ORACLE UNIVERSITY

Parallel Processing in Oracle Database 12*c*

Student Guide
D87438GC10
Edition 1.0 | March 2015 | D88744

Learn more from Oracle University at **oracle.com/education/**

ORACLE

Author

Mark E Fuller

Technical Contributors and Reviewers

Yasin Baskan
Joel Goodman
Domininque Jeunot
Gwen Lazenby
Yio Liong Liew
Brian Pottle

Editors

Raj Kumar
Anwesha Ray

Graphic Designer

Rajiv Chandrabhanu

Publishers

Syed Ali
Joseph Fernandez

Contents

1

Introduction

Objectives

After completing this lesson, you should be able to:

- Discuss the goals of the seminar
- Describe the flow and topics covered during the seminar
- View the tentative schedule for the day

Seminar Goals

- Understand the basic concepts and describe the primary components of parallel processing
- List operations that benefit from parallel processing
- Use manual degree of parallelism (DOP) with different SQL statements and read the execution plans
- Use Auto DOP and statement-queuing
- Understand and use In-Memory Parallel Execution
- Apply parallel execution to data loading
- Troubleshoot parallel processing issues
- Use Database Resource Manager (DBRM) to complement the control of parallel processing usage

ORACLE

After completing this seminar, you should be able to apply parallel execution in order to realize the benefits of parallelizing requests in an Oracle Database 12*c* Release 1 (12.1).

We will cover all of the basics of parallel execution in Oracle Database 12*c*, also focusing on the new features of parallelization in Oracle Database 12*c* Release 1 (12.1).

This seminar explains:

- Why and when parallel processing is relevant
- When parallel processing is most efficient to use
- How to control and use parallelization.

Some specific configurations such as Real Application Clusters (RAC) database and Database Machine are reviewed in the seminar.

Prerequisites and Related Courses

- Prerequisites:
  - *D78846GC10 Oracle Database 12c: Administration Workshop I*
  - *D79995GC10 Oracle Database 12c: SQL Tuning for Developers*
  - Knowledge of Oracle Database administration
  - Ability to read and understand execution plans
- Related Courses:
  - *D79991GC10 Oracle Database 12c: Analytic SQL for Data Warehouseing*
  - *D79236GC10 Oracle Database 12c: Performance Management and Tuning*
  - *D77758GC10 Oracle Database 12c: New Features for Administrators*

ORACLE

As shown in the slide, students are expected to have some knowledge of the database administration and SQL tuning. The two courses shown as prerequisites cover this knowledge requirement.

There are also three other courses that cover related subjects.

Seminar Agenda

Morning:

1. Introduction
2. Parallel Processing Concepts
3. Basics of Parallel Execution
4. Manual DOP Management
5. Simplified Auto DOP

Afternoon:

6. Statement Queuing
7. In-Memory Parallel Execution
8. Parallel Execution and Data Loading
9. Diagnose, Troubleshoot, & Trace Parallel Execution

ORACLE

In addition to the lesson content, some recorded demos are available to provide participants with the opportunity to see how parallelization works in a standard environment and also with specific configurations.

The recorded demonstrations are posted on the Oracle Learning Library (OLL) for access by students after the seminar. The recorded demonstrations were made for Oracle Database 11*g*, but the functionality and features in those videos are still applicable for Oracle Database 12*c*.

Summary

In this lesson, you should have learned about the:

- Goals of the seminar
- Flow and topics covered during the seminar
- Tentative schedule for the day

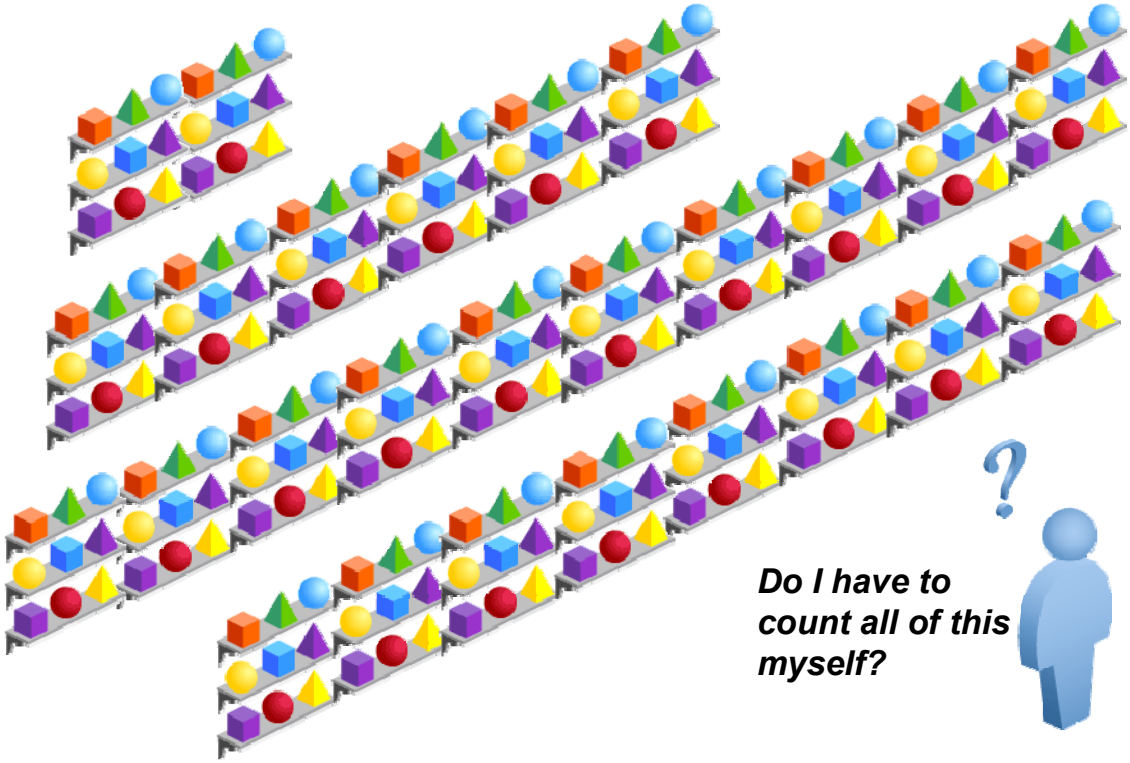
2 Parallel Processing Concepts

Objectives

After completing this lesson, you should be able to:

- Explain what parallel processing is, and why it is useful
- Define basic parallel processing terms
- Discuss when parallel processing is most effective

Why Use Parallel Processing?



Parallelism, also known as parallel processing or parallel execution, is a commonly used method of speeding up operations by splitting a task in smaller subtasks.

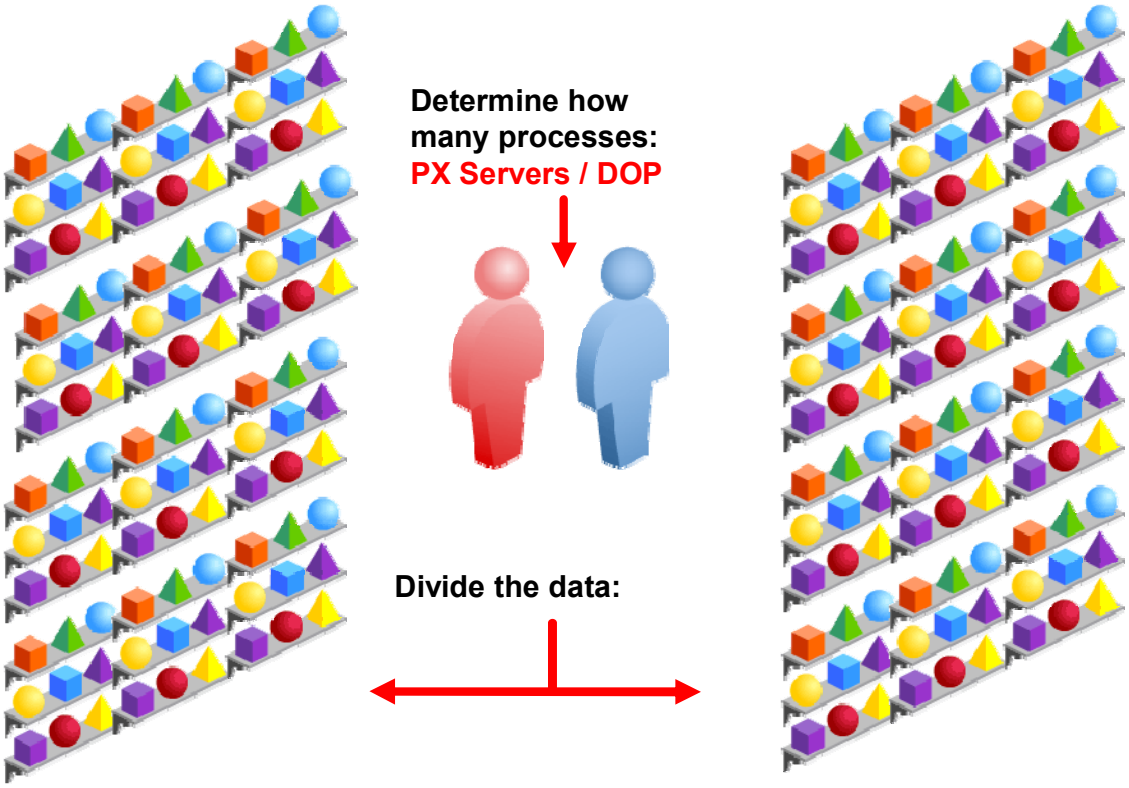
Imagine that you have been given the task of counting the number of items stacked on shelves in a store. There are two ways to do this:

- You can go up and down each aisle in the store and count the number of items by yourself.
- You can recruit one or more friends, divide up the store into sections, and each of you can count your section at the same time. When finished, you meet and add the results of all counts to complete the task.

Assuming that you all count at the same speed, if you have two counters, you expect to complete the task of counting all items in roughly half the time. If you have four counters, then you might complete the task four times as fast, and so on.

The database is not very different from this counting example. If you allocate more resources and achieve a processing time that is faster than the original time, then the operation scales up in terms of performance.

Basic Theory Behind Parallel Processing



In the counting example, certain assumptions are made to achieve better performance. These assumptions reflect some of the theory behind parallel processing.

Parallel Execution (PX) Servers / Degree of Parallelism (DOP)

First, you determine how many should do the counting. With the database, these workers are called *parallel execution (PX)* servers. The number of PX servers used by one parallel operation is called the *degree of parallelism (DOP)*.

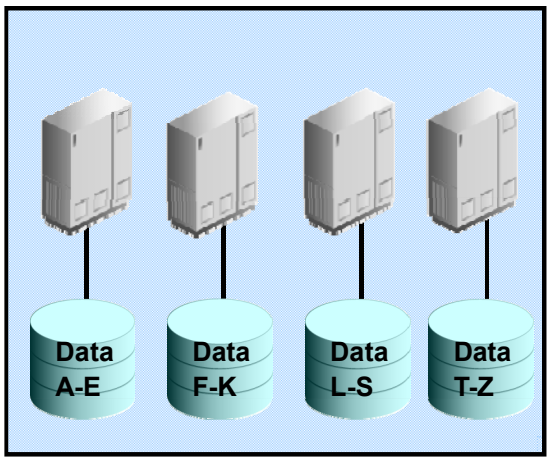
- In general, the bigger the workload, the more processes you could use.
- However, you must decide whether the overhead of having to both count and coordinate is worth the effort. For example, if the store has only 10 items on one shelf, it would take longer to decide who starts where than it takes to count the items.

Therefore, when applying parallel execution, you must decide how many parallel processes would be ideal to solve the problem fastest. In Oracle Database, the query optimizer can decide this based on the cost of the operation.

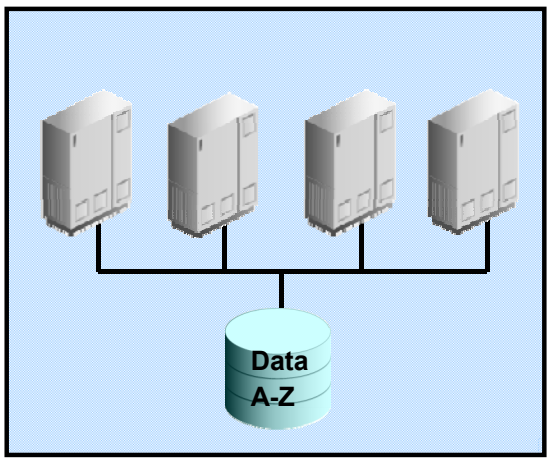
Dividing the Data

In the example, the work is divided into equal parts, and we assume each person counts at about the same speed. It is similar for parallel processing in a database. The data is divided into chunks of similar size, allowing them to be processed at the same time.

Understanding Data Partitioning Architectures



Shared Nothing
Static data partitioning is required.



Shared Everything
No data partitioning is required.

Partitioning of data is implemented in one of two basic ways, known as *shared-nothing* and *shared-everything* architectures. The main difference is whether physical (static) data partitioning is used as a foundation, and, therefore, required for parallelizing the work.

Shared-Nothing System

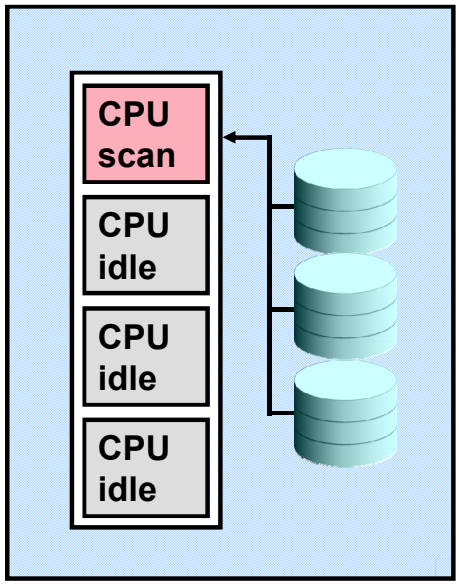
- Commonly known as a massively parallel processing (MPP) system, this architecture is divided into individual parallel processing units.
- Each processing unit has its own processing power and its own storage disk. Each CPU core is solely responsible for the individual data set on its own disks. To access a specific piece of data, it must use the processing unit that owns the subset of data.
- The partitioning strategy has to be decided upon initial creation of the system.

Shared-Everything System

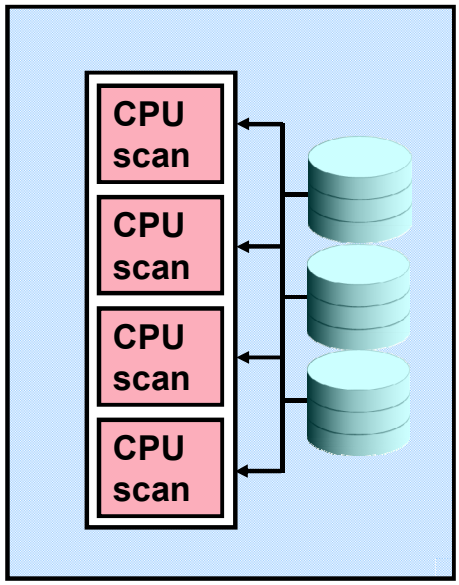
- This architecture does not require predefined data partitioning to enable parallelism.
- Parallelism is accomplished without the restrictions of fixed parallel access built in the data layout. Almost all operations may be parallelized in various ways and degrees.

Note: The Oracle Database Real Application Clusters (RAC) feature relies on a shared-everything architecture. When combined with Oracle Partitioning, Oracle Database can deliver exactly the same parallel processing capabilities as a shared-nothing system.

Benefits of Parallel Processing



Server without parallelism



Server with parallelism:
Maximum gain with four processes

In summary, parallelism is the idea of breaking down a task so that, instead of one process doing all the work in a task, many processes do part of the work at the same time. The improvement in performance can be quite high.

The benefits of parallel execution include:

- Faster access to data by using multiple processes
- Maximized machine resource usage, reducing elapsed time in query processing
- Optimal use of hardware resources, increasing system scalability

Parallelism Benefits: Ideal Versus Real

Although parallel execution provides the promise of increased performance, a variety of factors can influence how much faster operations perform when adding extra processes over a single process. Some of these factors include:

- Some operations consume more system resources (CPU, I/O, Network [RAC interconnect traffic] , and so on).
- Not all parallel processes run simultaneously on different CPUs because various other processes on the system compete for CPU time.
- At times, none of the processes involved in SQL processing are active on any CPU while they wait for their time slice.
- Overconsumption of system resources can lead to wait states.

When to Use Parallel Processing

Property	Candidates for Serial	Candidates for PX
Response time	Subseconds to seconds	Seconds to hours
Data organization	Application	Subject, time
Activities	Processes	Analysis
Nature of data	Generally 30–60 days, transactional	Snapshots over time, derived data/aggregates
Size	Small to large	Large to very large
Data sources	Operational, internal	Operational, internal, external
Duplicated data	Normalized RDBMS	Denormalized RDBMS (ROLAP or MOLAP)

Candidates for Parallel Execution

Parallel execution can dramatically reduce response time for data-intensive operations, commonly found in the following:

- Databases that are associated with decision support systems (DSSs) and data warehouses
- Symmetric multiprocessing (SMP) systems
- Clustered systems
- Massively parallel systems

These types of systems can gain the largest performance benefits from parallel execution because statement processing can be split among many CPUs on a single database.

