go ahead and count the cars.

You can do exactly the same on the road that is being done internally in the database with the SQL and execution plan shown in Figure 8: the only difference is that in this example the database is counting customers and there are no road sides it can use to distribute the work; we will discuss the work distribution in a second in the “Granules” section. You and your friend will go ahead and count the cars on your side; this is equivalent to the operations with the **Line ID 4**, **Line ID 5**, and **Line ID 6**, where **Line ID 5** is the equivalent to tell each one of you to only count the cars on your side of the road.

Line ID	Operation	Name
0	SELECT STATEMENT	
1	SORT AGGREGATE	
2	PX COORDINATOR	
3	PX SEND QC (RANDOM)	:TQ10000
4	SORT AGGREGATE	
5	PX BLOCK ITERATOR	
6	TABLE ACCESS FULL	CUSTOMERS

Figure 8: QC and PX server processes shown in an execution plan

After having counted your part of the road each of you tells the third person – the QC - your individual subtotals (**Line ID 3**) and he or she then adds up your two individual counts to get the final result (**Line ID 1**). This is the hand-over from the PX servers (processes doing the actual work) to the QC for final “assembly” of the result for returning it to the user process.

Using SQL Monitor¹ helps to easily identify the work being done by PX servers – many little blue or red “people” in front of a given operation - versus serial execution – a single green person.

PRODUCER/CONSUMER MODEL

Continuing with our car counting example, imagine the job is to count the total number of cars per car color. If you and your friend each cover one side of the road, each one of you potentially sees the same colors and gets a subtotal for each color, but not the complete result for a given color for the street. You could go ahead, memorize all this information and tell it back to the third person (the “person in charge”). But this poor individual then has to sum up all of the results by himself – what if all cars in the street were a different color? The third person would redo exactly the same work as you and your friend just did.

To parallelize the counting on a per-color base you simply ask two more friends to help you out: these friends both walk in the middle of the road with you, one of them getting the count of all dark colors from you and your friend scanning the road sides, the other one all of the bright colors (assuming this “car color separation” is approximately splitting the information in half). Whenever you count a new car, you tell the person that is in charge of this color about the new encounter – you **produce** the information, **redistribute** it based on the color information, and the color counter **consumes** the information. At the end, both color counting friends tell their result the person in charge – the QC – and you’re done; we had two sets of parallel workers, each with two friends doing a part of the job, working hand in hand.

That’s exactly how the database works: in order to execute a statement in parallel efficiently, sets of PX servers work in pairs: one set is producing rows (**producer**) and one set is consuming the rows (**consumer**). For example, for the parallel join between the `SALES` and `CUSTOMERS` tables one set of PX servers reads the tables and sends the data to another set which receives the data (consumer) and joins the two tables, as shown in Figure 9, the standard plan output of the `DBMS_XPLAN` package.

¹ SQL Monitor provides a very effective way to monitor and analyze the steps of SQL execution in detail..

Id	Operation	Name	Rows	Bytes	Cost (%CPU)	Time	TQ	IN-OUT	PQ D1
0	SELECT STATEMENT		960	44160	6 (17)	00:00:01			
1	PX COORDINATOR								
2	PX SEND QC (RANDOM)	:TQ10002	960	44160	6 (17)	00:00:01	Q1,02	P->S	QC (R)
3	HASH JOIN BUFFERED		960	44160	6 (17)	00:00:01	Q1,02	PCWP	
4	PX RECEIVE		630	12600	3 (0)	00:00:01	Q1,02	PCWP	
5	PX SEND HASH	:TQ10000	630	12600	3 (0)	00:00:01	Q1,00	P->P	HASH
6	PX BLOCK ITERATOR		630	12600	3 (0)	00:00:01	Q1,00	PCWC	
7	TABLE ACCESS FULL	CUSTOMERS	630	12600	3 (0)	00:00:01	Q1,00	PCWP	
8	PX RECEIVE		960	24960	2 (0)	00:00:01	Q1,02	PCWP	
9	PX SEND HASH	:TQ10001	960	24960	2 (0)	00:00:01	Q1,01	P->P	HASH
10	PX BLOCK ITERATOR		960	24960	2 (0)	00:00:01	Q1,01	PCWC	
11	TABLE ACCESS FULL	SALES	960	24960	2 (0)	00:00:01	Q1,01	PCWP	

Figure 9: Producers and consumers

Operations (row sources) that are processed by the same set of PX servers can be identified in an execution plan by looking at the TQ column. As shown in Figure 9, the first PX server set (Q1,00) is reading table CUSTOMERS in parallel and producing rows that are sent to PX server set 2 (Q1,02) that consumes these records and joins them to records coming from the SALES table (Q1,01). SQL Monitor shows the different sets of PX servers working on a parallel statement in alternating colors, making the identification of boundaries of units of work and where data has to be redistributed easier. Whenever data is distributed from producers to consumers you will see an entry of the form :TQxxxxxx (Table Queue x) in the NAME column as data output. Please disregard the content of the other columns in Figure 9 for now.

This producer/consumer model has a very important consequence for the number of PX servers that are allocated for a given parallel operation: the producer/consumer model expects two sets of PX servers for a parallel operation, so the number of PX servers is twice the requested degree of parallelism (DOP). For example, if the parallel join in Figure 9 runs with parallel degree of 4, then 8 PX servers will be used for this statement, 4 producers and 4 consumers.

The only case when PX servers do not work in pairs is if the statement is so basic that only one set of PX servers can complete the entire statement in parallel. For example, `SELECT COUNT(*) FROM customers;` requires only one PX server set (see Figure 4).

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 most of the other database vendors. Unlike these 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 – so-called **block-based granules**. These blocks may reside on storage, or in memory in the case of In-Memory Parallel Execution, which will be discussed later in this paper. This methodology is unique to Oracle and is independent of whether the objects have been partitioned or not. 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).

Line ID	Operation	Name
0	SELECT STATEMENT	
1	SORT AGGREGATE	
2	PX COORDINATOR	
3	PX SEND QC (RANDOM)	:TQ10000
4	SORT AGGREGATE	
5	PX BLOCK ITERATOR	
6	TABLE ACCESS FULL	CUSTOMERS

Figure 10: Block-based granules in the customer count example.

The number of granules is always much higher than the requested DOP in order to get an even distribution of work among PX servers. The operation 'PX BLOCK ITERATOR' shown in Figure 10 literally is the iteration over all generated block range granules.

Although block-based granules are the basis to enable parallel execution for most operations, there are some operations that can benefit from an underlying partitioned 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 DOP, the Oracle Database decides whether block-based or partition-based granules lead to a more optimal execution; you cannot influence this behavior.

In the car counting example, one side of the street – or even a block of a long street - could be considered the equivalent of a block-based granule. The existing data volume – the street – is subdivided into physical pieces on which the PX servers – you and your friend – are working on independently. With many blocks on a long road we can have only you and your friend working on it, each covering half of the blocks (“granules”). Alternatively, we can have you and three friends working on it, each covering one quarter of the blocks. You can choose the number of friends you work with, and your counting will scale.

If we were to consider the road as being “statically partitioned” with a left and a right side curb as partitions then you can only ask one friend to help you: there is nothing to work on for your other friends. Using such a static approach is how pure shared nothing systems work and shows the limitation of such an architecture.

DATA REDISTRIBUTION

Parallel operations – except for the most basic ones – typically require data redistribution. Data redistribution is required in order to perform operations such as parallel sorts, aggregations and joins. At the block-granule level there is no knowledge about the actual data contained in an individual granule; block-granules are just physical chunks without logical connotation. Consequently data has to be redistributed as soon as a subsequent operation relies on the actual content. In the car example the car color mattered, but you don't know – or even control – what color cars are parked where on the street. You redistributed the information about the amount of cars per color to the additional two friends based on their color responsibility, enabling them to do the total counting for the colors they are in charge of.

Data redistribution takes place between individual PX server sets either within a single machine, or, across multiple machines – nodes – in a Real Application Clusters (RAC) system. Of course in the latter case interconnect communication is used for the data redistribution.

Data redistribution is not unique to the Oracle Database. In fact, this is one of the most fundamental principles of parallel processing, being used by every product that provides parallelism capabilities. The fundamental difference and advantage of Oracle's capabilities, however, is that parallel data access (discussed in the granules section earlier) and therefore the necessary data redistribution are not constrained by any given hardware architecture or database setup (data partitioning).

Like shared-everything systems, shared-nothing systems also require data redistribution unless operations can fully rely on partition-wise joins (as explained further down in this section). In shared-nothing systems parallel operations that cannot benefit from a partition-wise join – such as a simple three-way table join on two different join keys - always requires data redistribution and always makes heavy use of interconnect communication. Because the Oracle Database can enable parallel execution within the context of a node, parallel operations do not necessarily always have to use interconnect communication, thus avoiding a potential bottleneck at the interconnect.

The following section will explain Oracle's data redistribution capabilities using the simple example of a table join without any secondary data structures, such as indexes or materialized views, and other optimizations.

Serial Join

In a serial two-way join a single session reads both tables involved and performs the join. In this example we assume two large tables `CUSTOMERS` and `SALES` are involved in the join. The database uses full table scans to access both tables as previously shown in Figure 3.

For a serial join the single serial session scans both of the tables and performs the full join. Figure 11 depicts the serial join.

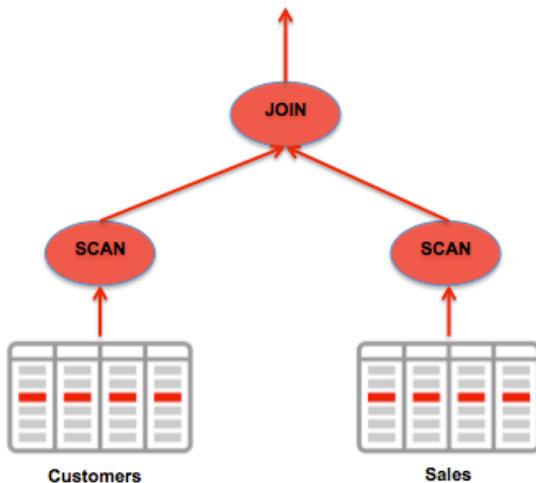


Figure 11: Serial join

Parallel Joins

Processing the same simple two-way join in parallel, a distribution of rows will become necessary to ensure that the data is properly divided for subsequent parallel processing. In this example PX servers scan physical parts of either table based on block ranges and in order to complete the join, rows have to be distributed based on the join key values between the PX server sets; you have to ensure that identical join key values are processed by the same PX server and that every row is processed only once.

Figure 12 depicts the data distribution for the parallel join shown before in Figure 9 at a DOP of 2. Since this join requires two sets of PX servers there are actually four PX servers allocated for this query, PX1 and PX2 read the tables, PX3 and PX4 perform the join. Both tables are read in parallel by both PX1 and PX2 using block-range granules and then each PX server distributes its result set based on the values of the join key to the subsequent parallel join operator; the same join key value from both tables has to be sent to the same PX server doing the join operation to ensure the data is joined correctly.

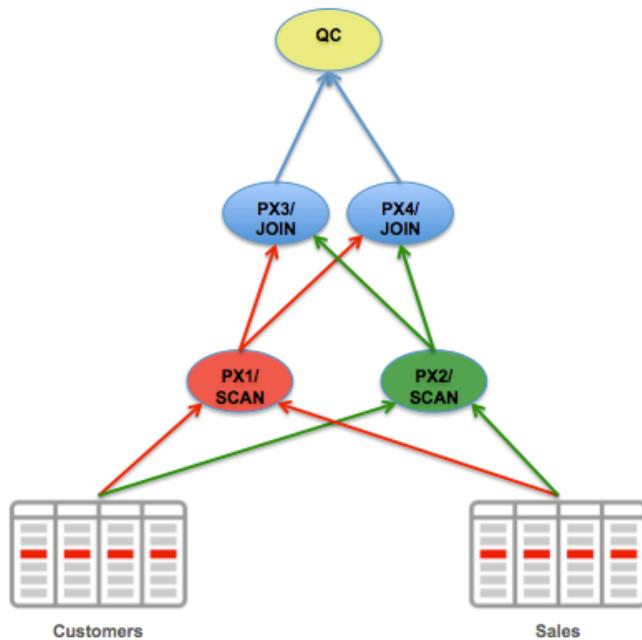


Figure 12: Data redistribution for a simple parallel join

There are many data distribution methods. The following are the most common ones:

HASH: Hash distribution is very common in parallel execution in order to achieve an equal distribution of work for individual PX servers based on a hash function. Hash distribution is the basic parallel execution enabling mechanism for most data warehouse systems.

Figure 13 below shows an execution plan that uses the hash distribution method. This is actually the plan for the join shown in Figure 12.

Id	Operation	Name	Rows	Bytes	Cost (%CPU)	Time	Pstart	Pstop	TQ	IN-OUT	PQ Dis
0	SELECT STATEMENT		960	27840	5 (20)	00:00:01					
1	PX COORDINATOR										
2	PX SEND QC (RANDOM)	:TQ10002	960	27840	5 (20)	00:00:01			Q1,02	P->S	QC (R)
* 3	HASH JOIN BUFFERED		960	27840	5 (20)	00:00:01			Q1,02	PCWP	
4	PX RECEIVE		630	12600	2 (0)	00:00:01			Q1,02	PCWP	
5	PX SEND HASH	:TQ10000	630	12600	2 (0)	00:00:01			Q1,00	P->P	HASH
6	PX BLOCK ITERATOR		630	12600	2 (0)	00:00:01			Q1,00	PCWC	
7	TABLE ACCESS FULL	CUSTOMERS	630	12600	2 (0)	00:00:01			Q1,00	PCWP	
8	PX RECEIVE		960	8640	2 (0)	00:00:01			Q1,02	PCWP	
9	PX SEND HASH	:TQ10001	960	8640	2 (0)	00:00:01			Q1,01	P->P	HASH
10	PX BLOCK ITERATOR		960	8640	2 (0)	00:00:01	1	16	Q1,01	PCWC	
11	TABLE ACCESS FULL	SALES	960	8640	2 (0)	00:00:01	1	16	Q1,01	PCWP	

Figure 13: Execution plan for hash distribution

Assuming a DOP of two for this plan, one PX set (PX1 and PX2) reads the `CUSTOMERS` table, applies a hash function on the join column and sends the rows to the PX servers of the other PX set (PX3 and PX4), this way PX3 gets some of the rows and PX4 gets the other rows depending on the computed hash value. Then PX1 and PX2 read the `SALES` table, apply a hash function on the join column and send the rows to the other PX set (PX3 and PX4). PX3 and PX4 each now have the matching rows from both tables and can perform the join.