Candidates for Serial Execution

Some tasks are not well suited for parallel execution. When the overhead of utilizing parallel execution is relatively large compared to the overall execution time, these tasks are not good candidates for parallel execution. For example, many online transaction processing (OLTP) operations are relatively fast, completing in mere seconds or fractions of seconds when run serially.

Serial execution may be preferable over parallel execution when system performance bottlenecks exist such as CPU and/or network saturation.

Summary

In this lesson, you should have learned how to:

- Explain what parallel processing is, and why it is useful
- Define basic parallel processing terms
- Discuss when parallel processing is most effective

Basics of Parallel Execution

Objectives

After completing this lesson, you should be able to:

- Describe operations that can be parallelized
- Explain primary parallel execution theory and terminology
- Identify the elements of parallel execution plans
- Describe how certain initialization parameters impact parallel execution

Parallelizable Operations

- Access methods: Table scans, index fast full scans, partitioned index range scans
- Join methods: Nested loop, sort merge, hash, star transformation
- DML statements: `UPDATE, DELETE, INSERT…SELECT, MERGE`
- DDL statements: `CREATE TABLE AS SELECT, CREATE INDEX, REBUILD INDEX, REBUILD INDEX PARTITION, MOVE/SPLIT/COALESCE PARTITION`
- Parallel queries: Queries and subqueries in `SELECT` statements, plus query portion of DDL/DML statements
- Miscellaneous SQL operations: `GROUP BY, NOT IN, SELECT DISTINCT, UNION, UNION ALL, CUBE`, and so on
- Utilities: SQL*Loader, RMAN, Data Pump, and external tables

Parallel execution benefits systems with all of the following characteristics:

- Symmetric multiprocessors (SMPs), clusters, or massively parallel systems
- Sufficient I/O bandwidth
- Underutilized or intermittently used CPUs (for example, systems where CPU usage is typically less than 30%)
- Sufficient memory to support additional memory-intensive processes, such as sorting, hashing, and I/O buffers

If your system lacks any of these characteristics, parallel execution might not significantly improve performance. In fact, parallel execution may reduce system performance on overutilized systems or systems with small I/O bandwidth.

The benefits of parallel execution can be seen in DSS and data warehouse environments. OLTP systems can also benefit from parallel execution during batch processing such as data loading, and during schema maintenance operations such as creation of indexes. The average simple DML or `SELECT` statements that characterize OLTP applications would not experience any benefit from being executed in parallel.

Note: Several Oracle Database utilities can perform parallel operations using different parallel architectures regarding the names and types of processes used to perform the parallel operations.

Previewing Execution Plans

Here are the serial execution plans for two SQL statements:

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

```
-----------------------------------------------------------------------------
| Id  | Operation          | Name      | Rows  | Cost (%CPU)| Time     |
-----------------------------------------------------------------------------
|   0 | SELECT STATEMENT   |           |     1 |     5   (0)| 00:00:01 |
|   1 |  SORT AGGREGATE    |           |     1 |            |          |
|   2 |   TABLE ACCESS FULL| CUSTOMERS |   630 |     5   (0)| 00:00:01 |
-----------------------------------------------------------------------------
```

1

```
SELECT c.name, s.purchase_date, s.amount
FROM customers c, sales s
WHERE s.customer_id = c.id;
```

```
-------------------------------------------------------------------------------------------------
| Id  | Operation           | Name      | Rows  | Bytes | Cost (%CPU)| Time     | Pstart| Pstop |
-------------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT    |           |   960 | 27840 |    11  (10)| 00:00:01 |       |       |
|*  1 |  HASH JOIN          |           |   960 | 27840 |    11  (10)| 00:00:01 |       |       |
|   2 |   TABLE ACCESS FULL | CUSTOMERS |   630 | 12600 |     5   (0)| 00:00:01 |       |       |
|   3 |   PARTITION RANGE ALL|          |   960 |  8640 |     5   (0)| 00:00:01 |     1 |    16 |
|   4 |    TABLE ACCESS FULL| SALES     |   960 |  8640 |     5   (0)| 00:00:01 |     1 |    16 |
-------------------------------------------------------------------------------------------------
```

2

ORACLE

Before examining operations that may be run in parallel, we must discuss some of the basic theory and concepts that support parallel execution.

To begin, we will compare some simple serial and parallel execution plans to help illustrate some important concepts.

Previewing Execution Plans

When you execute a SQL statement in the Oracle Database, it is deconstructed into individual steps or row sources, which are identified as separate lines in an execution plan.

The slide shows two examples.

The first statement touches just one table, and a returns the total number of customers in the `CUSTOMERS` table.

The second statement includes a join between the `CUSTOMERS` and `SALES` tables, and results in a more complex serial execution plan.

Comparing Serial and Parallel Execution Plans

Here are some possible parallel execution plans for the same SQL statements:

1

Id	Operation	Name	Rows	Cost (%CPU)	Time	TQ	IN-OUT	PQ Distrib
0	SELECT STATEMENT		1	2 (0)	00:00:01			
1	SORT AGGREGATE		1					
2	PX COORDINATOR							
3	PX SEND QC (RANDOM)	:TQ10000	1			Q1,00	P->S	QC (RAND)
4	SORT AGGREGATE		1			Q1,00	PCWP	
5	PX BLOCK ITERATOR		630	2 (0)	00:00:01	Q1,00	PCWC	
6	TABLE ACCESS FULL	CUSTOMERS	630	2 (0)	00:00:01	Q1,00	PCWP	

2

Id	Operation	Name	Rows	Bytes	Cost (%CPU)	Time	Pstart	Pstop	TQ	IN-OUT	PQ Distrib
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 (RAND)
* 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	

ORACLE

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 reexecute the same two statements shown from the previous slide in parallel, we could get the execution plans that are shown in the slide.

Note: Parallel plans will look different in versions prior to Oracle Database 10*g*.

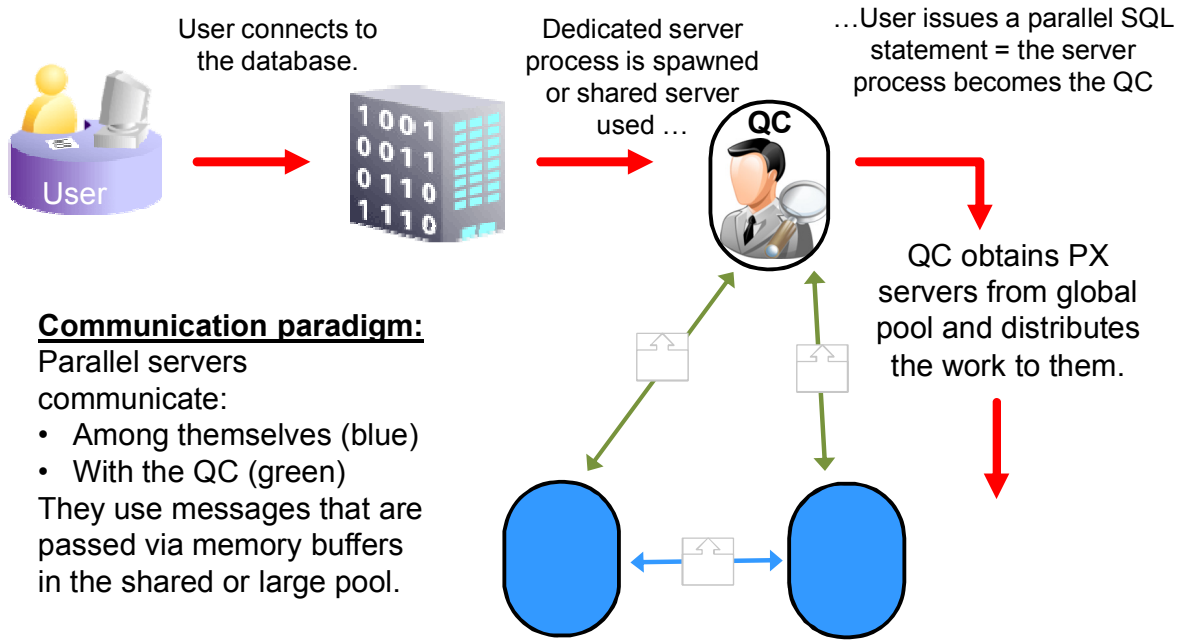
As you can see, parallel execution plans look quite a bit different than the serial plans, mainly because the database has additional logistical processing steps due to the parallel processing.

Although the execution plans look more complex, SQL parallel execution in the Oracle database is based on just a few fundamental concepts. These concepts, which are discussed in the next few slides, will help you understand:

- The parallel execution paradigm in the database
- The structure of parallel SQL execution plans

Note: Parallel execution not only does work in parallel or one step of a plan, but separate steps can work at the same time, offering a type of pipelining.

Understanding Parallel Execution: Query Coordinator (QC) and Parallel (PX) Servers



SQL parallel execution in the Oracle Database is based on the interaction of a session called the *query coordinator (QC)* and *parallel execution (PX) server* processes. To use the analogy between parallel execution and the example of people counting items on store shelves, there would be a third person (the QC) telling you and your friend (the PX servers) how to count the items.

Query Coordinator: The QC is the session that initiates the parallel SQL statement and distributes the work to the PX servers. In addition, it:

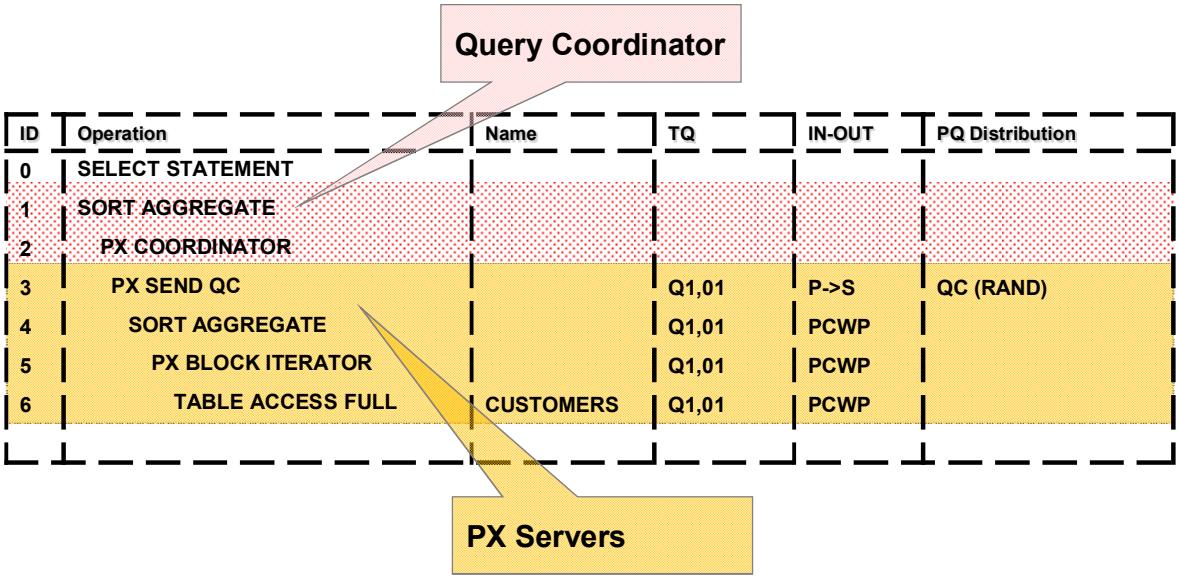
- Parses the query and determines the degree of parallelism (a concept covered later)
- Controls the query and sends instructions to the PX servers
- Determines the work ranges for the PX servers
- Produces the final output to the user
- Executes serial row sources

PX Servers: The PX servers are the individual sessions that perform work in parallel. They:

- Are taken from a pool of globally available PX server processes
- Are assigned to a given operation by the QC
- Communicate with each other and with the QC by using messages
- Execute some serial row sources

Parallel Execution Plans: Identifying the QC and PX Servers

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



The slide shows a possible parallel execution plan for the simple SQL query shown previously against the `CUSTOMERS` table.

Identifying the QC: The QC is easily identified in parallel execution plans as `'PX COORDINATOR'`—the operation with the ID 2 in the execution plan shown in the slide.

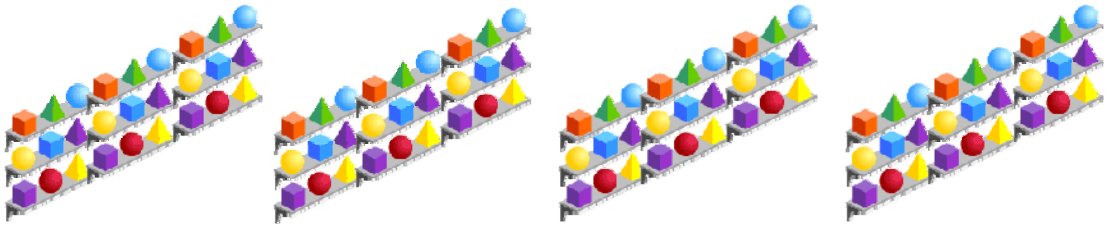
Identifying PX Servers: All the work shown below the QC entry in parallel execution plans is performed by the PX servers.

Understanding Operations: Continuing the counting analogy, the operations in the sample execution plan can be described like this:

- You and your friend count the items that have been divided into your sections. This is equivalent to the operations with the ID 4, ID 5, and ID 6, shown in the execution plan.
- The ID 5 operation is equivalent to the instruction from the QC to count only the items in your section.
- Each of you tells the QC your individual subtotals (ID 3). This is the handover from the PX servers back to the QC.
- In ID 1, the QC adds the final total for “assembly” of the result, which will be returned to the user process.

Understanding Parallel Execution: The Producer/Consumer Model

Continuing with the counting example, imagine that the job is to count the total number of items *per color*.



- Add a second set of counters: One set of workers counts items. The other set of workers counts colors.
- An item counter *produces* count information, distributes the counts based on the color, and a color counter *consumes* the information.
- In the end, the consumer (in this case, the set of color counters) returns the ultimate results to the QC.

ORACLE

Continuing even further with our counting analogy, consider the following:

- If the item counters each cover one half of the store, they each potentially see the same colors and get a subtotal for each color, but not the complete result for the entire store.
- They both could record this information and report it back to the QC. But the QC then has to sum up all of the results by himself. What if all items on a shelf were a different color? The QC would redo exactly the same work as you and your friend just did.

To parallelize the counting, another set of friends is brought in to help you out. The new set of color counters walk along with each of the item counters. Whenever you count a new item, you also tell the person that is in charge of that color about the new item.

In this way, the item counter *produces* the information, redistributes it based on the color identified, and the color counter *consumes* the information.

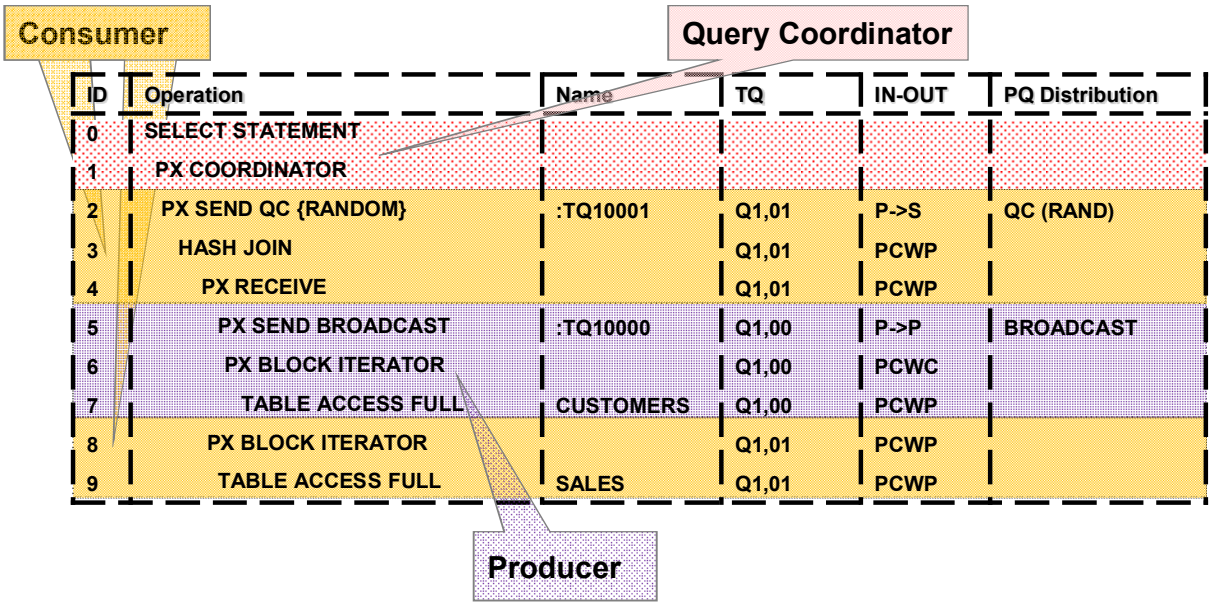
When finished, the color-counting friends tell their result to the person in charge (the QC). There are two sets of counters, each doing a part of the job, working in unison.

That is similar to how the database works:

- 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).

Parallel Execution Plans: Identifying the Producers and Consumers

```
SELECT /*+ monitor parallel(4) */ c.cust_last_name,s.time_id,s.quantity_sold
FROM oe.customers c, sh.sales s
WHERE s.cust_id = c.customer_id;
```



ORACLE

The producer/consumer model is illustrated by the parallel join between the SALES and CUSTOMERS tables, shown in the slide. Rows have to be redistributed based on the join key, ensuring that matching join keys from both tables are sent to the same PX server process that is performing the join. In this example:

- One set of PX servers reads and sends the data from the CUSTOMERS table (producer).
- The other set of PX servers receives the data (consumer) and joins it with the SALES table.

Table Queue (TQ) Column

In general, 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. In the example in the slide:

- The first PX set (Q1,00) reads the CUSTOMERS table and produces (sends) rows to the second PX set.
- The second PX set (Q1,01):
  - Reads from the SALES table
  - Consumes (receives) records from PX set 1 and performs the join
  - Sends the joined result to the QC

Note: Whenever data is distributed (from producers to consumers or consumers to the QC), you will also see an entry of the form :TQxxxxx (Table Queue x) in the NAME column.

The Producer/Consumer Model and the DOP

Impact on the Degree of Parallelism:

- DOP represents the number of PX servers associated with a *single* operation.
- The producer/consumer model expects two sets of PX servers for a parallel operation. Therefore, the number of PX servers processes may be twice the DOP.

Query examples: DOP is set to 4; PX Servers =

1
```
SELECT c.cust_last_name,s.time_id,s.quantity_sold
FROM oe.customers c, sh.sales s
WHERE s.cust_id = c.customer_id;
```
= 8

2
```
SELECT count(*) FROM customers c;
```
= 4

ORACLE

The producer/consumer model has a very important consequence for the number of PX server processes that are spawned for a given parallel operation: The model *expects two sets of PX servers for a parallel operation*.

Therefore, the number of PX server processes is twice the requested degree of parallelism (DOP).

General Rule

The only case when PX servers do not work in pairs is if the statement is so basic that one set of PX servers can complete the entire statement in parallel.

Examples

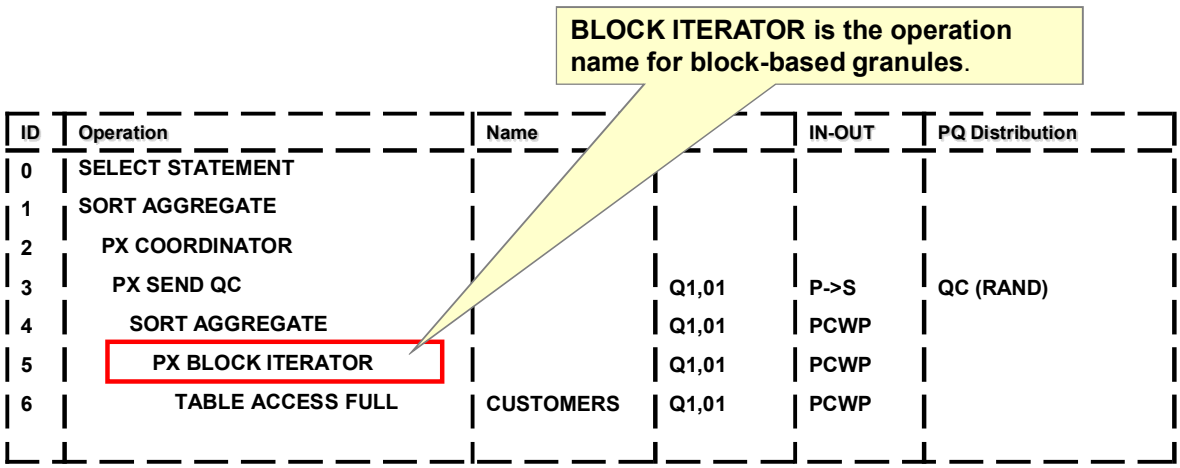
A DOP of 4 indicates the following:

1. The parallel join query shown in the slide requests 8 PX server processes in two sets: 4 producers and 4 consumers.
2. The simple select count(*) query shown in the slide requires only one set of 4 PX server processes.

Parallel Execution Plans: Understanding Granules

Granules are the:

- Smallest unit of work when accessing data
- Basic mechanism the Oracle Database uses to distribute work for parallel execution (block-based granules)



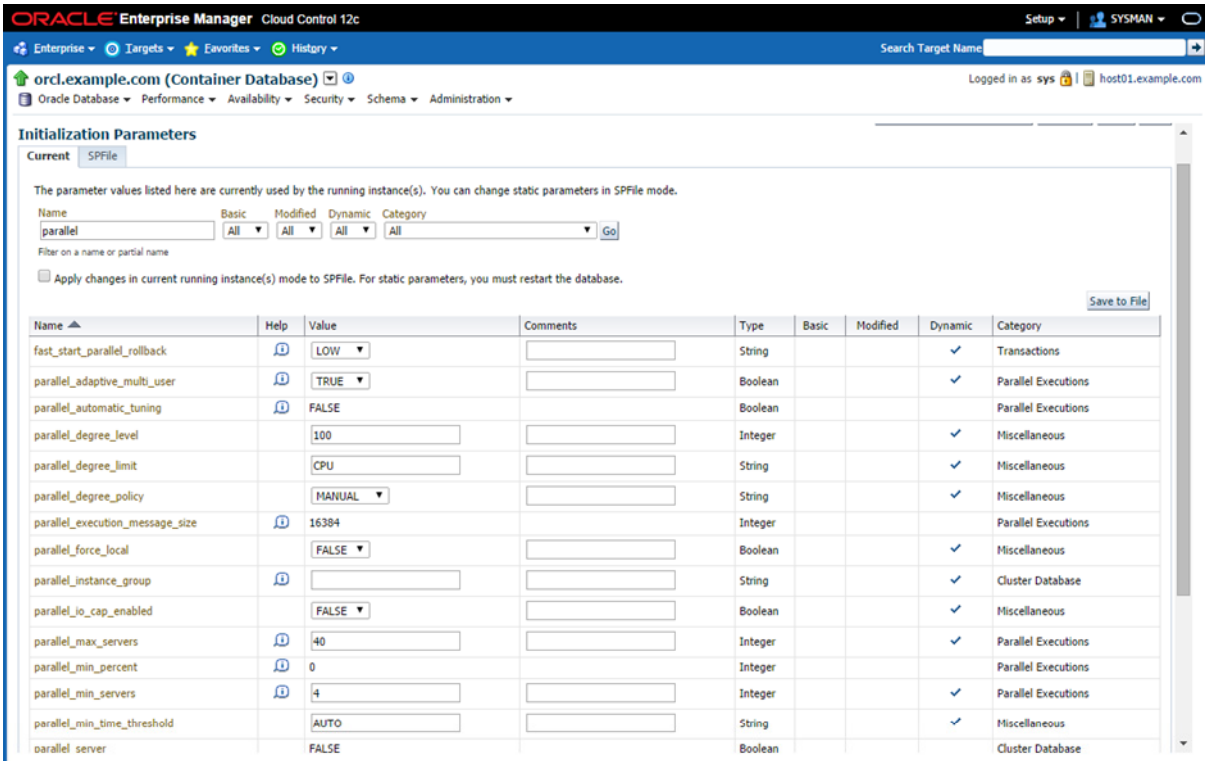
Because Oracle Database uses a shared-everything architecture, any CPU core in a configuration can access any piece of data from a storage perspective. This design enables the database to choose this smallest unit of work depending solely on a query's requirements.

In Oracle Database, the smallest unit of work when accessing data is called a *granule*.

The basic mechanism the Oracle Database uses to distribute work for parallel execution is block ranges on disk, also known as *block-based granules*. This methodology 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 distributed to PX servers to work on. When a PX server finishes the work for one granule, the next one is distributed.
- The number of granules is always much higher than the requested DOP so that an even distribution of work can be achieved among parallel server processes.
- The operation `PX BLOCK ITERATOR` shown in the execution plan in the slide is literally the iteration over all generated block-range granules.

Previewing Parameters for Parallel Execution



As is evident in this screenshot of Enterprise Manager Cloud Control 12*c*, there are many parameter settings that govern different aspects of parallel execution. You will learn to use many of these parameters in this seminar.

For now, only a few primary parameters are introduced in order to give you a general understanding of how these settings influence parallel execution. In addition, several parameters that are essential for parallel execution in Oracle 12*c* Release 1 are also introduced.

Example PX Parameters

Parameter	Default Value	Description
PARALLEL_MAX_SERVERS	(See description in notes page)	Limits the number of PX servers, except queries against GV$ views
PARALLEL_MIN_SERVERS	0	Specifies the number of PX servers that are always running
PARALLEL_EXECUTION_MESSAGE_SIZE	Operating System Dependent	Size of the buffers used for communication between PX server processes in a query
PARALLEL_ADAPTIVE_MULTI_USER	TRUE	TRUE improves performance in a multiuser environment. FALSE is used for batch processing.
PARALLEL_MIN_PERCENT	0	This parameter enables users to wait for an acceptable DOP, depending on the application in use.

ORACLE

PARALLEL_MAX_SERVERS

- The default on a single instance is DOP x (1,2, or 4) x 5 or more specifically:
  PARALLEL_THREADS_PER_CPU x CPU_COUNT x concurrent_parallel_users x 5
  In the formula, the value assigned to *concurrent_parallel_users* running at the default degree of parallelism on an instance depends on the memory management setting. If automatic memory management is disabled (manual mode), then the value of concurrent_parallel_users is 1. If PGA automatic memory management is enabled, then the value of concurrent_parallel_users is 2. If global memory management or SGA memory target is used in addition to PGA automatic memory management, then the value of concurrent_parallel_users is 4.
- When concurrent users have too many PX server processes, memory contention (paging), I/O contention, CPU bottlenecks, or excessive context switching can occur. This contention can reduce system throughput to a level lower than if parallel execution were not used. Increase the PARALLEL_MAX_SERVERS value only if the system has sufficient CPU, memory, and I/O bandwidth for the resulting load.
- You can use performance monitoring tools of the operating system to determine how much memory, swap space, and I/O bandwidth are free.

Note: The default value allows for 5 to 20 DOP sets of PX servers. Because many parallel operations use 2 DOP sets, this allows 2 to 10 concurrent users at full DOP.

PARALLEL_MIN_SERVERS

- It specifies the number of PX servers that are created at instance startup.
- A high number can cause a large chunk of memory to be allocated at database startup time for the PX message buffers.

PARALLEL_EXECUTION_MESSAGE_SIZE

On most platforms, the default value is `16384` bytes if `COMPATIBLE` is set to 11.2.0 or higher. Larger values result in better performance at the cost of higher memory use.

PARALLEL_ADAPTIVE_MULTI_USER

As you have learned, the DOP specifies the number of available processes used in a single parallel operation. Each parallel thread can use one or two query processes, depending on the query's complexity. The adaptive multiuser feature adjusts the DOP on the basis of user load.

For example, you might have a table with a DOP of 6. This DOP may be acceptable with 12 users.

However, if 12 more users enter the system and you enable `PARALLEL_ADAPTIVE_MULTI_USER`, the DOP is reduced to spread resources more evenly according to the perceived load. After the DOP for a query is determined, the DOP does not change for the duration of the query.

It is best to use the parallel adaptive multiuser feature when there is high probability of users processing simultaneous parallel execution operations.

- By default, `PARALLEL_ADAPTIVE_MULTI_USER` is set to `TRUE`, which optimizes the performance of systems with concurrent parallel SQL execution operations.
- If you set `PARALLEL_ADAPTIVE_MULTI_USER` to `FALSE`, each parallel SQL execution operation receives the requested number of parallel execution server processes regardless of the impact to the performance of the system as long as sufficient resources have been configured.

Unless you have a need to do otherwise, the best practice is *not* to change the default value for `PARALLEL_ADAPTIVE_MULTI_USER`.

PARALLEL_MIN_PERCENT

This parameter enables users to wait for an acceptable DOP, depending on the application in use. The recommended value for the `PARALLEL_MIN_PERCENT` parameter is 0 (zero).

Setting this parameter to values other than 0 (zero) causes Oracle Database to return an error when the requested DOP cannot be satisfied by the system at a given time.

For example, if you set `PARALLEL_MIN_PERCENT` to 50, which translates to 50 percent, and the DOP is reduced by 50 percent or greater because of the adaptive algorithm or because of a resource limitation, then Oracle Database returns ORA-12827.

```
ORA-12827: insufficient parallel query slaves available
```

PX Parameters for Auto DOP

Parameter	Default Value	Description
PARALLEL_DEGREE_LIMIT	"CPU"	Max DOP that can be granted with Auto DOP
PARALLEL_DEGREE_POLICY	"MANUAL"	Specifies if Auto DOP, Queuing, and In-Memory PE are enabled
PARALLEL_MIN_TIME_THRESHOLD	"AUTO"	Specifies min execution time of a statement before AUTO DOP becomes active
PARALLEL_SERVERS_TARGET	CPU_COUNT* PARALLEL_THREADS_ PER_CPU * *concurrent_parallel_users* * 2	Specifies number of parallel processes allowed to run parallel statements before statement queuing is used

Note: Auto DOP will be covered in the lesson titled "Simplified Auto DOP."

The Auto DOP feature is examined in detail in the lesson titled "Simplified Auto DOP," but these parameters are essential for taking advantage of this powerful feature.

The `PARALLEL_DEGREE_POLICY` parameter has an impact on the other parameters in the following way:

1. `PARALLEL_DEGREE_POLICY=MANUAL` — **PX Features: none**
   a. None of the parameters have any impact
2. `PARALLEL_DEGREE_POLICY=LIMITED` — **PX Features:**
   a. `PARALLEL_MIN_TIME_THRESHOLD=10s` — **- Auto DOP**
   b. `PARALLEL_DEGREE_LIMIT=CPU`
3. `PARALLEL_DEGREE_POLICY=AUTO` — **PX Features:**
   a. `PARALLEL_MIN_TIME_THRESHOLD=10s` — **- Auto DOP**
   b. `PARALLEL_DEGREE_LIMIT=CPU` — **- Queuing**
   c. `PARALLEL_SERVERS_TARGET=` — **- In-Memory**
   `CPU_COUNT*PARALLEL_THREADS_PER_CPU*`*`concurrent_parallel_users`*`*2`

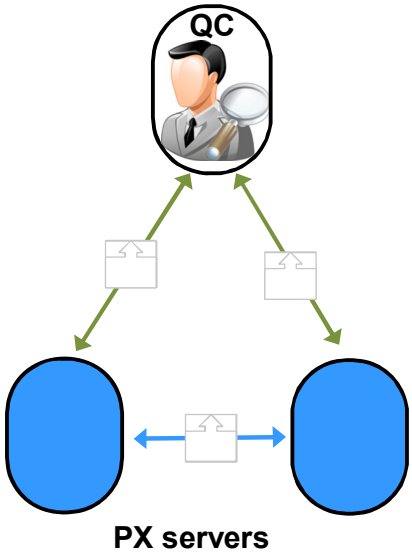
Parameters for PX Messaging: Shared and Large Pools

Managing memory for PX messages:

- SGA_TARGET
- MEMORY_TARGET

Communication paradigm:
Parallel servers communicate:
- **Among themselves (blue)**
- **With the QC (green)**

Using messages that are passed *via memory buffers in the shared or large pool*



ORACLE

As previously mentioned, the QC and PX Servers exchange data and messages with each other during parallel execution. These messages are passed by using memory buffers in either the shared pool or the large pool.

- By default, PX messages use the shared pool.
- If the large pool is configured (optional) with enough memory available, PX messages will use the large pool.

The SGA_TARGET and MEMORY_TARGET parameters have been added to the database since version 10*g*, allowing the SHARED_POOL_SIZE and LARGE_POOL_SIZE values to be dynamically resized by Oracle when necessary.

Note: If not enough is memory available for these messages, PX queries can fail, resulting in an ORA-4031 error.

Summary

In this lesson, you should have learned to:

- Describe operations that can be parallelized
- Explain primary parallel execution theory and terminology
- Identify the elements of parallel execution plans
- Describe how certain initialization parameters impact parallel execution

4

Manual DOP Management

Objectives

After completing this lesson, you should be able to:

- Define the prerequisites required to parallelize statement execution
- Explain a single parallel table scan
- Explain parallel execution with a `GROUP BY`
- Explain parallel execution with an `ORDER BY`
- Describe parallel join execution
- Describe parallel join execution with bloom filters
- Explain parallel execution and advantages with partition-wise join
- Describe bulk DML and parallel processing
- Describe DDL statements and parallel processing
- Explain how indexes rebuild in parallel

Manual Parallel Execution

Approaches to applying parallelism in earlier releases of Oracle Database:

- Intimate knowledge of SQL and the workload is required.
- The mechanisms are manual, and involve trial and error.
- There is no easy way to run a set of queries with different degrees of parallelism values.
  - Running every query with a single degree of parallelism value is suboptimal in terms of overall response time.
- DBAs tend to restrict the use of parallelism to very narrow and highly controlled workloads due to:
  - Difficulty in enabling parallelism
  - Systemwide impact of specifying an object's degree of parallelism

ORACLE

In earlier releases of Oracle Database, Oracle provided various mechanisms to enable parallelism for a given SQL command. All of these mechanisms are manual and required intimate knowledge of both SQL and the workload.