The distribution method for each table can be seen in the plan in the `PQ Distrib` and `Operation` columns, here `PQ Distrib` shows `HASH` for both tables and `Operation` shows `PX SEND HASH` at lines 5 and 9.

BROADCAST: Broadcast distribution happens when one of the two result sets in a join operation is much smaller than the other result set. Instead of distributing rows from both result sets the database sends the smaller result set to all PX servers in order to guarantee the individual servers are able to complete their join operation. The advantage of broadcasting the smaller table of a join operation is that the larger table does not have to be redistributed at all.

Figure 14 below shows an execution plan that uses the broadcast distribution method.

Id	Operation	Name	Rows	Bytes	Cost (%CPU)	Time	Pstart	Pstop	TQ	IN-OUT	PQ Distrib
0	SELECT STATEMENT		960	44160	6 (17)	00:00:01					
1	PX COORDINATOR										
2	PX SEND QC (RANDOM)	:TQ10001	960	44160	6 (17)	00:00:01			Q1,01	P->S	QC (RAND)
* 3	HASH JOIN		960	44160	6 (17)	00:00:01			Q1,01	PCWP	
4	PX RECEIVE		630	12600	3 (0)	00:00:01			Q1,01	PCWP	
5	PX SEND BROADCAST	:TQ10000	630	12600	3 (0)	00:00:01			Q1,00	P->P	BROADCAST
6	PX BLOCK ITERATOR		630	12600	3 (0)	00:00:01			Q1,00	PCWC	
7	TABLE ACCESS FULL	CUSTOMERS	630	12600	3 (0)	00:00:01			Q1,00	PCWP	
8	PX BLOCK ITERATOR		960	24960	2 (0)	00:00:01			Q1,01	PCWC	
9	TABLE ACCESS FULL	SALES	960	24960	2 (0)	00:00:01			Q1,01	PCWP	

Figure 14: Execution plan for broadcast distribution

Assuming a DOP of two for this plan, one PX set (PX1 and PX2) reads the `CUSTOMERS` table and broadcasts all its result set to the other PX set (PX3 and PX4). PX3 and PX4 can now read the `SALES` table and perform the join since they both have all rows from the `CUSTOMERS` table.

In the execution plan we can see the distribution method in line 5, `PQ Distrib` column shows `BROADCAST` and `Operation` column shows `PX SEND BROADCAST`.

RANGE: Range distribution is generally used for parallel sort operations. Individual PX servers work on data ranges so that the QC does not have to do any sorting but only to present the individual PX server results in the correct order.

Figure 15 shows an execution plan that uses the range distribution method for a simple ORDER BY query.

Id	Operation	Name	Rows	Bytes	TempSpc	Cost (%CPU)	Time	Pstart	Pstop	TQ	IN-OUT	PQ Di
0	SELECT STATEMENT		960	28800		3 (34)	00:00:01					
1	PX COORDINATOR											
2	PX SEND QC (ORDER)	:TQ10001	960	28800		3 (34)	00:00:01			Q1,01	P->S	QC (0
3	SORT ORDER BY		960	28800	58000	3 (34)	00:00:01			Q1,01	PCWP	
4	PX RECEIVE		960	28800		2 (0)	00:00:01			Q1,01	PCWP	
5	PX SEND RANGE	:TQ10000	960	28800		2 (0)	00:00:01			Q1,00	P->P	RANGE
6	PX BLOCK ITERATOR		960	28800		2 (0)	00:00:01	1	16	Q1,00	PCWC	
7	TABLE ACCESS FULL	SALES	960	28800		2 (0)	00:00:01	1	16	Q1,00	PCWP	

Figure 15: Execution plan for range distribution

Assuming a DOP of two for this plan, one PX set (PX1 and PX2) reads the SALES table, PX1 and PX2 sends the rows they read to either PX3 or PX4 depending on the values of the columns in the ORDER BY clause. Each of PX3 and PX4 own a range of the data so they order the rows they get and send the result to the QC, the QC does not need to do any sorting since the rows are already sorted by PX3 and PX4. The QC only have to guarantee to return the rows first from the PX server that worked on the range that has to be returned first. For example, if the ordering is based on time_id with the newest data first, PX3 owns the data range before January and PX4 any newer data, then the QC first returns the sorted result of PX4 to the end user before returning the result from PX3 to ensure the correct ordering of the complete result.

In the execution plan we can see the distribution method in line 5, PQ Distrib column shows RANGE and Operation column shows PX SEND RANGE.

KEY: Key distribution ensures result sets for individual key values to be clumped together. This is an optimization that is primarily used for partial partition-wise joins (see further down) to ensure only one side in the join has to be distributed.

Figure 16 shows an execution plan that uses the key distribution method.

Id	Operation	Name	Rows	Bytes	Cost (%CPU)	Time	Pstart	Pstop	TQ	IN-OUT	PQ Di
0	SELECT STATEMENT		5760	410K	7 (15)	00:00:01					
1	PX COORDINATOR										
2	PX SEND QC (RANDOM)	:TQ10001	5760	410K	7 (15)	00:00:01			Q1,01	P->S	QC (R
3	HASH JOIN		5760	410K	7 (15)	00:00:01			Q1,01	PCWP	
4	PX RECEIVE		5760	146K	3 (0)	00:00:01			Q1,01	PCWP	
5	PX SEND PARTITION (KEY)	:TQ10000	5760	146K	3 (0)	00:00:01			Q1,00	P->P	PART
6	PX BLOCK ITERATOR		5760	146K	3 (0)	00:00:01			Q1,00	PCWC	
7	TABLE ACCESS FULL	SALES	5760	146K	3 (0)	00:00:01			Q1,00	PCWP	
8	PX PARTITION HASH ALL		630	29610	3 (0)	00:00:01	1	32	Q1,01	PCWC	
9	TABLE ACCESS FULL	CUSTOMERS	630	29610	3 (0)	00:00:01	1	32	Q1,01	PCWP	

Figure 16: Execution plan for key distribution

The CUSTOMERS table is hash partitioned on the join column whereas the SALES table is not partitioned. The plan shows that one PX set (PX1 and PX2) reads the SALES table and sends the rows to the other PX set servers (PX3 and PX4) based on the partitioning of the CUSTOMERS table. This way PX3 and PX4 can work on separate partitions at the same time since they have all matching rows from SALES for the partition they need to join. PX3 and PX4 have no row redistribution.

In the execution plan we can see the distribution method in line 5, PQ Distrib column shows PART (KEY) and Operation column shows PX SEND PARTITION (KEY).

HYBRID HASH: The hybrid hash method, introduced in Oracle Database 12c, is an adaptive distribution technique that delays the final distribution method decision until the execution time and is based on the size of the result set. A new plan step called `STATISTICS COLLECTOR` is put in front of the new hybrid hash distribution, it counts the number of rows returned from the PX servers and checks the count against a maximum threshold. If the threshold is reached then it is more cost effective to distribute the rows based on a hash distribution. If the size of the complete result set is below the threshold then the data is broadcasted.