It is a burden for the DBA to decide on the degree of parallelism (DOP) of objects such as tables and indexes. This attribute is used to indicate the DOP of any query touching the objects. For a DBA, it is, therefore, very hard to settle on a good DOP that performs well in any circumstances for any queries on any object sizes. Therefore, this DOP is at best a compromise.

Because of the difficulty of enabling parallelism and the systemwide impact of specifying a DOP of an object (it affects all the statements touching it), the DBA might restrict the use of parallelism to very narrow and highly controlled workloads.

With automatic parallelism introduced with the Oracle Database 11*g* Release 2 version, the database server compensates for wrong or missing user settings for parallel execution, ensuring more optimal resource consumption and overall system behavior.

Note: This feature automatically determines the DOP for the parallel query execution client. It will be covered in the lesson titled "Simplified Auto DOP."

Degree of Parallelism

- The degree of parallelism (DOP) is the number of parallel execution (PX) servers used per PX set.
- A statement can have more than one PX set but at most two.
- Parallelism can be enabled or disabled at the session level.
- The DOP can be:
  - Set as an attribute of a table or index
  - Specified as a hint in a query statement
    - Object-level hint: PARALLEL (emp, n)
    - Statement-level hint: PARALLEL (n)

ORACLE

You can enable parallel processing during DML or DDL at the session level:

- This session attribute allows or prohibits the use of parallel processing only during the session duration. All subsequent DML (`INSERT, UPDATE, DELETE`), DDL (`CREATE, ALTER`), or query (`SELECT`) operations are executed either serially `(DISABLE)` or in parallel `(ENABLE)` after such a statement is issued. They will be executed serially `(DISABLE)` regardless of any parallel attribute associated with the table or indexes involved.

You can set the degree of parallelism (DOP) at either of the following levels:

- At the object level. The `DEGREE` attribute is stored in the data dictionary.
- During a statement execution. A hint defined at the statement level overrides the DOP defined at the object level.

When a statement is executed, the DOP requested is not necessarily the one provided. It depends on the resources available to satisfy the DOP requested.

The object-level hint is explained in this lesson, whereas the statement-level hint is explained in the lesson titled “Simplified Auto DOP,” which covers Auto DOP.

Prerequisites to Parallel Execution

You can enable or disable parallel execution at the session level.

```
ALTER SESSION ENABLE|DISABLE|FORCE PARALLEL
QUERY|DML|DDL;
```

The DOP can also be defined at the object level as an attribute.

A. Setting the DOP as an attribute for the tables or indexes:

```
CREATE TABLE tab_2 (c NUMBER, c2 date) PARALLEL 5;
```

B. Specifying a hint in the statement:

```
SELECT /*+ PARALLEL (4) */ * from hr.employees;
```

```
SELECT /*+ parallel(c) parallel(s) */ c.c1 a, sum(s.c) b
FROM customers c, sales s WHERE s.customer_id = c.id
AND s.purchase_date=to_date('01-JAN-2007','DD-MON-YYYY')
GROUP BY c.c1;
```

A DML statement can be parallelized only if you have explicitly enabled parallel DML in the session. This mode is required because parallel DML and serial DML have different locking, transaction, and disk space requirements and parallel DML is disabled for a session by default. When parallel DML is disabled, no DML is executed in parallel even if the `PARALLEL` hint is used.

The `CREATE TABLE` example shows the creation of a table with a default DOP defined at 5, stored as `DEGREE 5` in the data dictionary. The default DOP is valid for queries and the DML `INSERT, UPDATE, DELETE,` and `MERGE` after table creation. Any subsequent DML or queries on the table, for which parallelization is possible, will attempt to use parallel execution.

Enabling/Disabling Parallel Execution at Session Level

```
SQL> alter session disable parallel query;
Session altered.
```

```
SQL> explain plan for select * from hr.employees;
Explained.
```

```
SQL> select * from table(dbms_xplan.display);

PLAN_TABLE_OUTPUT
------------------------------------------------------------------------
Plan hash value: 2215152728
------------------------------------------------------------------------
| Id  | Operation         | Name      | Rows  | Bytes | Cost (%CPU)|
------------------------------------------------------------------------
|   0 | SELECT STATEMENT  |           |   294K|   864K|   139   (3)|
|   1 |  TABLE ACCESS FULL| EMPLOYEES |   294K|   864K|   139   (3)|
------------------------------------------------------------------------
```


The slide example shows that you can control parallel SQL execution for a session by using the `ALTER SESSION` statement to disable parallel query. Independent of the DEGREE setting for the `HR.EMPLOYEES` table, the statement runs in serial mode.

Declaring DOP for Tables

The degree of parallelism (DOP) can be defined or modified with the PARALLEL clause when:

- Creating or altering nonpartitioned or partitioned tables

```
CREATE TABLE tab_8(c NUMBER, c2 date) PARALLEL 8;
```

```
CREATE TABLE tab_def(c NUMBER, c2 date) PARALLEL;
```

```
CREATE TABLE dept (no NUMBER, dname VARCHAR2(32))
PARTITION BY HASH (no) PARTITIONS 8 PARALLEL 4;
```

```
ALTER TABLE tab_8 PARALLEL 5;
```

```
ALTER TABLE sales
SPLIT PARTITION sales_q4_2000 AT ('15-NOV-2000')
INTO (PARTITION sales_q4_1, PARTITION sales_q4_2)
PARALLEL 2;
```

ORACLE

If no PARALLEL clause is declared at object creation, NOPARALLEL is set. It involves serial execution by default if no hint is used. The PARALLEL clause can be set for a number of objects, including nonpartitioned and partitioned tables, nonpartitioned and partitioned indexes, IOTs, and external tables.

Specify PARALLEL if you want Oracle to select a degree of parallelism equal to the number of CPUs available on all participating instances times the value of the PARALLEL_THREADS_PER_CPU initialization parameter.

- For a single instance, DOP = PARALLEL_THREADS_PER_CPU * CPU_COUNT
- For an Oracle RAC configuration, DOP = SUM(CPU_COUNT for each instance) * PARALLEL_THREADS_PER_CPU. If services are used, only instances with the service running are included in the count. The parameter PARALLEL_THREADS_PER_CPU is assumed to be the same on all instances.

Specifying PARALLEL n indicates the DOP, which is the number of parallel threads used by one PX set. Each parallel thread may use one or two PX servers.

A query accessing the TAB_8 is processed with a DOP of 8 and potentially allocated 16 parallel servers, 8 producers, and 8 consumers. A query accessing tables with different DOPs is processed with the highest DOP.

The last example splits the SALES_Q4_2000 partition into two new partitions. This operation is performed in parallel with a DOP explicitly set to two. The SELECT is parallelized.

Restrictions

- When partition granules are used, a parallel server process works on an entire partition or subpartition of a table or index. Because partition granules are statically determined by the structure of the table or index when a table or index is created, partition granules do not give you the flexibility in executing an operation in parallel that block granules do. The maximum allowable DOP is the number of partitions. This might limit the utilization of the system and the load balancing across parallel execution servers.
  Partition granules are used in limited situations such as:
  - Parallel DML (PDML) accessing more than one table partition and where a local partitioned bitmap index exists. 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. 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 index range scan that accesses more than one index partition. (The key here is in the index partitions and not the table partitions.)
  - Partition granules that may also be used to create partitioned indexes where the number of index partitions to be created limits the DOP.
- The `PARALLEL` clause for external tables lets you parallelize subsequent queries on the external data and subsequent operations that populate the external table.
- You cannot specify the `PARALLEL` clause for clustered tables.

Declaring DOP for IOTs and Indexes

- Creating IOTs

```
CREATE TABLE admin_iot(I PRIMARY KEY, j, k, l)
ORGANIZATION INDEX PARALLEL;
```

- Creating underlying nonpartitioned or partitioned indexes

```
CREATE INDEX ord_ix  ON orders (customer_id) PARALLEL;
```

```
SQL> CREATE INDEX gix ON ord_items (order_id) GLOBAL
  3  PARTITION BY HASH (order_id)
  4  PARTITIONS 4 STORE IN (ts_1,ts_2,ts_3,tbs_4)
  5  PARALLEL 8;
```

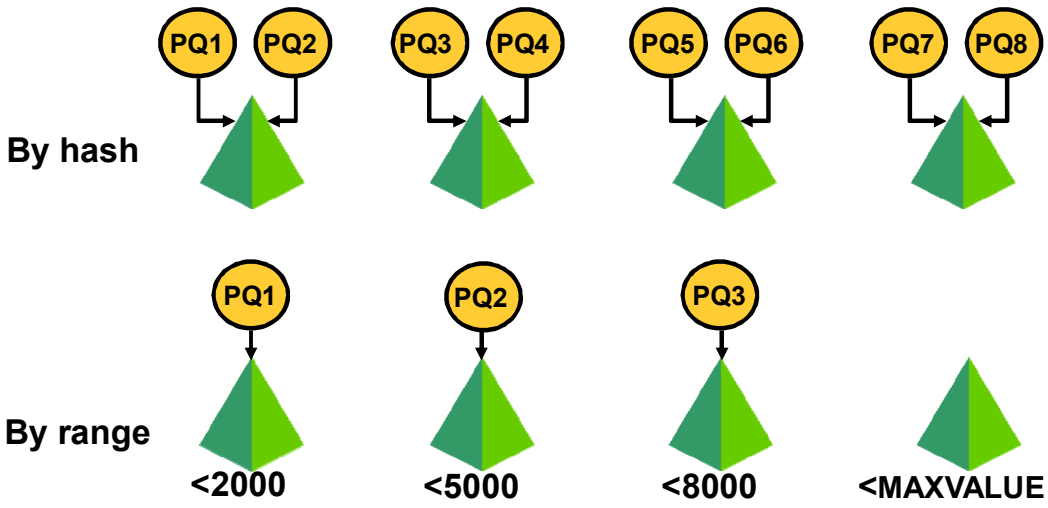
- With range-partitioned indexes:
  - Pruning can be performed, but the degree of parallelism is limited to the number of partitions accessed.
- With hash-partitioned indexes:
  - All partitions can be accessed in parallel.

The slide provides the following examples:

- The first example sets the dictionary DOP of the `ADMIN_IOT` table.
- If you created the `ORDERS` table by using a fast parallel load, you may issue the `CREATE INDEX` statement shown in the second example to quickly create an index in parallel. (Oracle Database chooses the appropriate degree of parallelism.) Note that in this case, the default degree of parallelism used to create the index is also stored as the dictionary DOP.
- The last example adjusts the DOP of the `CREATE INDEX` to the number of created partitions. The DOP is 8, allowing two parallel execution servers per partition.

DOP Usage with Hash-Partitioned Indexes

```
SQL> SELECT /*+ PARALLEL_INDEX(oi,gix,8) */ count(*)
  2  FROM ord_items oi
  3  WHERE order_id BETWEEN 1000 AND 6000;
```



Usage Example

When they are parallelized, queries with bad selectivity (for example, long-running queries with range predicates) perform better with hash-partitioned indexes than with range-partitioned indexes. With range-partitioned indexes, pruning can be performed, but the degree of parallelism is limited to the number of partitions accessed. However, for hash-partitioned indexes, all partitions can be accessed in parallel.

This example retrieves all rows for which `ORDER_ID` is between 1000 and 6000.

If the hash-partitioned global index `GIX` is used, partition pruning cannot be performed, but a parallel fast full scan of each partition is possible.

When a range-partitioned global index is used, partition pruning is performed to identify only the relevant partitions. Only one execution server per scanned partition can be used in this case.

Parallel Hints

- The scope of parallel hints is at the *statement* level, superseding parallelism specified at the *object* level.
  - Hints beginning with `PARALLEL` indicate the degree of parallelism for a specified *object*, when defined as follows:

```
/*+ PARALLEL(emp, 3)                    */
/*+ PARALLEL(dept)                      */
/*+ PARALLEL_INDEX(dept, ix_dept ,4)    */
/*+ NOPARALLEL_INDEX(dept, ix_dept)     */
```

  - The scope of the `PARALLEL` hint is the *statement*, not an *object*, when defined as follows:

```
/*+ PARALLEL(MANUAL) | NO_PARALLEL      */
/*+ PARALLEL(n)                         */
```

ORACLE

Parallel execution hints instruct the optimizer about whether and how to parallelize operations. The scope of parallel hints is at the *statement* level, superseding whatever parallelism might be specified at the table and object level.

Object-Level PARALLEL Hint

- `PARALLEL (object, integer)`: The integer value specifies the degree of parallelism for the specified table.
- `PARALLEL (object, DEFAULT)`: Specifying `DEFAULT` or no value signifies that the query coordinator should examine the settings of the initialization parameters to determine the default degree of parallelism.
- `PARALLEL_INDEX(table1, index1, integer)`: The hint instructs the optimizer to use the specified number of concurrent servers to parallelize index range scans, full scans, and fast full scans for partitioned indexes.
- `PARALLEL (MANUAL)`: The optimizer is forced to use the parallel settings of the objects in the statement.
- `PARALLEL (integer)`: The optimizer uses the degree of parallelism specified by integer.

Implication of Statement-Level Parallel Hints

- Set parallelism on the EMPLOYEES table to 2 and disable parallelism on the DEPARTMENTS table, as follows:

```
ALTER TABLE employees PARALLEL 2;
ALTER TABLE departments NOPARALLEL;
```

- Example of parallel hint at *object-level*:

```
SELECT /*+ PARALLEL(employees, 3) */ e.last_name, d.department_name
FROM   employees e, departments d
WHERE  e.department_id=d.department_id;
```

- Example of parallel hint at *statement-level*:

```
SELECT /*+ PARALLEL(MANUAL) */ last_name
FROM employees hr_emp;
```

The PARALLEL hint advises the optimizer to use the degree of parallelization currently in effect for the table itself, which is 2.

Using the parallelism settings given at the top of this slide, the implications of statement-level parallel hints are compared to object-level hints.

In an example where PARALLEL_DEGREE_POLICY is set to MANUAL (the default), the PARALLEL hint specifies a degree of parallelism of 3 for EMPLOYEES. The PARALLEL hint applies to the object. This setting overrides the degree of parallelism of 2 specified in the EMPLOYEES table definition.

In an example where PARALLEL_DEGREE_POLICY is set to LIMITED, the scope of the PARALLEL hint is the statement, not an object. In this case, the PARALLEL hint forces the statement to execute in parallel.

In contrast to most hints, the PARALLEL statement-level hints take precedence over object-level hints, except when the hint contains PARALLEL (MANUAL). In this last case, the optimizer is forced to use the parallel settings of the objects in the statement.

Note: Additional parallel hints are not listed. For a complete list, please consult the *Oracle Database SQL Language Reference 12c Release 1* documentation manual.

View PARALLEL Degree

View parallel degree values of objects:

```
SELECT table_name, degree FROM DBA_TABLES
WHERE table_name like '%TAB_%';
```

```
TABLE_NAME                     DEGREE
------------------------------ ----------------------
PART_TAB_2                     2
TAB_DEF                        DEFAULT
TAB_2                          2
```

```
SELECT index_name, degree FROM DBA_INDEXES
WHERE index_name like 'I_TAB_%';
```

```
INDEX_NAME                     DEGREE
------------------------------ ----------------------
I_TAB_2                        2
```


Those objects having a DEFAULT degree value are created with one of three possible values:

1. DEFAULT - Parallel Execution: The actual degree is determined by calculating the default degree of parallelism, based on other parameters.
2. An integer - Parallel Execution: The degree is a specified number.
3. NULL - Serial Execution: No valued is defined.

Example

In the following code example, the output for DEGREE is the value DEFAULT:

```
SQL>  create table TEST (c number) PARALLEL;
SQL>  SELECT table_name, degree FROM DBA_TABLES
  2   WHERE table_name='TEST';

TABLE_NAME                     DEGREE
------------------------------ ----------------------------------
TEST                            DEFAULT
```

Reminder of Parallel Execution Terminology

- Parallel execution coordinator (QC): Parses the query and distributes work between the parallel execution (PX) servers.
- PX servers: Do the work and pass results back to the QC.
- Table queue (TQ):
  - Uses memory/piping mechanism used to move data between PX servers and the QC process
  - Is used to communicate to or from the QC and PX servers
- PX server set: Is a collection of PX servers that execute one operation:
  - Sort operation
  - `GROUP BY` operation
- Intraoperation within a PX server set and interoperation between PX server sets: Producers and consumers

ORACLE

Each SQL statement undergoes an optimization and parallelization process when it is parsed. After the optimizer determines the execution plan of a statement, the QC 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 QC must decide whether an operation can be performed in parallel and, if so, how many PX servers to enlist. The number of PX servers in one set is the degree of parallelism (DOP).

The QC examines each operation in a SQL statement's execution plan, and then determines the way in which the rows operated on by the operation must be divided or redistributed among the parallel execution servers. If parallel execution is chosen, then the following steps occur:

1. The shadow process takes on the role of a coordinator, often called the query coordinator (QC).
2. The QC obtains the necessary number of PX 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 PX servers perform each operation in parallel if possible.

4. When the PX servers are finished executing the operation they are in charge of, the QC 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 QC returns the result to the user.

V$PQ_TQ_STAT View

The work performed by producers and consumers is visible in the V$PQ_TQSTAT view :

```
SELECT * FROM v$pq_tqstat
ORDER BY tq_id, server_type desc;
```

TQ_ID	SERVER_TYPE	NUM_ROWS	BYTES	OPEN_TIME	AVG_LATENCY	WAITS	TIMEOUTS	PROCESS	INSTANCE	CON_ID
0	Producer	4	125	0	0	0	0	P003	1	0
0	Producer	4	125	0	0	0	0	P002	1	0
0	Consumer	2	86	0	0	17	13	P001	1	0
0	Consumer	6	164	0	0	17	13	P000	1	0
1	Producer	1	35	0	0	2	0	P001	1	0
1	Producer	3	60	0	0	2	0	P000	1	0
1	Consumer	4	95	0	0	1	1	QC	1	0

Note: The column DFO_NUMBER is not displayed for formatting reasons.

The work performed by producers and consumers is visible in V$PQ_TQSTAT view. The V$PQ_TQSTAT view shows the statistics from the last parallel statement. The user will need to commit or roll back to see this data after a parallel DML statement. The columns displayed include:

- DFO_NUMBER: Data Flow Operator (DFO) tree number to differentiate queries
- TQ_ID: Table queue ID within the query, which represents the connection between two DFO nodes in the query execution tree
- SERVER_TYPE: The role in table queue - producer/consumer/ranger
- NUM_ROWS: The number of rows produced/consumed
- BYTES: The number of bytes produced/consumed
- OPEN_TIME: Time (seconds) the table queue remained open
- AVG_LATENCY: Time (minutes) for a message to be dequeued after it enters the queue
- WAITS: The number of waits encountered during dequeue
- TIMEOUTS: The number of timeouts when waiting for a message
- PROCESS: Process ID
- INSTANCE: Instance ID
- CON_ID: The ID of the container to which the data pertains

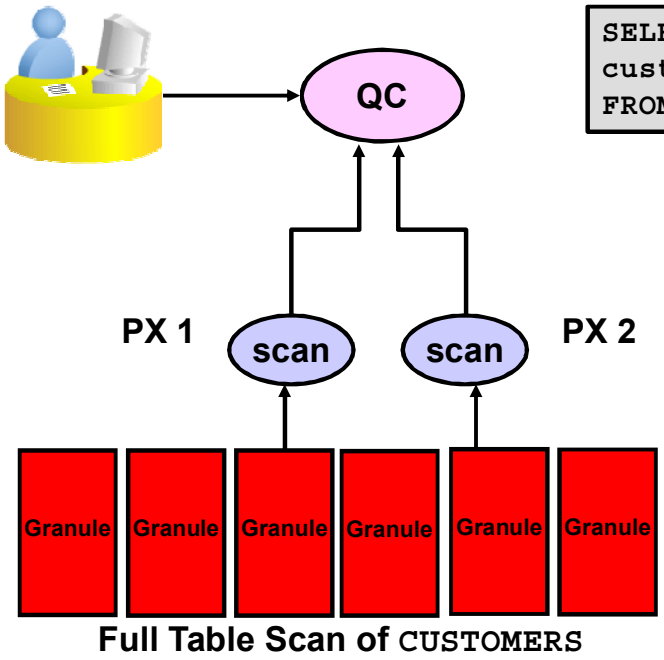
Parallel Execution Example 1: Single Nonparallel Table Scan

```
SELECT /*+ NO_PARALLEL */ cust_address
FROM customers;
```


```
-------------------------------------------------------------------------
|Id  | Operation          | Name      |Rows | Bytes|Cost (%CPU)| Time    |
-------------------------------------------------------------------------
| 0 | SELECT STATEMENT   |           |  319| 18821|    5   (0)| 00:00:01
| 1 |  TABLE ACCESS FULL| CUSTOMERS |  319| 18821|    5   (0)| 00:00:01
-------------------------------------------------------------------------
```


This slide has a simple example of a query that can be parallelized, as will be detailed in the following pages. However, the query can also be executed in serial mode. In this case, the execution plan shows a Table Access Full on the `CUSTOMERS` table performed by a single process, the process server.

Parallel Execution Example 1: Single Parallel Table Scan



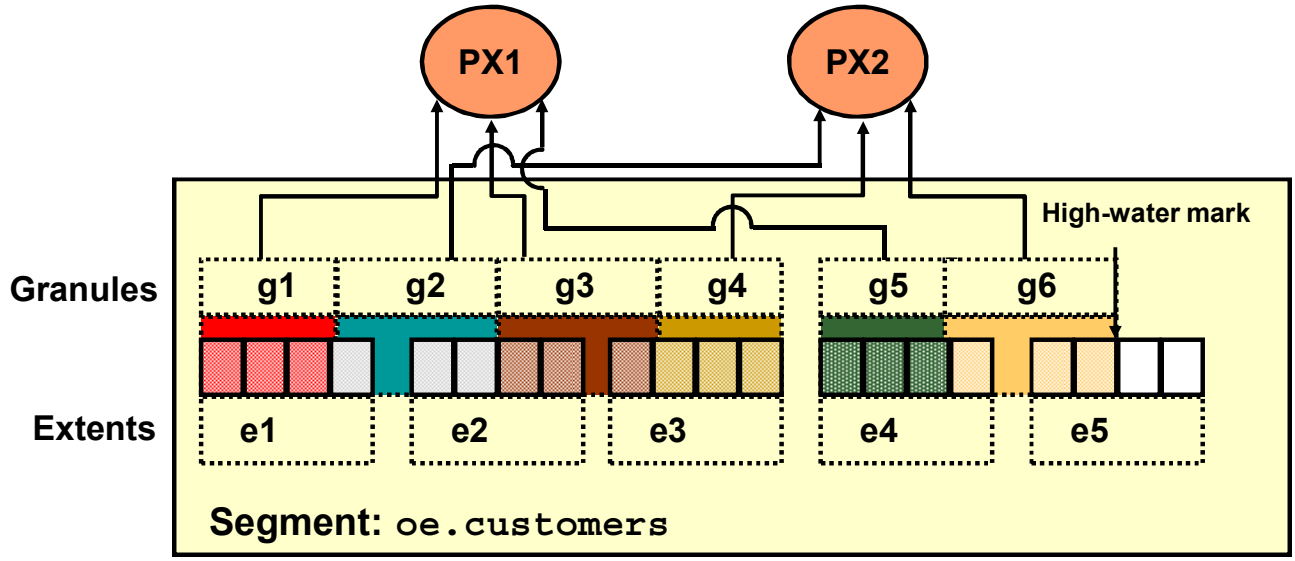

```
SELECT /*+ PARALLEL (c,2) */
cust_address
FROM oe.customers c;
```

ORACLE

Given one set of two PX servers for the query plan, the execution proceeds as follows:

1. The two PX servers scan the `CUSTOMERS` table in parallel according to the type of granule. The type of granularity is either block-range granule or partition granule. The QC decided to distribute the scanning work to the two PX servers by block-range method, with each having a range of blocks to read.
2. In the slide example, when the scan operation of the `CUSTOMERS` table is completed in parallel, the QC returns the complete query result to the user.

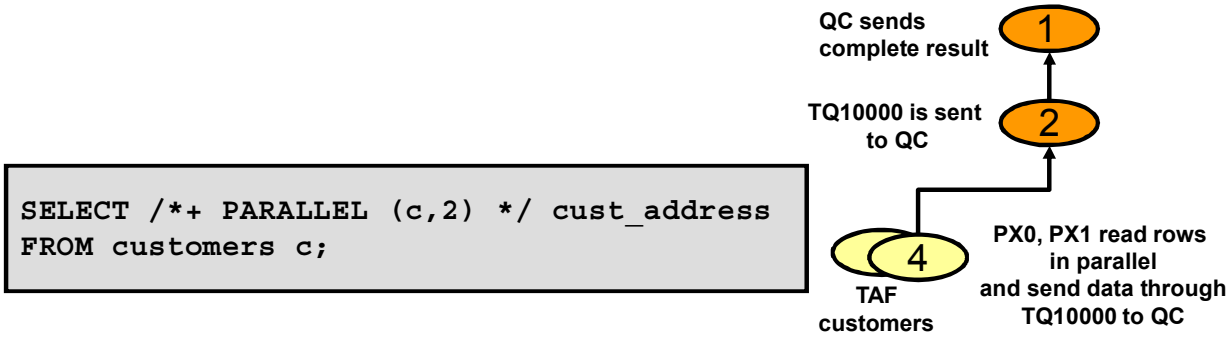
Block-Range Granules



The QC has shared the amount of blocks among two PX servers.

1. PX1 reads granule g1 and meanwhile PX2 reads granule g2.
2. When completed, PX1 reads granule g3 and meanwhile PX2 reads granule g4.
3. When completed, PX1 reads granule g5 and meanwhile PX2 reads granule g6.

Parallel Execution Example 1: Single Parallel Table Scan




```
SELECT /*+ PARALLEL (c,2) */ cust_address
FROM customers c;
```

Id	Operation	Name	Rows	TQ	IN-OUT	PQ Distrib
0	SELECT STATEMENT		319			
1	PX COORDINATOR					
2	PX SEND QC (RANDOM)	:TQ10000	319	Q1,00	P->S	QC (RAND)
3	PX BLOCK ITERATOR		319	Q1,00	PCWC	
4	TABLE ACCESS FULL	CUSTOMERS	319	Q1,00	PCWP	

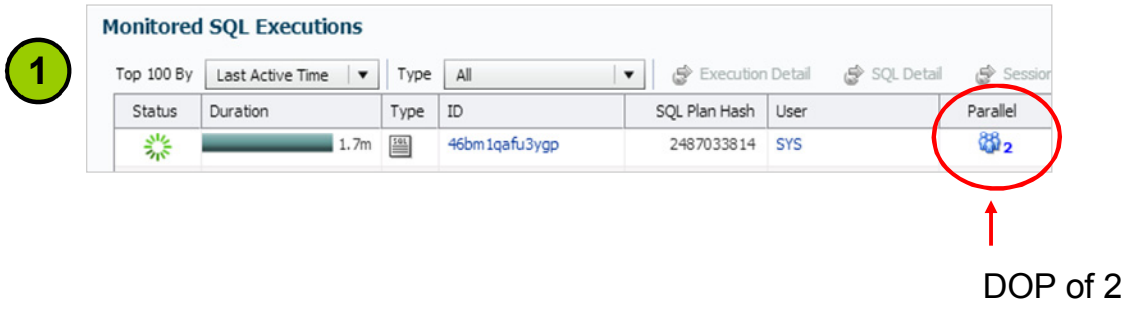
ORACLE

The `PX BLOCK ITERATOR` row source represents the splitting up of the `CUSTOMERS` table into pieces (block-range granules) so as to divide the scan workload between the parallel scan servers.

The `PX SEND QC` row source represents the data being sent to the `QC` in random order (RAND) through the TQ10000 table queue. The `QC` consumes the input randomly in serial (P -> S). Random order is used when the statement does not have an `ORDER BY` clause.

The `PX COORDINATOR` row source represents the `QC` or query coordinator, which controls and schedules the parallel plan appearing below it in the plan tree.

Parallel Execution Example 1: Single Parallel Table Scan



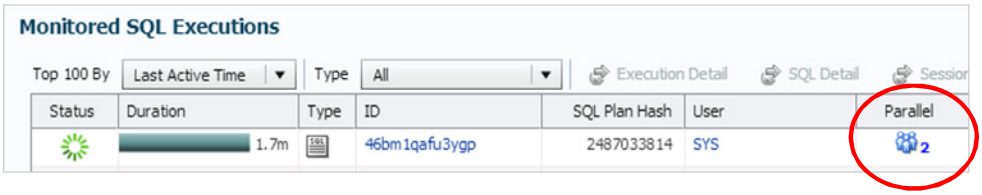
DOP of 2

You can access the SQL Monitoring feature in Enterprise Manager Cloud Control 12*c* by navigating to the database home page and clicking the Performance menu item. Scroll down and click **SQL Monitoring**.

Enterprise Manager Monitored SQL Executions shows the request, which has completed with a DOP of 2.

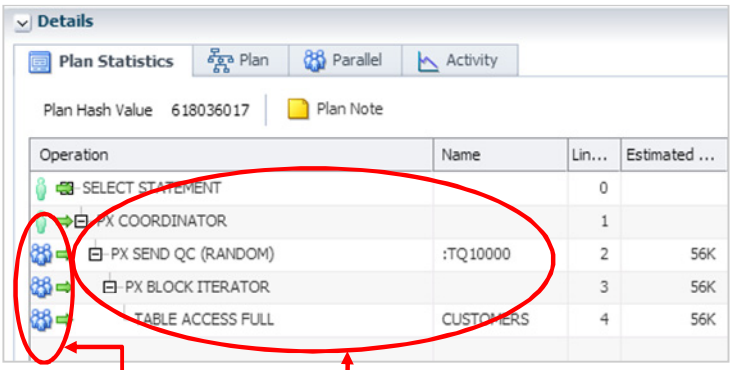
Parallel Execution Example 1: Single Parallel Table Scan

1



DOP of 2

2

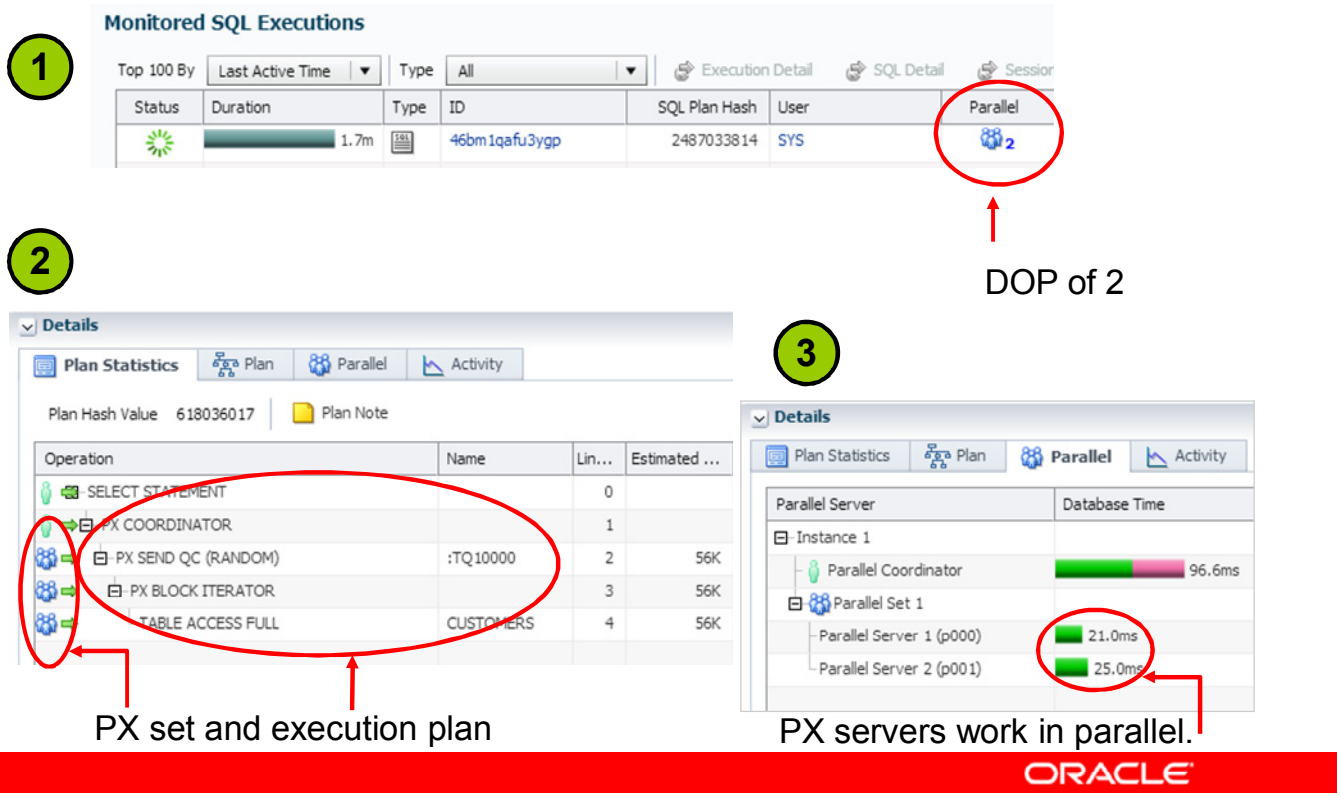


PX set and execution plan

ORACLE

Clicking the ID link of the statement shows the execution plan with the QC (the man in green) and one PX server set (several men in blue).

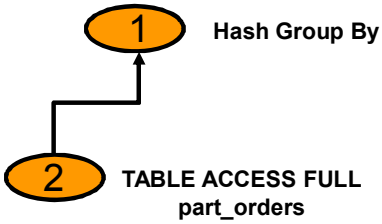
Parallel Execution Example 1: Single Parallel Table Scan



The Parallel tab of the same statement shows the number of PX servers that were involved in the PX server set. In this case, two PX servers were involved.