Figure 17 shows an execution plan that uses the hybrid hash distribution method.

Id	Operation	Name	Rows	Bytes	Cost (%CPU)	Time	Pstart	Pstop	TQ	IN-OUT	PQ Dist
0	SELECT STATEMENT		960	27840	5 (20)	00:00:01					
1	PX COORDINATOR										
2	PX SEND QC (RANDOM)	:TQ10002	960	27840	5 (20)	00:00:01			Q1,02	P->S	QC (F
* 3	HASH JOIN BUFFERED		960	27840	5 (20)	00:00:01			Q1,02	PCWP	
4	PX RECEIVE		630	12600	2 (0)	00:00:01			Q1,02	PCWP	
5	PX SEND HYBRID HASH	:TQ10000	630	12600	2 (0)	00:00:01			Q1,00	P->P	HYBR1
6	STATISTICS COLLECTOR								Q1,00	PCWC	
7	PX BLOCK ITERATOR		630	12600	2 (0)	00:00:01			Q1,00	PCWC	
8	TABLE ACCESS FULL	CUSTOMERS	630	12600	2 (0)	00:00:01			Q1,00	PCWP	
9	PX RECEIVE		960	8640	2 (0)	00:00:01			Q1,02	PCWP	
10	PX SEND HYBRID HASH	:TQ10001	960	8640	2 (0)	00:00:01			Q1,01	P->P	HYBR1
11	PX BLOCK ITERATOR		960	8640	2 (0)	00:00:01	1	16	Q1,01	PCWC	
12	TABLE ACCESS FULL	SALES	960	8640	2 (0)	00:00:01	1	16	Q1,01	PCWP	

Figure 17: Execution plan for hybrid hash distribution

In the execution plan we can see the distribution method in line 5 and 10, PQ Distrib column shows HYBRID HASH and Operation column shows PX SEND HYBRID HASH. The new plan step STATISTICS COLLECTOR can be seen in line 6.

As a variation on the data distribution methods you may see a **LOCAL** suffix in a parallel execution plan on a Real Application Clusters (RAC) database. LOCAL distribution is an optimization for RAC environments and minimizes interconnect traffic for inter-node parallel queries. For example you may see a `BROADCAST LOCAL` distribution in an execution plan indicating that the row set is produced on the local node and only sent to the PX servers on that node.

PARALLEL PARTITION-WISE JOINS

Even if the database tries to choose the best distribution method depending on the optimizer statistics, distributing rows between PX sets requires inter-process or sometimes inter-instance communication. Techniques like full or partial partition-wise joins can minimize or even prevent data distribution, leading to potentially dramatic performance improvements in parallel execution.

If at least one of the tables accessed in the join is 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.

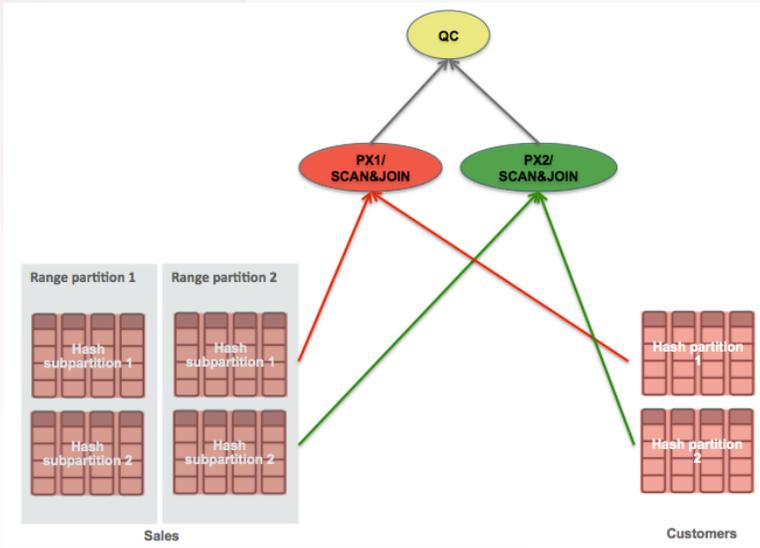


Figure 18: Full partition-wise joins do not require data distribution

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

Figure 18 shows the same join statement as in Figure 12, but this time the tables are equi-partitioned on the join column `cust_id`, in this example `CUSTOMERS` table is hash partitioned by the `cust_id` column whereas the `SALES` table is first range partitioned by a date column and then hash subpartitioned by the `cust_id` column. As shown in the figure PX1 reads all subpartitions for a range partition of the `SALES` table and then reads the equivalent hash partitions of the `CUSTOMERS` table (this is expressed as the `PARTITION RANGE ALL` iterator on top of the table scan for the `SALES` table in the execution plan in figure 19); the equi-partitioning of both tables on the join key guarantees that there will be no matching rows for the join outside of these partitions. The PX server will always be able to complete the full join by reading just these matching partitions. The same is true for PX2 too and for any equivalent partitions of these two tables. Note that partition-wise joins use partition-based granules rather than block-based granules. Compared to Figure 12 you can also see that partition-wise joins use a single PX server set rather than two.

Figure 19 shows the execution plan for this join statement.

Id	Operation	Name	Rows	Bytes	Cost (%CPU)	Time	Pstart	Pstop	TQ	IN-OUT	PQ Distrib
0	SELECT STATEMENT		816	59568	28 (4)	00:00:01					
1	PX COORDINATOR										
2	PX SEND QC (RANDOM)	:TQ10000	816	59568	28 (4)	00:00:01			01,00	P->S	QC (RAND)
3	PX PARTITION HASH ALL		816	59568	28 (4)	00:00:01	1	32	01,00	PCWC	
4	HASH JOIN		816	59568	28 (4)	00:00:01			01,00	PCWP	
5	PX PARTITION RANGE ALL		816	21216	24 (0)	00:00:01	1	16	01,00	PCWC	
6	TABLE ACCESS FULL	SALES	816	21216	24 (0)	00:00:01	1	512	01,00	PCWP	
7	TABLE ACCESS FULL	CUSTOMERS	630	29610	3 (0)	00:00:01	1	32	01,00	PCWP	

Figure 19: Execution plan for partition-wise join

The Operation and PQ Distrib columns indicate no data distribution and the TQ column shows only one PX set is used.

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.

In-Memory Parallel Execution

Unlike the traditional parallel processing where data bypasses any shared cache and gets transferred directly into the memory (PGA) of PX server processes, in-memory parallel execution (in-memory PX) leverages shared memory cache (SGA) to store data for subsequent parallel processing. In-memory PX takes advantage of the ever-increasing memory of today's database servers; this is especially beneficial on large-scale cluster environments where the aggregated total amount of memory can be multiples of Terabytes even when an individual database server "only" holds tens or hundreds of Gigabytes of memory. With in-memory PX Oracle uses the aggregated memory cache of servers in a Real Application Clusters (RAC) environment to deterministically cache objects distributed across the memory of all nodes. This ensures that subsequent parallel queries will read data out of cache from all nodes that can speed up processing times enormously instead of reading data from storage.

With the Oracle Database In-Memory option, a columnar in-memory data store specifically designed for in-memory processing to enable real-time analysis of large volumes of data. Its compressed columnar format and optimized data processing algorithms for in-memory provide the most optimal in-memory processing possible. Leveraging Oracle's new Database In-Memory technology is the recommended method for in-memory processing.

Systems without the Database In-Memory option can also take advantage of in-memory PX but will not be able to use the in-memory compressed columnar storage and optimized in-memory algorithms. In this case the database is using the standard database buffer cache for in-memory PX, the same cache that is used for online transactional processing. Since they use the same cache there is some risk of 'competing' for the buffer cache between OLTP and parallel operations, to ensure parallel operations do not take up the whole cache the Oracle Database limits the percentage of the buffer cache that can be used by in-memory PX to 80%. Depending on the data volumes processed in parallel and the memory requirement of the OLTP applications the benefits of in-memory PX can be limited with this model. Most commonly systems with smaller data volumes and with more mixed (and changing) workload characteristics benefit from this approach.

To broaden the applicability of in-memory PX, both in terms of the size of objects eligible for in-memory processing and to provide optimized full-table-scan-aware caching, the Automatic Big Table Caching (ABTC) functionality reserves a dedicated portion of the buffer cache specifically for parallel in-memory processing.

With ABTC, a part of the database buffer cache is reserved to store large objects (or portions of them) in memory so that more queries can take advantage of in-memory parallel execution. The size of objects that are eligible to be cached can be up to three times larger than the available reserved memory. ABTC uses an optimized segment and temperature-based algorithm to ensure the most optimal usage of the available cache.

The setting of the two following parameters is necessary to enable in-memory PX with ABTC:

- `PARALLEL_DEGREE_POLICY`: has to be set to `AUTO` or `ADAPTIVE`
- `DB_BIG_TABLE_CACHE_PERCENT_TARGET`: specifies the percentage of the database buffer cache to be reserved for in-memory PX.

With in-memory PX and ABTC the database decides if the objects accessed by the statement should be cached in ABTC or not. An object can either be a table, an index, or in the case of partitioned objects one or more partitions. This decision is based on an advanced set of heuristics that include the size of an object, the frequency at which the object is accessed, and the size of ABTC. If the object meets these criteria in-memory processing will be enabled, and the accessed object will be cached in

ABTC. In case of a RAC environment with multiple nodes the object will be fragmented (broken up into pieces) and distributed to all participating nodes; each fragment will be deterministically mapped (affinitized) to a specific RAC node and stored in its ABTC cache. Fragments can be physical ranges of blocks of a table or in the case of partitioned objects individual partitions.

Once a fragment has been mapped all subsequent accesses of that fragment will happen on that node. If a subsequent parallel SQL statement that requires the same data is issued from any node in the cluster, the PX servers on the nodes where the data resides will access the data in its ABTC cache and return only the result to the node where the statement was issued; no data is moved between nodes via Cache Fusion.

If an object is not considered to be cached it will be accessed via direct path IO.

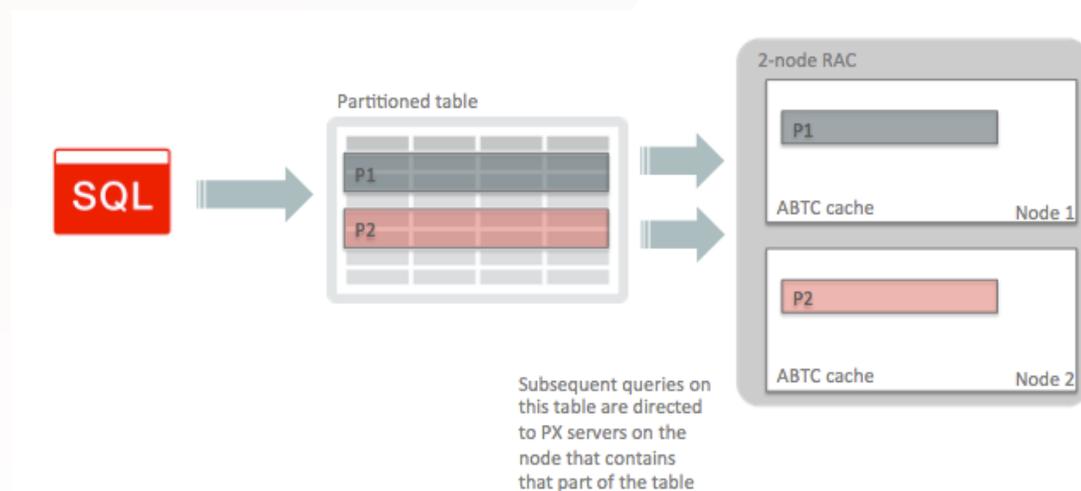


Figure 20: Sample data distribution for in-memory parallel execution for a partitioned table across a two node RAC cluster

CONTROLLING PARALLEL EXECUTION

To use parallel execution in an efficient manner you need to consider how to enable it, how to specify the degree of parallelism (DOP) for statements and how to use it in concurrent environments where many parallel statements may be running at the same time.

Enabling parallel execution

By default the Oracle database is enabled for parallel execution for queries and DDL statements. For DML statements you need to enable it at the session level with an ALTER SESSION statement.

```
ALTER SESSION ENABLE PARALLEL DML;
```

Managing the degree of parallelism

Even though parallel execution is enabled the decision to run a statement in parallel or not depends on some other factors. You can give Oracle full control over choosing and managing the degree of parallelism using Oracle's Automatic Degree of Parallelism (Auto DOP) framework, or you can control the chosen degree of parallelism manually. Using Auto DOP is Oracle's recommended way to control parallel execution with Oracle Database.

AUTOMATIC DEGREE OF PARALLELISM (AUTO DOP)

With Auto DOP the database automatically decides if a statement should execute in parallel or not and what DOP to use. The decision to use parallel execution and the DOP chosen are based on the resource requirements (a.k.a. "cost") of a statement. If the estimated elapsed time for the statement is less than `PARALLEL_MIN_TIME_THRESHOLD` (default is `AUTO`, equivalent to 10 seconds) the statement will run serially.

If the estimated elapsed time is greater than `PARALLEL_MIN_TIME_THRESHOLD` the Optimizer uses the cost of all operations (full table scan, index fast full scan, aggregations, joins, and so on) in the execution plan to determine an **ideal DOP** for the statement. Oracle Database uses the CPU and IO costs of all operations

Depending on the cost of a statement the ideal DOP can become very large. To ensure that you will not allocate too many parallel execution servers for a single statement the Optimizer will cap the actual DOP used. This cap is set by the parameter `PARALLEL_DEGREE_LIMIT`. The default value for this parameter is `CPU`, which means the maximum DOP is limited by the default DOP of the system. The formula used to derive the default DOP is:

```
PARALLEL_THREADS_PER_CPU * SUM(CPU_COUNT across all cluster nodes)
```

The optimizer will compare its ideal DOP with `PARALLEL_DEGREE_LIMIT` and take the lower value.

```
ACTUAL DOP = MIN(IDEAL DOP, PARALLEL_DEGREE_LIMIT)
```

Setting `PARALLEL_DEGREE_LIMIT` to a specific number controls the maximum DOP that will be used system-wide by Auto DOP. For a more fine-grained control of different user groups or applications, Oracle Database Resource Manager (DBRM) allows to set different DOP limits for individual resource consumer groups. It is recommended to use Database Resource Manager for fine-grained control of the maximum DOP in addition to a system-wide upper limit set by `PARALLEL_DEGREE_LIMIT`.