Parallel Execution Example 2: Nonparallel GROUP BY

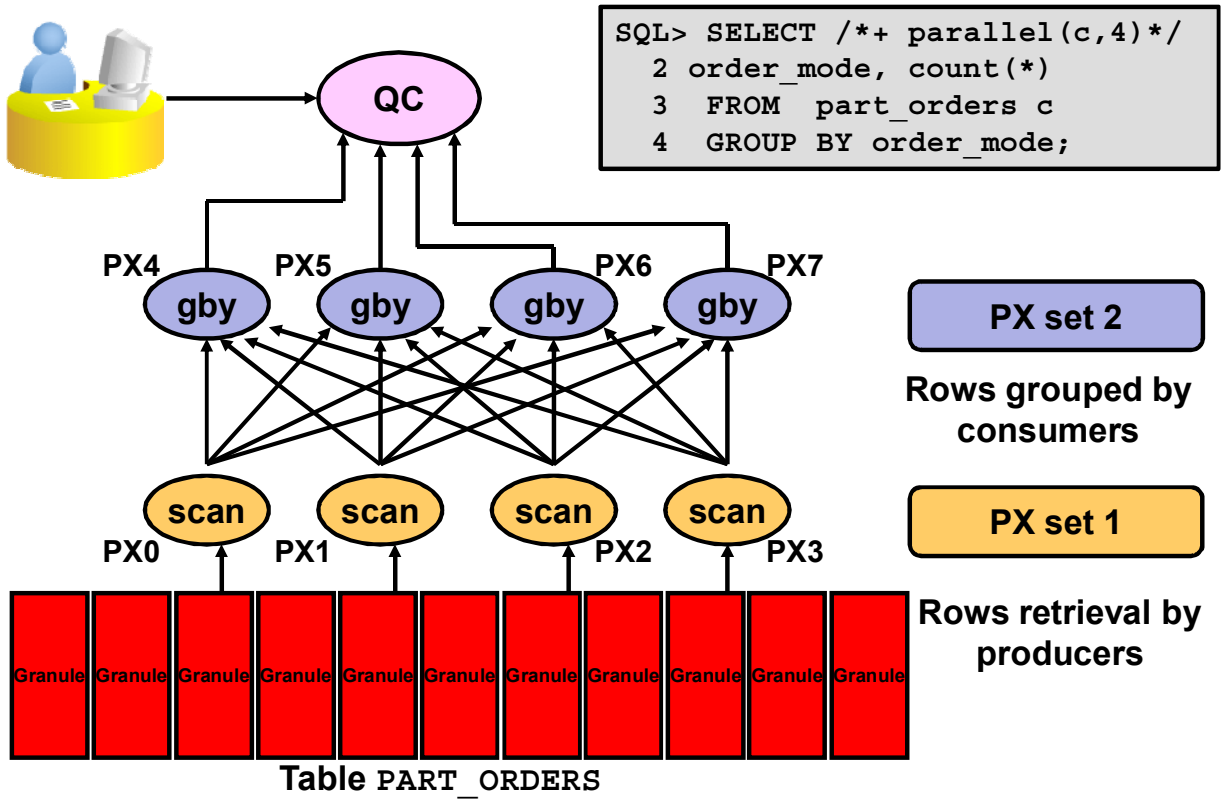
```
SELECT /*+ NO_PARALLEL */ order_mode,
count(*)
FROM part_orders c
GROUP BY order_mode;
```




```
-------------------------------------------------------------------------
| Id  | Operation          | Name        | Rows  | Bytes | Cost (%CPU)|
-------------------------------------------------------------------------
|   0 | SELECT STATEMENT   |             |   171 |  1197 |     6  (17)|
|   1 |  HASH GROUP BY     |             |   171 |  1197 |     6  (17)|
|   2 |   TABLE ACCESS FULL|PART_ORDERS  |   319 |  2233 |     5   (0)|
-------------------------------------------------------------------------
```


The `HASH GROUP BY` sort operator hashes sets of rows from the `PART_ORDERS` table into groups due to the `GROUP BY` clause.

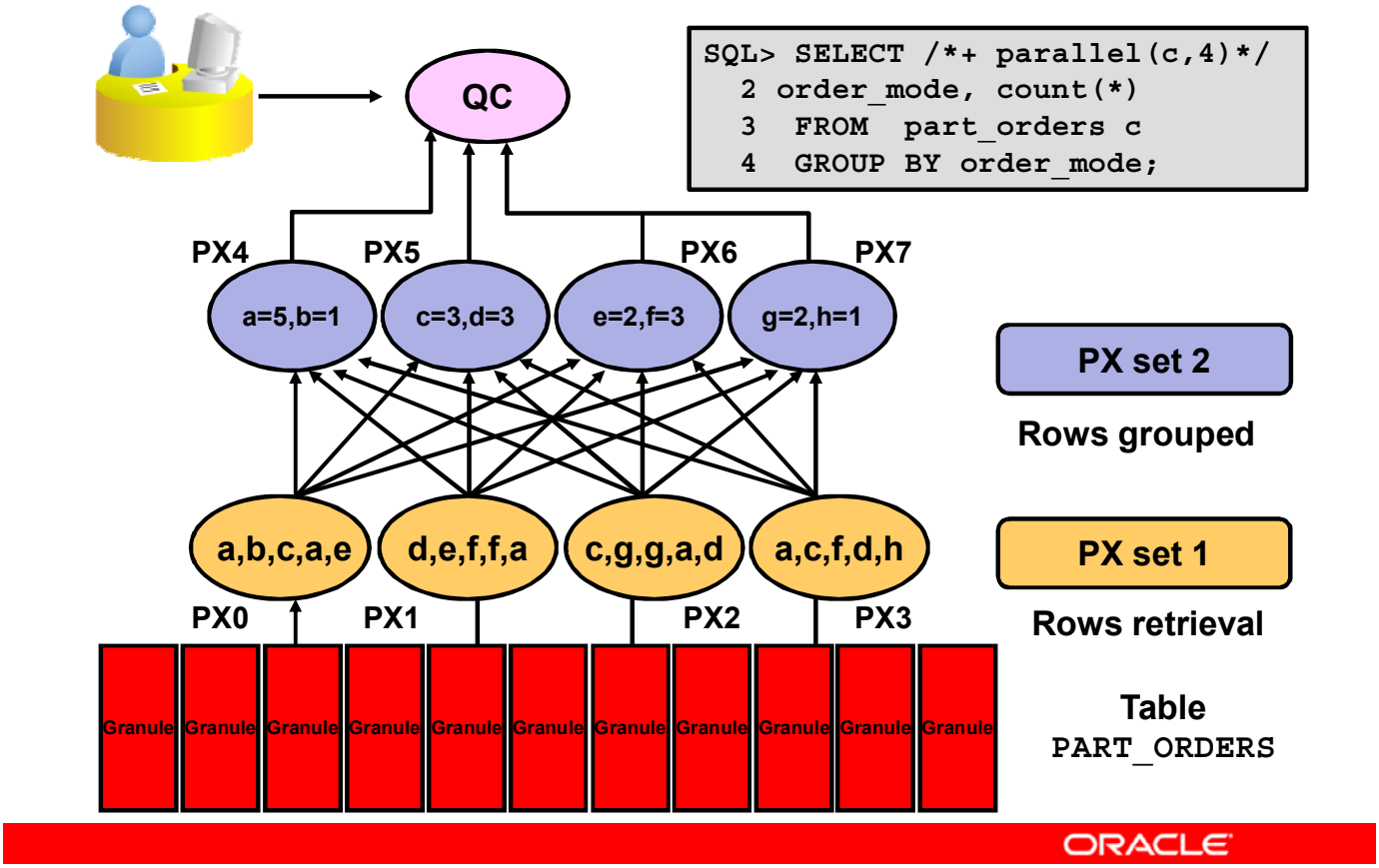
Parallel Execution Example 2: Parallel GROUP BY



Given two sets of PX servers for the statement illustrated in the slide, the execution proceeds as follows:

1. Each server set (PX set 1 and PX set 2) has four execution processes according to the PARALLEL hint in the query that specifies the DOP. The query gets 8 PX servers for the DOP of 4.
2. PX set 1 first scans the table PART_ORDERS (the producers) in parallel and sends rows to PX set 2 (the consumers). This is an example of parallel intraoperation.
   Each PX server of PX set 2 gets rows from the PX server of PX set 1, performs the HASH GROUP BY operation in parallel, hashing sets of rows into groups. This is an example of parallel interoperation.
3. After the PX set 2 servers have completed the HASH GROUP BY operation, they send their rows to the QC.
4. The QC sends the result to the user.

Parallel Execution Example 2: Parallel GROUP BY



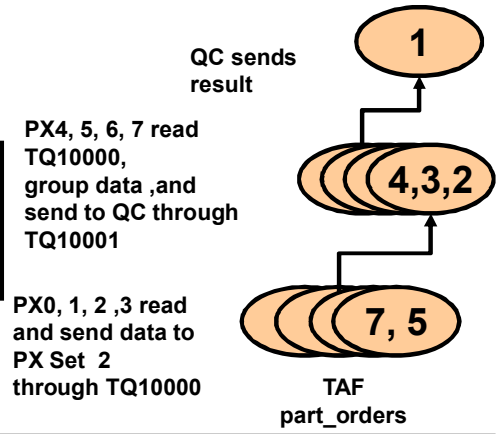
While PX set 1 scans the PART_ORDERS table in parallel and sends rows to PX set 2, each PX server of PX set 2 gets rows from the PX server of PX set 1, performs the HASH GROUP BY operation in parallel, hashing sets of rows into groups.

Here is an example: While PX server PX0 is retrieving rows a, b, c, a, e, PX server PX1 is retrieving rows d, e, f, f, a, PX server PX2 is retrieving rows c, g, g, a, d, and PX server PX3 is retrieving rows a, c, f, d, h. As soon as each PX server has retrieved its rows, it sends the appropriate rows to the appropriate PX server of PX set 2 because a *HASH* maps rows to query servers of PX set 2 by using a hash function on the ORDER_MODE column which is the column in the GROUP BY clause.

Then each PX server of PX set 2 completes the GROUP BY operation on its own set of values.

Parallel Execution Example 2: Parallel GROUP BY

```
SELECT /*+ PARALLEL (c,4) */ order_mode,
count(*)
FROM part_orders c
GROUP BY order_mode;
```




```
-------------------------------------------------------------------------------
|Id  | Operation                | Name        | Rows |  TQ   |IN-OUT | PQ Distrib
-------------------------------------------------------------------------------
|  0 | SELECT STATEMENT         |             |  171 |       |       |
|  1 |  PX COORDINATOR          |             |      |       |       |
|  2 |   PX SEND QC (RANDOM)    | :TQ10001    |  171 | Q1,01 | P->S  | QC (RAND)
|  3 |    HASH GROUP BY         |             |  171 | Q1,01 | PCWP  |
|  4 |     PX RECEIVE           |             |  319 | Q1,01 | PCWP  |
|  5 |      PX SEND HASH        | :TQ10000    |  319 | Q1,00 | P->P  | HASH
|  6 |       PX BLOCK ITERATOR  |             |  319 | Q1,00 | PCWC  |
|  7 |        TABLE ACCESS FULL |PART_ORDERS  |  319 | Q1,00 | PCWP  |
```


The PART_ORDERS table is scanned in parallel by one set of PX servers (PX set 1) while the aggregation for the GROUP BY is performed by the second set (PX set 2).

The PX BLOCK ITERATOR row source represents the splitting up of the CUSTOMERS table into pieces (block-range granules) so that the scan workload can be divided between the parallel scan servers.

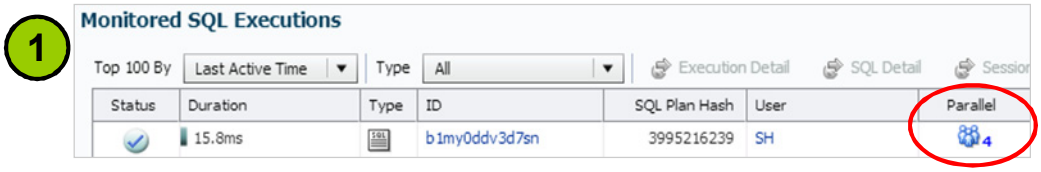
The PX SEND and PX RECEIVE row sources represent the pipe that connects the two PX sets (PX set 1 and PX set 2) as rows flow up from the parallel scan performed by PX set 1, are partitioned through the HASH table queue TQ10000, and then read by and aggregated on the top PX set 2 by the QC.

The PX SEND HASH implements the distribution method taking place between the two PX sets. This shows the boundary between the two PX server sets and how data is repartitioned on the send/producer side. This information was formerly displayed in the DISTRIBUTION column. The HASH maps rows to query servers of PX set 2 by using a hash function on the ORDER_MODE join key.

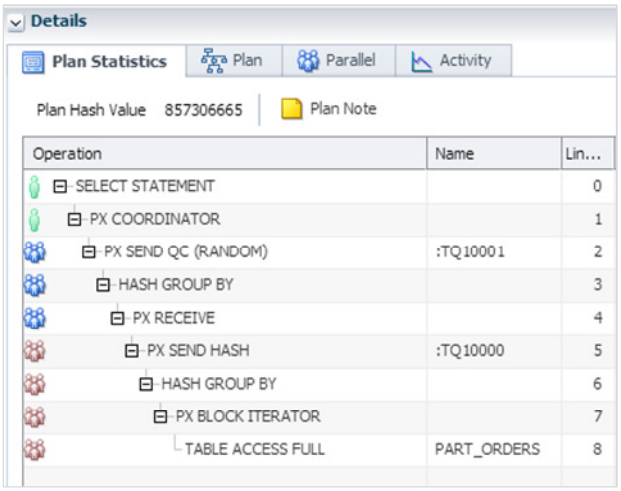
The PX SEND QC row source represents the aggregated values being sent to the QC in random order (RANDOM) through the TQ10001 table queue.

The PX COORDINATOR row source represents the QC or query coordinator. It consumes the input received from PX set 2 randomly.

Parallel Execution Example 2: Parallel GROUP BY



DOP of 4



But two PX sets:

- Consumers in blue
- Producers in red

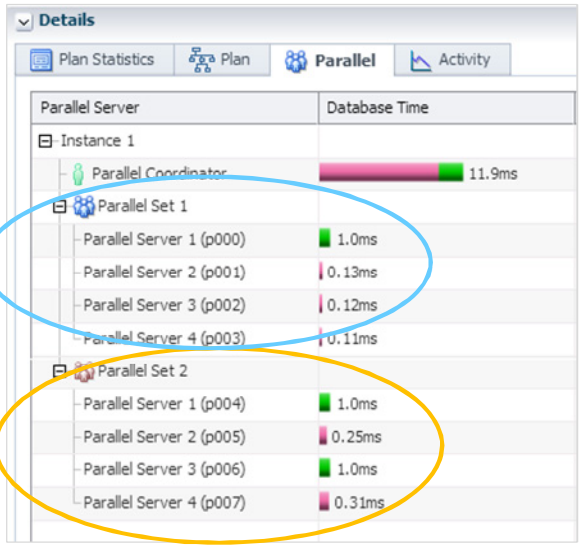
You can access the SQL Monitoring feature in Enterprise Manager Cloud Control 12*c* by navigating to the database home page and clicking the Performance menu item. Scroll down and click SQL Monitoring.

Enterprise Manager Monitored SQL Executions shows the request that has completed with a DOP of 4.

Clicking the ID link of the statement shows the execution plan with the QC (the man in green) and two PX server sets (several men in blue and men in red).

Parallel Execution Example 2: Parallel GROUP BY

3



PX set 2

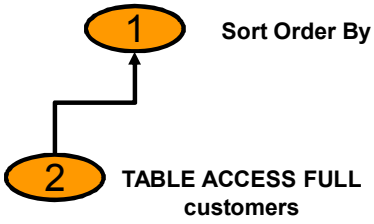
PX set 1

- Four PX servers (PX1) for producing rows working in parallel
- Four PX servers (PX2) for consuming rows working in parallel

The Parallel tab of the same statement shows the number of PX servers that were involved in each PX server set. In this case, four PX servers in each PX set were involved.