The following diagram shows conceptually how the decisions to parallelize a statement and what DOP to use are made system-wide, without Database Resource Manager in place. With Database Resource Manager, the computation of the ideal DOP will change to:

ACTUAL DOP = MIN(IDEAL DOP, PARALLEL_DEGREE_LIMIT, DBRM limit)

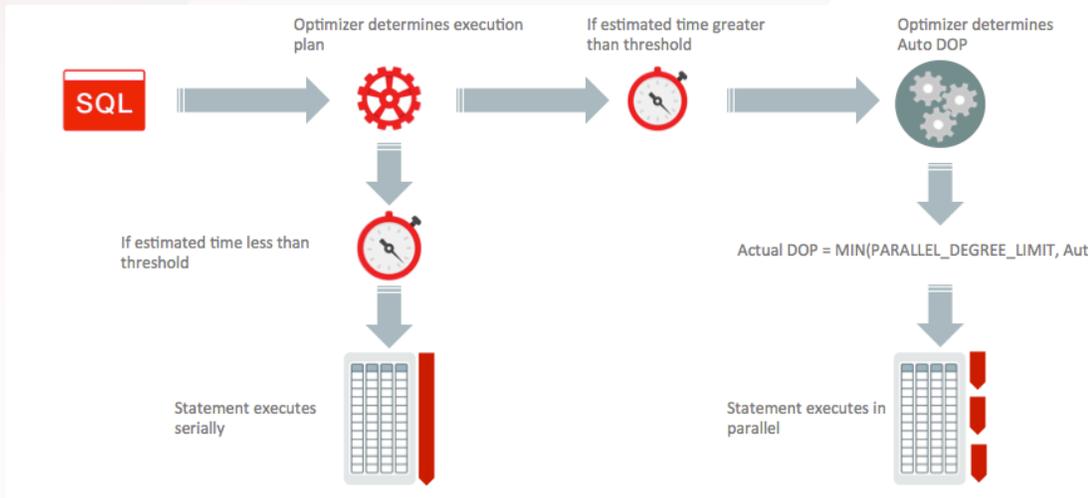


Figure 21: Conceptual decision path for Auto DOP without Database Resource Manager

The actual DOP selected is shown and explained in the note section of an execution plan. This information is provided for both statements explained with the explain plan command and executed statements (information stored in `V$SQL_PLAN`). For example the execution plan shown below was generated on a single instance database with `CPU_COUNT=32`, `PARALLEL_THREADS_PER_CPU=2`, and `PARALLEL_DEGREE_LIMIT=CPU`. In the note section, you will notice that a DOP of 64 has been selected. A degree of parallelism of 64 is the maximum DOP allowed by `PARALLEL_DEGREE_LIMIT` on this system ($2 * 32$).

Id	Operation	Name	Rows	Cost (%CPU)	Time	Pstart	Pstop	TQ	IN-OUT	PQ	Di
0	SELECT STATEMENT		1	1442 (8)	00:00:01						
1	SORT AGGREGATE		1								
2	PX COORDINATOR										
3	PX SEND QC (RANDOM)	:TQ10000	1					Q1,00	P->S	QC (R	
4	SORT AGGREGATE		1					Q1,00	PCWP		
5	PX BLOCK ITERATOR		1472M	1442 (8)	00:00:01	1	68	Q1,00	PCWC		
6	TABLE ACCESS STORAGE FULL	SALES	1472M	1442 (8)	00:00:01	1	68	Q1,00	PCWP		

Note

- automatic DOP: Computed Degree of Parallelism is 64 because of degree limit

Figure 22: Automatic Degree of Parallelism displayed in the notes section of the execution plan

Auto DOP is controlled by the initialization parameter `PARALLEL_DEGREE_POLICY` and enabled with a setting of either `LIMITED`, `AUTO`, or `ADAPTIVE`.

This initialization parameter can be applied on a system or session level. Furthermore, Auto DOP can be invoked for a specific SQL statement using the hint `PARALLEL` or `PARALLEL (AUTO)`:

```
SELECT /*+ parallel(auto) */ COUNT(*) FROM customers;
```

MANUALLY SETTING THE DEGREE OF PARALLELISM

With `PARALLEL_DEGREE_POLICY` set to `MANUAL` the functionality of Auto DOP is disabled and the end user has to manage the usage of parallel execution in the system. You can either request the so-called default DOP or a specific fixed value for the DOP on a session, statement, or object level.

DEFAULT parallelism

DEFAULT parallelism uses a formula to determine the DOP based on the initialization parameters. It is calculated as `PARALLEL_THREADS_PER_CPU * CPU_COUNT` in single instance databases and as `PARALLEL_THREADS_PER_CPU * SUM(CPU_COUNT)` in RAC environments. So, on a four node cluster with each node having `CPU_COUNT=8` and `PARALLEL_THREADS_PER_CPU=2`, the default DOP would be $2 * 8 * 4 = 64$.

You can use one of the following ways to get DEFAULT parallelism.

1. Set the object's parallel clause.

```
ALTER TABLE customers PARALLEL;
```

2. Use the statement level hint `parallel(default)`.

```
SELECT /*+ parallel(default) */ COUNT(*) FROM customers;
```

3. Use the object level hint `parallel(table_name, default)`.

```
SELECT /*+ parallel(customers, default) */ COUNT(*) FROM customers;
```

Note that the table setting shown above in #1 gives you the default DOP in manual mode, namely when `PARALLEL_DEGREE_POLICY` is set to `MANUAL`.

DEFAULT parallelism targets a single-user workload and is designed to use maximum resources assuming the operation will finish faster if you use more resources. The database does not check whether parallelism makes sense or whether parallelism will provide you any scalability. For example you can run a `SELECT * FROM emp`; with default parallelism on the system described before, but you will not see any scalability in returning these 14 records. In a multi-user environment DEFAULT parallelism will rapidly starve system resources leaving no available resources for other users to execute parallel statements concurrently.

Fixed Degree of Parallelism (DOP)

Unlike the DEFAULT parallelism, a specific DOP can be requested from the Oracle Database. You can use one of the following ways to get a fixed DOP.

1. Set a fixed DOP for the objects.

```
ALTER TABLE customers PARALLEL 8 ;
ALTER TABLE sales PARALLEL 16 ;
```

2. Use the statement level hint `parallel(integer)`.

```
SELECT /*+ parallel(8) */ COUNT(*) FROM customers;
```

3. Use the object level hint `parallel(table_name, integer)`.

```
SELECT /*+ parallel(customers, 8) */ COUNT(*) FROM customers;
```

Note that the table settings shown in #1 above only give you the fixed DOP in manual mode and limited mode, namely when `PARALLEL_DEGREE_POLICY` is set to `MANUAL` or `LIMITED`. In AutoDOP mode (`AUTO` or `ADAPTIVE`) any table decoration will be ignored.

Also note that for the example settings in #1, Oracle will choose the requested DOP as follows:

- Queries accessing just the `CUSTOMERS` table use a requested DOP of 8.
- Queries accessing the `SALES` table will request a DOP of 16.
- Queries accessing both the `SALES` and the `CUSTOMERS` table will be processed with a DOP of 16. Oracle is using the higher DOP².

The number of allocated PX servers can always become twice the requested DOP in case the parallel processing requires two PX server sets for producer/consumer processing.

Managing the concurrency of parallel operations

Regardless of your expected workload pattern you want to ensure that Oracle's parallel execution capabilities are used most optimally for your environment. This implies three fundamental tasks in addition to controlling the degree of parallelism:

1. Ensure that the system does not get overloaded by parallel processing
2. Ensure that any given statement will get the parallel resources required
3. Adhere to the potential different priorities for different user groups.

Oracle's Auto DOP framework not only controls the usage of parallel processing holistically without any user intervention, it also addresses the first two requirements. Together with Database Resource Manager, which is responsible for the third requirement, Oracle provides a comprehensive workload management framework to tackle the world's most complex mixed workload requirements.

MANAGING THE NUMBER OF PARALLEL EXECUTION (PX) SERVER PROCESSES

The Oracle Database allocates processes to parallel operations from a pool of PX server processes. The maximum number of PX server processes in this pool is set by the parameter `PARALLEL_MAX_SERVERS`. This is a hard limit used to prevent overloading the system with too many processes. By default, this parameter is set to

```
5 * concurrent_parallel_users * CPU_COUNT * PARALLEL_THREADS_PER_CPU
```

The value for `concurrent_parallel_users` is calculated as follows:

- If `MEMORY_TARGET` or `SGA_TARGET` initialization parameter is set, then the number of `concurrent_parallel_users` = 4.
- If neither `MEMORY_TARGET` or `SGA_TARGET` is set, then `PGA_AGGREGATE_TARGET` is examined. If a value is set for `PGA_AGGREGATE_TARGET`, then `concurrent_parallel_users` = 2. If a value is not set for `PGA_AGGREGATE_TARGET`, then `concurrent_parallel_users` = 1.

When all of the processes in the pool are allocated, new operations requiring parallelism are executed serially or with a downgraded DOP causing degraded performance for these operations.

To prevent reaching `PARALLEL_MAX_SERVERS` and serializing or downgrading operations, Auto DOP uses an additional limit set by the parameter `PARALLEL_SERVERS_TARGET`.

² Some statements do not fall under this rule, such as a parallel `CREATE TABLE AS SELECT`; a discussion of these exceptions is beyond the scope of this paper.

By default, this parameter is set to:

```
2 * concurrent_parallel_users * CPU_COUNT * PARALLEL_THREADS_PER_CPU
```

`PARALLEL_SERVERS_TARGET` is the number of PX server processes available to run parallel statements before statement queuing will be used. It is set lower than `PARALLEL_MAX_SERVERS` to ensure each parallel statement will get all of the PX server resources required and to prevent overloading the system with PX servers. All serial (non-parallel) statements will execute immediately even if statement queuing has been activated. The lower limit of `PARALLEL_SERVERS_TARGET` is only taken into account when running with `PARALLEL_DEGREE_POLICY` set to `AUTO` or `ADAPTIVE`, the mandatory initialization setting to invoke the full functionality of Auto DOP.

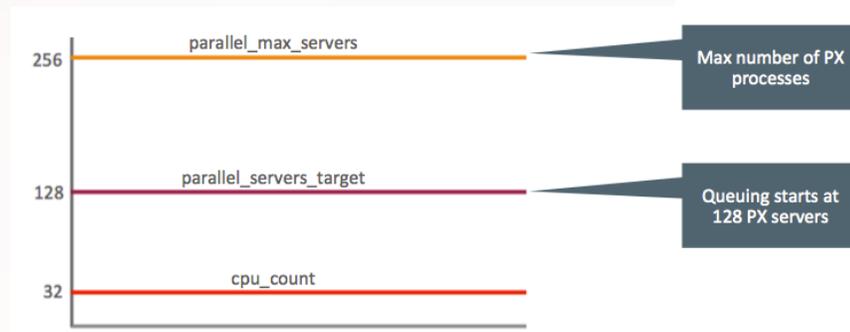


Figure 23: A sample configuration showing the limits for the number of PX server processes

MANAGING CONCURRENT PARALLEL PROCESSING WITH STATEMENT QUEUING

Once a SQL statement starts executing with a given DOP it will not change the DOP throughout the execution. As a consequence, if you start with a low DOP – for example; if there were not enough PX servers available - it may take longer than anticipated to complete the execution of the SQL statement.

With statement queuing, Oracle will queue SQL statements that require more parallel resources than currently available. Once the necessary resources become available, the SQL statement will be dequeued and allowed to execute. By queuing statements rather than allowing them to execute with a lower DOP or even serially, Oracle guarantees that any statement will execute with the requested DOP.

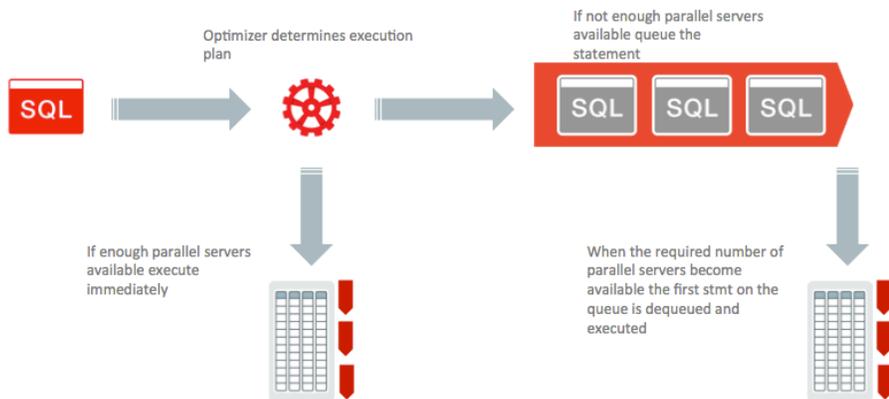


Figure 24: How statement queuing works

The statement queue is a First In - First Out (FIFO) queue based on the time a statement was issued. Statement queuing will kick-in once the number of parallel server processes active on the system is equal to or greater than `PARALLEL_SERVERS_TARGET`³.

Note that the main goal of parallel statement queuing is not to have statements in the queue “forever”, but to ensure to queue statements only for some short period of time until the requested amount of resources becomes available. If you happen to have statements in a queue for a long time you either have to revisit your maximum allowed DOP (`PARALLEL_DEGREE_LIMIT`) to allow a higher concurrency or your system is undersized for your workload.

You can identify which SQL statements are being queued using `[GV|V]$_SQL_MONITOR` or the SQL Monitor screens in Oracle Enterprise Manager (EM).

```
SELECT sql_id, sql_text
FROM [GV|V]$_SQL_MONITOR
WHERE status='QUEUED';
```

There is also a wait event to help identify if a statement has been queued. The wait event `resmgr:pq` queued indicates that the session is waiting in the queue.

Statements using the hint `NO_STATEMENT_QUEUING` and the statements executed by users having the Database Resource Manager directive `PARALLEL_STATEMENT_CRITICAL` set to `BYPASS_QUEUE` are allowed to bypass the statement queue and are executed immediately.

Statement queuing is only active when the parameter `PARALLEL_DEGREE_POLICY` is set to `AUTO` or `ADAPTIVE`.