Parallel Execution Example 3: Nonparallel ORDER BY

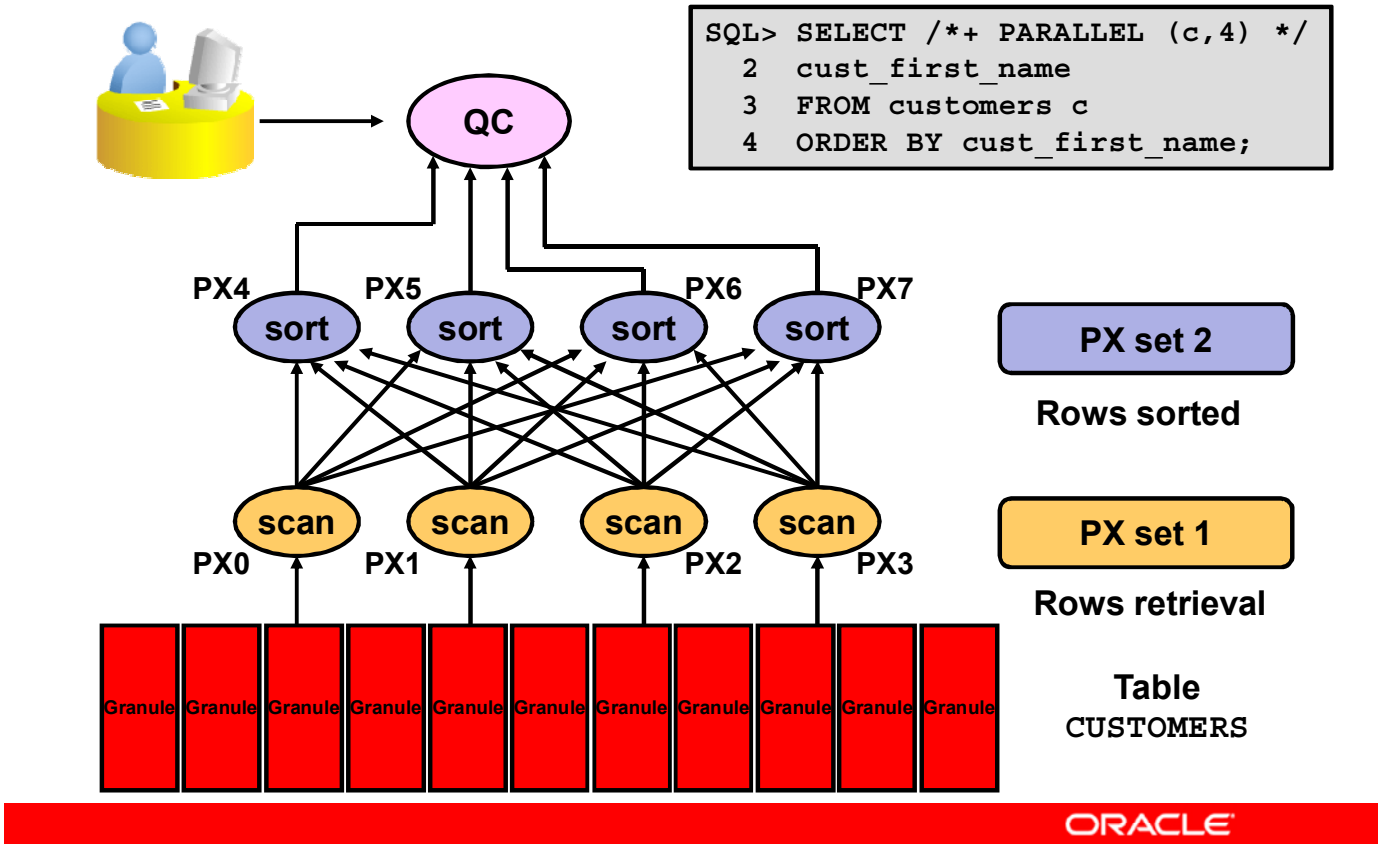
```
SELECT /*+ NO_PARALLEL */ cust_first_name
FROM customers c
ORDER BY cust_first_name;
```




```
-------------------------------------------------------------------------
| Id  | Operation          | Name      | Rows  | Bytes | Cost (%CPU)|
-------------------------------------------------------------------------
|   0 | SELECT STATEMENT   |           |   319 |  2233 |     6  (17)|
|   1 |  SORT ORDER BY     |           |   319 |  2233 |     6  (17)|
|   2 |   TABLE ACCESS FULL| CUSTOMERS |   319 |  2233 |     5   (0)|
-------------------------------------------------------------------------
```


The slide example is a simple example of a query that can be parallelized, as will be detailed in the next pages. But it can also be executed in serial mode. In this case, the execution plan shows a Table Access Full on the `CUSTOMERS` table and a `SORT ORDER BY` operation performed by a single process, the process server.

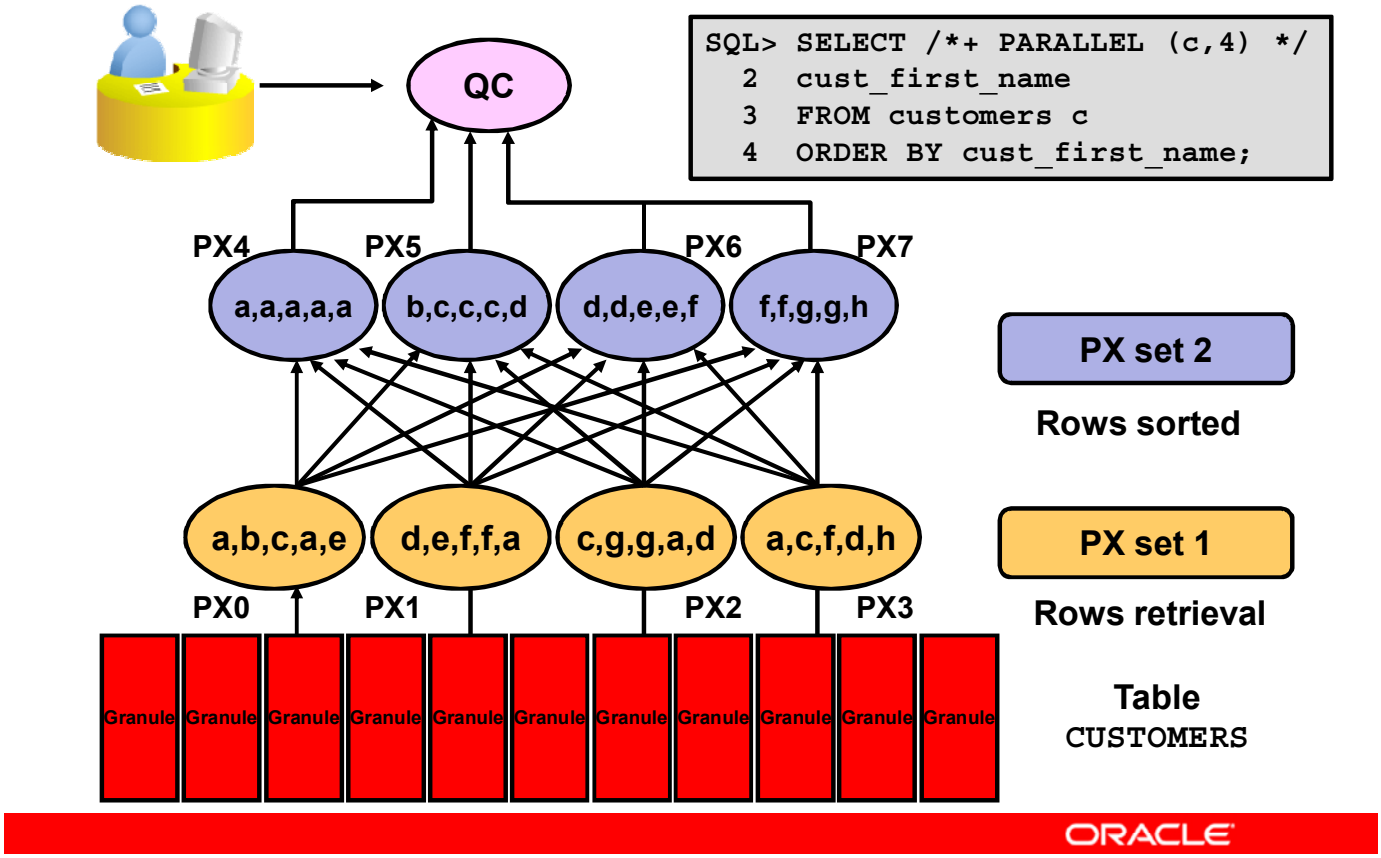
Parallel Execution Example 3: Parallel ORDER BY



Because the DOP for the query is set to 4, which means that four parallel execution servers can be active for any given operation, there are actually eight PX servers involved in the query even though the DOP is 4. The execution proceeds as follows:

1. The execution plan implements a full scan of the `CUSTOMERS` table performed in parallel by the four servers of PX set SS1.
2. While the parallel execution servers of SS1 are producing rows in the `FULL SCAN` of the `CUSTOMERS` table, another set of parallel execution servers SS2 can begin consuming the rows to perform the `SORT ORDER BY` operation.
   All of the PX 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 `CUST_FIRST_NAME` column between Aa and Bi, that row is sent to the first `ORDER BY` parallel execution server of SS2. Each PX server of PX set 1 maps rows to query servers by using ranges of the sort key.
3. When the scan operation is complete, the sorting processes can return the sorted results to the query coordinator in the right order.
4. The QC returns the complete query results to the user.

Parallel Execution Example 3: Parallel ORDER BY



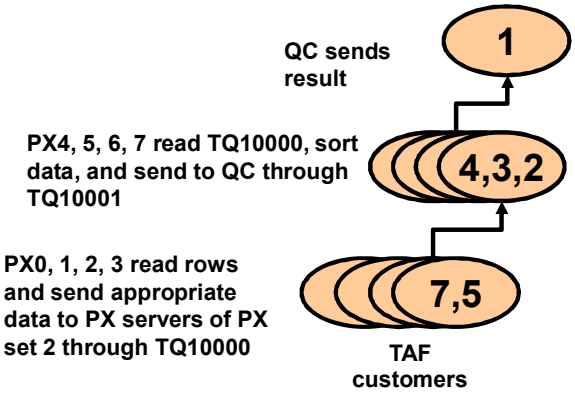
While PX set 1 scans the CUSTOMERS table in parallel and sends rows to PX set 2, each PX server of PX set 2 gets rows from the PX server of PX set 1 and performs the SORT operation in parallel. If a row scanned by a parallel execution server contains a value for the CUST_FIRST_NAME column between Aa and Bi, that row is sent to the first ORDER BY parallel execution server of SS2.

Here is an example: While PX server PX0 is retrieving rows a, b, c, a, e, PX server PX1 is retrieving rows d, e, f, f, a, PX server PX2 is retrieving rows c, g, g, a, d, and PX server PX3 is retrieving rows a, c, f, d, h. As soon as each PX server in PX set 1 retrieved its rows, it sends the appropriate rows to the appropriate PX server of PX set 2 because a map occurs from retrieved rows to query servers using ranges of the CUST_FIRST_NAME sort key.

Then each PX server of PX set 2 completes the ORDER BY operation on its own set of values.

Parallel Execution Example 3: Parallel ORDER BY

```
SQL> SELECT /*+ PARALLEL (c,4) */
  2  cust_first_name
  3  FROM customers c
  4  ORDER BY cust_first_name;
```



Id	Operation	Name	Rows	TQ	IN-OUT	PQ Distrib
0	SELECT STATEMENT		319			
1	PX COORDINATOR					
2	PX SEND QC (ORDER)	:TQ10001	319	Q1,01	P->S	QC (ORDER)
3	SORT ORDER BY		319	Q1,01	PCWP	
4	PX RECEIVE		319	Q1,01	PCWP	
5	PX SEND RANGE	:TQ10000	319	Q1,00	P->P	RANGE
6	PX BLOCK ITERATOR		319	Q1,00	PCWC	
7	TABLE ACCESS FULL	CUSTOMERS	319	Q1,00	PCWP	

ORACLE

The CUSTOMERS table is scanned in parallel by one set of four PX servers while the sorting for the SORT ORDER BY is performed by the second set of PX servers.

The PX BLOCK ITERATOR row source represents the splitting of the CUSTOMERS table into pieces (block-range granules) so that the scan workload can be divided between the parallel scan servers.

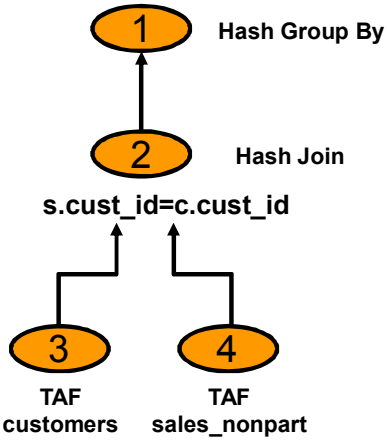
The PX SEND and PX RECEIVE row sources represent the pipe that connects the two PX server sets as rows flow up from the parallel scan performed by PX set 1, are repartitioned through the *RANGE* table queue TQ10000, and then read by and sorted on the top PX server set (PX set 2) by the QC.
The PX SEND *RANGE* maps rows to query servers by using ranges of the sort key. This method is used when the statement contains an ORDER BY clause.

The PX SEND QC (*ORDER*) row source represents the ordered values being sent to the QC in sorted (*ORDER*) order through the table queue TQ10001. The logic of the TQ10001 takes care that the range of sorted values are sent in order to the QC. The PQ consumes the data in serial (P->S).

The PX COORDINATOR row source represents the QC. The PX COORDINATOR consumes the input received from PX set 2 in order, from the first to the last query server. It is used when the statement contains an ORDER BY clause.

Parallel Execution Example 4: Nonparallel Join

```
SELECT /*+ NO_PARALLEL */ c.cust_first_name,
c.cust_last_name,
MAX(quantity_sold), AVG(quantity_sold)
FROM sales_nonpart s, customers c
WHERE s.cust_id=c.cust_id
GROUP BY c.cust_first_name,c.cust_last_name;
```




```
-------------------------------------------------------------------------
| Id | Operation           | Name          |Rows  |TempSpc| Cost (%CPU)|
-------------------------------------------------------------------------
|   0| SELECT STATEMENT    |               |  834K|       | 12956   (1)|
|   1|  HASH GROUP BY      |               |  834K|    45M| 12956   (1)|
|*  2|   HASH JOIN         |               |  845K|  1736K|  3254   (1)|
|   3|    TABLE ACCESS FULL| CUSTOMERS     | 55500|       |   405   (1)|
|   4|    TABLE ACCESS FULL| SALES_NONPART |  845K|       |  1236   (1)|
-------------------------------------------------------------------------
```


Reminder of `HASH JOIN` Method

Hash join is used for joining large data sets. The optimizer uses the smaller of two tables or data sources to build a hash table on the join key in memory. It then scans the larger table, probing the hash table to find the joined rows. This method is best used when the smaller table fits in the available memory. The cost is then limited to a single read pass over the data for the two tables.

The system reads the first data set and builds an array of hash buckets in memory. A hash bucket is little more than a location that acts as the starting point for a linked list of rows from the build table. A row belongs to a hash bucket if the bucket number matches the result that the system gets by applying an internal hashing function to the join column or columns of the row.

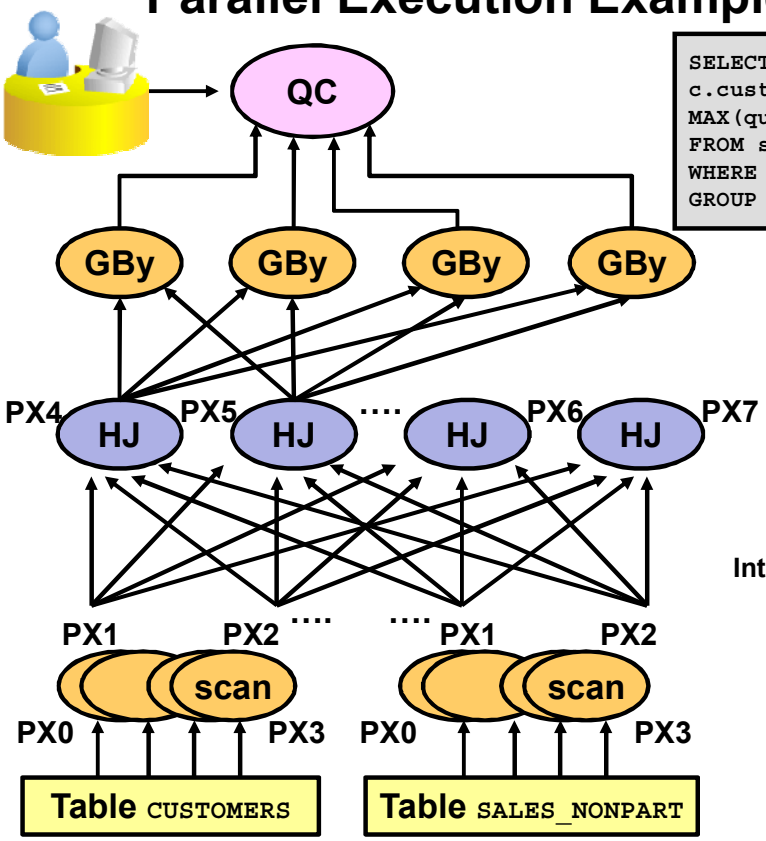
The system starts to read the second set of rows, using whatever access mechanism is most appropriate for acquiring the rows, and uses the same hash function on the join column or columns to calculate the number of the relevant hash bucket. The system then checks whether there are any rows in that bucket. This is known as probing the hash table.
If there are no rows in the relevant bucket, the system can immediately discard the row from the probe table.

If there are some rows in the relevant bucket, the system does an exact check on the join column or columns to see if there is a proper match. Any rows that survive the exact check can immediately be reported (or passed on to the next step in the execution plan). So, when you perform a hash join, you must fetch all rows from the smallest row source to return the first row to the next operation.

Reminder of the `HASH GROUP BY` Sorting Operator

`HASH GROUP BY` hashes a set of rows into groups for a query with a `GROUP BY` clause.