MANAGING CONCURRENT PARALLEL PROCESSING WITH DATABASE RESOURCE MANAGER

Oracle Database Resource Manager (DBRM) enables you to prioritize work within an Oracle Database and restrict access to resources for certain groups of users. It is highly recommended to use DBRM if you are using parallel execution in concurrent environments. With DBRM you control the DOP limit and the number of PX servers an individual statement or the whole consumer group can use, prioritize parallel statements across consumer groups so that high priority requests are given more PX resources, and use different queues for different users so that high priority requests do not get queued behind low priority requests (rather than having a single queue for all users you have separate queues for different groups of users by using the Oracle Database Resource Manager starting with 11.2.0.2).

An in-depth discussion of how to use DBRM in conjunction with parallel execution is beyond the scope of this paper.

³ With Oracle Database Resource Manager, each consumer group has its own parallel statement queue and a percentage of the overall amount of `parallel_servers_target`.

INITIALIZATION PARAMETERS CONTROLLING PARALLEL EXECUTION

Several initialization parameters can be used to control or govern the parallel execution behavior in the database. However, you really only need to worry about the most fundamental ones to control parallel execution for an Oracle database:

PARALLEL_DEGREE_POLICY: controls whether or not In-Memory PX, Auto DOP and statement queuing are enabled. For backwards compatibility reasons the default value is `MANUAL`, which disables these features.

When set to `LIMITED` only Auto DOP will be enabled. In-Memory PX and statement queuing are disabled. Auto DOP is only applied to statements that access tables or indexes decorated with the `PARALLEL` clause but without specifying an explicit DOP as table attribute; tables and indexes that have a specific DOP specified will use that specified DOP.

When set to `AUTO` or `ADAPTIVE`, Auto DOP, statement queuing, and In-Memory PX are all enabled. Auto DOP will be applied to all SQL statements regardless of whether or not they access objects that have been explicitly decorated with a `PARALLEL` clause.

It is recommended to set this parameter to `AUTO`.

PARALLEL_SERVERS_TARGET: This parameter specifies the number of parallel server processes allowed to run before statement queuing will be used and is only active when the parameter `PARALLEL_DEGREE_POLICY` is set to `AUTO` or `ADAPTIVE`, Oracle will queue SQL statements that require parallel execution if the required number of PX server processes are not available. Statement queuing will begin once the number of PX server processes active on the system is equal to or greater than `PARALLEL_SERVERS_TARGET`. The default for this parameter is 40% of `PARALLEL_MAX_SERVERS`, ensuring some resource buffer for other operations, such as parallel statements bypassing the statement queue.

It is recommended to set this parameter to a value that is necessary for your parallel operations managed by Auto DOP under consideration of the allocated system resources available for parallel processing.

PARALLEL_DEGREE_LIMIT: With automatic degree of parallelism, Oracle automatically decides whether a statement should execute in parallel and what degree of parallelism the statement should use. The optimizer automatically determines the degree of parallelism for a statement based on the resource requirements of the statement. However, the optimizer will limit the degree of parallelism used to ensure parallel server processes do not flood the system. This limit is enforced by `PARALLEL_DEGREE_LIMIT`. By default, this parameter is set to `CPU` which limits the maximum DOP to the default DOP of the system. It is recommended to leave this parameter as default and use Database Resource Manager for fine-grained control of the maximum DOP.

The table below gives a brief explanation of the other parallel execution initialization parameters.

Summary Of OTHER INITIALIZATION PARAMETERS

Parameter name	Default value	Recommended value	Explanation
PARALLEL_MIN_SERVERS	CPU_COUNT * PARALLEL_THREADS_PER_CPU * 2	Default. Consider increasing to the minimum number of PX servers that are continuously in use.	Defines the minimum number of parallel execution servers that will be allowed by the database instance all the time.
PARALLEL_MAX_SERVERS	PARALLEL_THREADS_PER_CPU * CPU_COUNT * concurrent_parallel_users * 5	Default	Defines the maximum number of parallel execution servers a database instance can allocate. This is a hard limit and cannot be exceeded.
PARALLEL_ADAPTIVE_MULTI_USER	TRUE	FALSE	Throttles DOP of a statement based on concurrent work load. Can lead to non-deterministic response times.
PARALLEL_FORCE_LOCAL	FALSE	Default	In a RAC environment controls if parallel server processes will be allowed to the node the statement is issued on or not.
PARALLEL_MIN_PERCENT	0	Default	Minimum percentage of the requested number of parallel execution processes required for parallel execution.
PARALLEL_MIN_TIME_THRESHOLD	AUTO	Default	Minimum execution time a statement should have before AUTO DOP kicks in. Default 10 seconds.
PARALLEL_THREADS_PER_CPU	2	1 for platforms with hyperthreading enabled, 2 for other platforms	Number of parallel processes that a CPU can handle during parallel execution.
PARALLEL_EXECUTION_MESSAGE_SIZE	16KB	Default.	Size of the buffers used by the parallel server processes to communicate each other and the QC.

CONCLUSION

Parallel execution reduces the response time of database operations significantly by using multiple system resources. Resource availability is the most important prerequisite for scalable parallel execution.

This paper provides a detailed insight into how parallel execution works in the Oracle Database and provides recommendations on how to enable and use parallel execution successfully.

The Oracle Database provides a powerful SQL parallel execution engine that can parallelize any SQL-based operation – DDL, DML and queries. Oracle's Auto DOP together with Database Resource Manager provides a comprehensive workload management framework to tackle the world's most complex mixed workload requirements for any kind of database application.

ORACLE CORPORATION

Worldwide Headquarters

500 Oracle Parkway, Redwood Shores, CA 94065 USA

Worldwide Inquiries

TELE + 1.650.506.7000 + 1.800.ORACLE1

FAX + 1.650.506.7200

oracle.com

CONNECT WITH US

Call +1.800.ORACLE1 or visit oracle.com. Outside North America, find your local office at oracle.com/contact.

 blogs.oracle.com/oracle

 facebook.com/oracle

 twitter.com/oracle

Integrated Cloud Applications & Platform Services

Copyright © 2019, Oracle and/or its affiliates. All rights reserved. This document is provided for information purposes only, and the contents hereof are subject to change without notice. This document is not warranted to be error-free, nor subject to any other warranties or conditions, whether expressed orally or implied in law, including implied warranties and conditions of merchantability or fitness for a particular purpose. We specifically disclaim any liability with respect to this document, and no contractual obligations are formed either directly or indirectly by this document. This document may not be reproduced or transmitted in any form or by any means, electronic or mechanical, for any purpose, without our prior written permission.

Oracle and Java are registered trademarks of Oracle and/or its affiliates. Other names may be trademarks of their respective owners.

Intel and Intel Xeon are trademarks or registered trademarks of Intel Corporation. All SPARC trademarks are used under license and are trademarks or registered trademarks of SPARC International, Inc. AMD, Opteron, the AMD logo, and the AMD Opteron logo are trademarks or registered trademarks of Advanced Micro Devices. UNIX is a registered trademark of The Open Group. 0219

White Paper **Parallel Execution with Oracle Database**

February 2019/February 2019



Oracle is committed to developing practices and products that help protect the environment

ORACLE®