Parallel Execution Example 4: Parallel Join




```
SELECT /*+ PARALLEL(4) */ c.cust_first_name,
c.cust_last_name,
MAX(quantity_sold), AVG(quantity_sold)
FROM sales_nonpart s, customers c
WHERE s.cust_id=c.cust_id
GROUP BY c.cust_first_name,c.cust_last_name;
```

PX set 1

Consume rows from PX set 2
and produce rows for QC

PX set 2

Consume rows from PX set 1
Interoperation between PX set 1 and PX set 2
Then produce rows for PX set 1

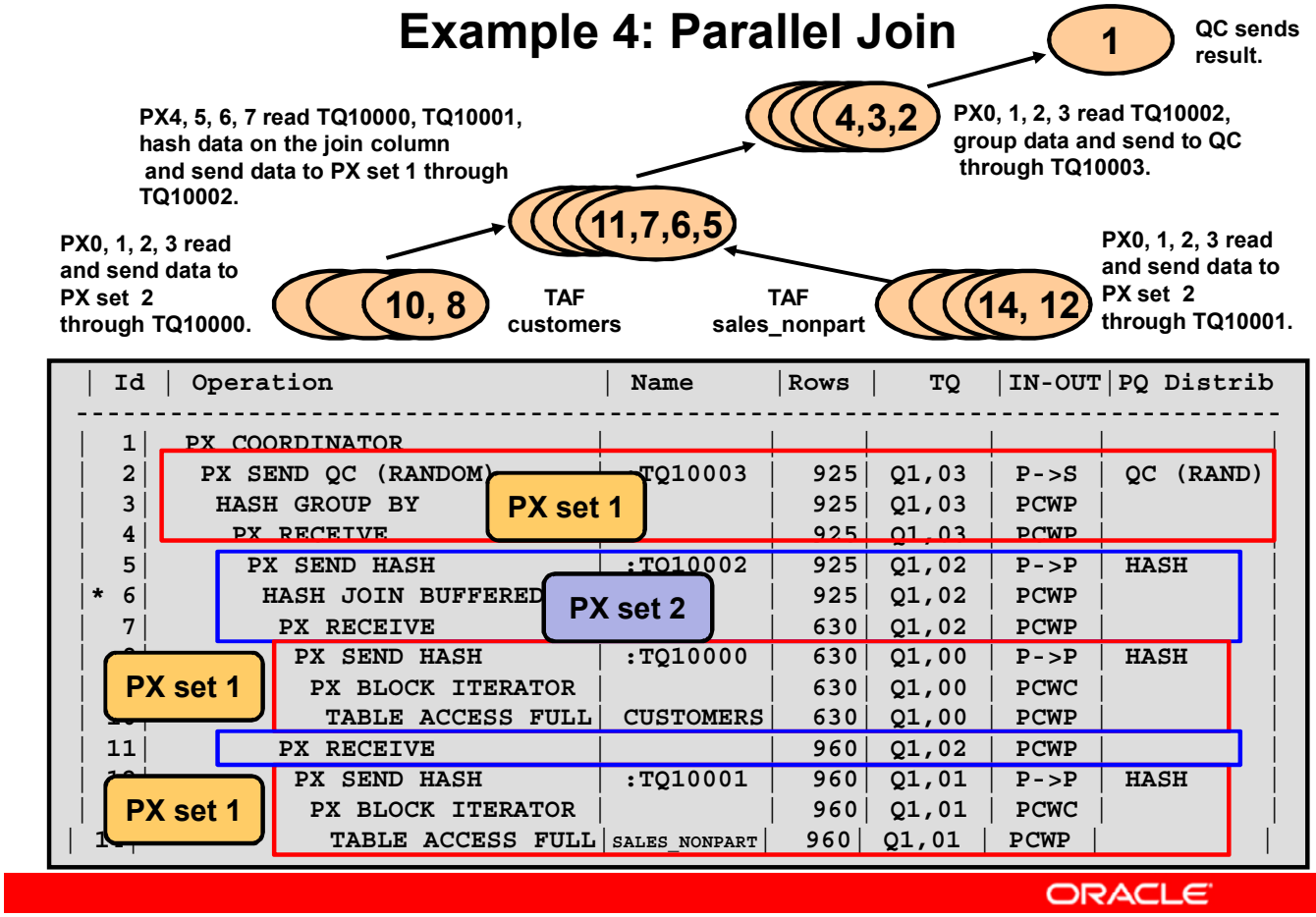
PX set 1

Produce rows
Intraoperation within a PX servers set

ORACLE

Given the DOP defined in the slide statement, the execution proceeds as follows:

1. Each server set (PX set 1 and PX set 2) has four PX server processes.
2. PX set 1 first scans the table `CUSTOMERS` and sends rows to PX set 2, which builds a hash table on the rows. That is, the consumers in PX set 2 and the producers in PX set 1 work concurrently: one is 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 interoperation parallelism.
3. After PX set 1 has finished scanning the entire `CUSTOMERS` table, it scans the `SALES_NONPART` table in parallel. It sends its rows to servers in PX set 2, which then performs the probes to finish the hash join in parallel.
4. After PX set 1 has scanned the `SALES_NONPART` table in parallel and sent the rows to PX set 2, it switches to performing the `GROUP BY` operation in parallel. This is how two server sets run concurrently to achieve interoperation parallelism across various operators in the query tree.



Id	Operation	Name	Rows	TQ	IN-OUT	PQ Distrib
1	PX COORDINATOR					
2	PX SEND QC (RANDOM)	:TQ10003	925	Q1,03	P->S	QC (RAND)
3	HASH GROUP BY		925	Q1,03	PCWP	
4	PX RECEIVE		925	Q1,03	PCWP	
5	PX SEND HASH	:TQ10002	925	Q1,02	P->P	HASH
* 6	HASH JOIN BUFFERED		925	Q1,02	PCWP	
7	PX RECEIVE		630	Q1,02	PCWP	
	PX SEND HASH	:TQ10000	630	Q1,00	P->P	HASH
	PX BLOCK ITERATOR		630	Q1,00	PCWC	
	TABLE ACCESS FULL	CUSTOMERS	630	Q1,00	PCWP	
11	PX RECEIVE		960	Q1,02	PCWP	
	PX SEND HASH	:TQ10001	960	Q1,01	P->P	HASH
	PX BLOCK ITERATOR		960	Q1,01	PCWC	
	TABLE ACCESS FULL	SALES_NONPART	960	Q1,01	PCWP	

To find which PX set performed which task, find the information in the `V$PQ_TQSTAT` view.

A better way to display this information is through Enterprise Manager SQL Monitor, using the detailed page of the statement execution and the Parallel tab. The red and blue colors of the PX sets show who did what.

Distribution of Rows

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. In the slide query plan, after a server process in PX set 1 scans a row from the `CUSTOMERS` table, which server process in PX set 2 should it send the row to?

The operator into which the rows are flowing decides the redistribution. In this case (a `PARALLEL JOIN` operator), the redistribution of rows flows up from PX set 1 performing the parallel scan of `CUSTOMERS`, into PX set 2 performing the parallel hash join. This is performed by mapping rows to PX servers using a hash function on the join key. That is, a server process scanning `CUSTOMERS` computes a hash function of the value of the `CUSTOMERS.CUST_ID` column to decide the number of the server process in PX set 2 to send it to. The redistribution method used in parallel queries explicitly shows in the `Distrib` column in the `EXPLAIN PLAN` of the query, or now in the `Operation` itself (`PX SEND` *HASH*).

Note: The next slide shows different distribution methods used according to the operators.

Distribution Methods

- HASH maps rows to query servers by using a hash function on the join key column for joins. It is also used on the grouping column of a GROUP BY clause.
- RANGE maps rows to query servers by using ranges of the sort key. This is used when the statement contains an ORDER BY clause.
- ROUND-ROBIN randomly maps rows to query servers.
- BROADCAST is used when one table is very small compared to the other.
- QC (ORDER): The QC consumes the input in order, from the first to the last query server. This is used when the statement contains an ORDER BY clause.
- QC (RANDOM): The QC consumes the input randomly. This is used when the statement does not have an ORDER BY clause.

Distribution Methods

- HASH maps rows to query servers by using a hash function on the join key. HASH is used for PARALLEL JOIN or PARALLEL GROUP BY. Hash redistribution is a very common way in parallel execution to achieve an equal distribution of work for an individual parallel server.
- RANGE maps rows to query servers by using ranges of the sort key. RANGE is used when the statement contains an ORDER BY clause.
- ROUND-ROBIN randomly maps rows to query servers.
- BROADCAST is used when one table is very small compared to the other.
- QC (ORDER): The QC consumes the input in order, from the first to the last query server. This is used when the statement contains an ORDER BY clause.
- QC (RANDOM): The QC consumes the input randomly. This is used when the statement does not have an ORDER BY clause.

PARALLEL Hint PQ_DISTRIBUTE

You can control the distribution method for joins by using the PQ_DISTRIBUTE hint to specify two distribution methods, one for the outer table and one for the inner table.

The possible distribution values for the PQ_DISTRIBUTE hint are the following:

- HASH, HASH: The rows of each table are mapped to consumer query servers, using a hash function on the join keys. When mapping is complete, each query server performs the join between a pair of resulting partitions. This distribution is recommended when the tables are comparable in size and the join operation is implemented by hash join or sort merge join.
- BROADCAST, NONE: All rows of the outer table are broadcast to each query server. The inner table rows are randomly partitioned. This distribution is recommended when the outer table is very small compared to the inner table. As a general rule, use this distribution when the inner table size multiplied by the number of query servers is greater than the outer table size.
- NONE, BROADCAST: All rows of the inner table are broadcast to each consumer query server. The outer table rows are randomly partitioned. This distribution is recommended when the inner table is very small compared to the outer table. As a general rule, use this distribution when the inner table size multiplied by the number of query servers is less than the outer table size.
- PARTITION, NONE: The rows of the outer table are mapped by using the partitioning of the inner table. The inner table must be partitioned on the join keys. This distribution is recommended when the number of partitions of the outer table is equal to or nearly equal to a multiple of the number of query servers (for example, 14 partitions and 15 query servers).

  Note: The optimizer ignores this hint if the inner table is not partitioned or not equijoined on the partitioning key.
- NONE, PARTITION: The rows of the inner table are mapped by using the partitioning of the outer table. The outer table must be partitioned on the join keys. This distribution is recommended when the number of partitions of the outer table is equal to or nearly equal to a multiple of the number of query servers (for example, 14 partitions and 15 query servers).

  Note: The optimizer ignores this hint if the outer table is not partitioned or not equijoined on the partitioning key.
- NONE, NONE: Each query server performs the join operation between a pair of matching partitions, one from each table. Both tables must be equipartitioned on the join keys.

Examples: Given two tables *r* and *s* that are joined using a hash join, the following query contains a hint to use hash distribution:

```
SELECT /*+ORDERED PQ_DISTRIBUTE(s, HASH, HASH) USE_HASH (s)*/
column_list
FROM r,s WHERE r.c=s.c;
```

Note: You can also control the distribution of rows for parallel INSERT ... SELECT and parallel CREATE TABLE ... AS SELECT statements to direct how rows should be distributed between the producer (query) and the consumer (load) servers.

Join Operations and Bloom Filters

To reduce overheads of data communication between PX sets during parallel joins, Oracle can create and use a memory structure called a bloom filter (BF).

- Used to quickly probe the existence of rows between different row sets
- Used during PX-set-to-PX-set distribution via TQs when PX servers are in the same instance
- Used during PX-set-to-PX-set distribution via TQs and interconnect when using RAC and PX servers to distribute across interconnect
- Used during PX-set-to-PX-set distribution via TQs using an Exadata Database machine to enable cells to reduce the data sent over the storage network

ORACLE

Bloom filters are memory structures that allow a probe for the existence of rows between two sets, preventing data that does not fulfill a join condition from being unnecessarily sent. This reduces the overhead of data communication during parallel join operation in several ways:

- Between two PX sets distributing data within a single instance
- Between two PX sets distributing data over the interconnect in a RAC environment
- Between two PX sets distributing data by using Exadata Database machine cells

Bloom Filter Usage

The optimizer decides to create a BF according to the estimated join selectivity and the amount of data to be processed.

- To force BF usage, use the hint `PX_JOIN_FILTER`. With small tables, Oracle may not follow your suggestion.
- To avoid BF, use the hint `NO_PX_JOIN_FILTER`.

Example 5: Parallel Join with Bloom Filter

```
select * from t1,t2 where t1.id=t2.id and t1.mod=42;
```

```
-------------------------------------------------------------------------
|Id  Operation                          |Name     |Rows |   TQ  |IN-OUT|
-------------------------------------------------------------------------
|   0|SELECT STATEMENT                  |         |3812G|       |      |
|   1| PX COORDINATOR                   |         |     |       |      |
|   2|  PX SEND QC (RANDOM)             |:TQ10002 |3812G|Q1,02  | P->S |
|* 3|   HASH JOIN BUFFERED             |         |3812G|Q1,02  | PCWP |
|   4|    JOIN FILTER CREATE            |:BF0000  |6182M|Q1,02  | PCWP |
|   5|     PX RECEIVE                   |         |6182M|Q1,02  | PCWP |
|   6|      PX SEND HASH                |:TQ10000 |6182M|Q1,00  | P->P |
|   7|       PX BLOCK ITERATOR          |         |6182M|Q1,00  | PCWC |
|* 8|        TABLE ACCESS STORAGE FULL |T1       |6182M|Q1,00  | PCWP |
|   9|    PX RECEIVE                    |         | 13G |Q1,02  | PCWP |
|  10|     PX SEND HASH                 |:TQ10001 | 13G |Q1,01  | P->P |
|  11|      JOIN FILTER USE             |:BF0000  | 13G |Q1,01  | PCWP |
|  12|       PX BLOCK ITERATOR          |         | 13G |Q1,01  | PCWC |
|*13|        TABLE ACCESS STORAGE FULL |T2       | 13G |Q1,01  | PCWP |
-------------------------------------------------------------------------
```

4
2
1
3

In the slide example, the execution plan shows the following :

- The PX set (Q1,00) scans table T1, applies the filter t1.mod=42, and sends the remaining data to the PX set (Q1,02) through TQ10000 (operations 8 and 6).
- The PX set (Q1,02) receives the data from TQ10000 and builds a hash table (operations 3 and 5).

To avoid unnecessary communication later due to data sent that might not fulfill the join condition, Oracle may create a bloom filter identified by BF0000:

- The PX set Q1,02 creates the BF0000 (operation 4).
- The PX set (Q1,01) scans table T2 (operation 13), probes the BF0000 (operation 11) by sending only data that fulfills the join condition to the PX set (Q1,02) through TQ10001 (operation 10). (**Note:** Because of the nature of BF, some false positives might be sent to the TQ: This is why the hash join must still be performed.)
- The PX set (Q1,02) receives the data from TQ10001 (operation 9), probes the hash table (operation 3), and sends the rows fulfilling the join to the QC through TQ10002 (operations 2).
- The QC receives the data from TQ10002 and returns the result set (operations 0 and 1).

Notice that the join is performed by PX set Q1,02 and, therefore, part of the data sent by the PX sets Q1,00 and Q1,01 may be discarded because it does not fulfill the join condition. Data may be unnecessarily sent to the PX set Q1,02. This is particularly noticeable when two conditions are met:

- Most of the data is discarded because it does not fulfill the join condition.
- The PX servers are distributed over several nodes in a RAC environment.

Note: There are only two sets of parallel execution servers (PX sets). For the example given, each set of parallel execution servers is reassigned to different table queues moving forward in the steps of the explain plan.

RAC or Exadata Environment

The bloom filter must be sent to the slaves accessing the second table, because the filter is constructed by the slaves querying the first table, but the slaves for the second table may be in another instance, the RAC DB, or the cells.

Partition-Wise Join Parallel Execution

With partitioned tables, there are two possible parallel execution methods:

- Full partition-wise join (PWJ)
  - A full PWJ divides a large join into smaller joins on pairs of partitions from the two joined tables.
  - To use this feature, equipartition both tables on their join keys, or use reference partitioning.
- Partial partition-wise join
  - Unlike full PWJ, partial PWJ requires only one table to be partitioned on the join key, not both tables.
  - Oracle Database can perform partial PWJ only in parallel.
  - Partial PWJ is more common than full PWJ.

ORACLE

Full Partition-Wise Join

A full partition-wise join divides a large join into smaller joins on pairs of partitions from the two joined tables. To use this feature, you must equipartition both tables on their join keys, or use reference partitioning.

Consider a large join between a `SALES` table and a `CUSTOMERS` table on the column `CUST_ID`. The query finds the records of all customers who bought more than 100 articles in Quarter 3 of 1999 and is a typical example of a SQL statement performing such a join. Such a large join is typical in data warehousing environments. In this case, the entire `CUSTOMERS` table is joined with one quarter of the `SALES` data. In large data warehouse applications, this might mean joining millions of rows. The join method to use in that case is obviously a hash join. You can reduce the processing time for this hash join even more if both tables are equipartitioned on the `CUST_ID` column. This functionality enables a full partition-wise join.

If you parallelize the execution, instead of joining one partition pair at a time, partition pairs are joined in parallel by the PX servers.

Advantages

When using the capability of PWJ, you reduce:

- The amount of data exchanged between the query servers when joins execute in parallel
- Both CPU and memory usage
- In a RAC environment, the data traffic over the Interconnect

You also remove the communication between PX sets, because only one PX set is used instead of two to collect the data from pairs of partitions. Each PX server is responsible for its pair of partition. This is why you can have only as many PX servers as partitions.

Partial Partition-Wise Join

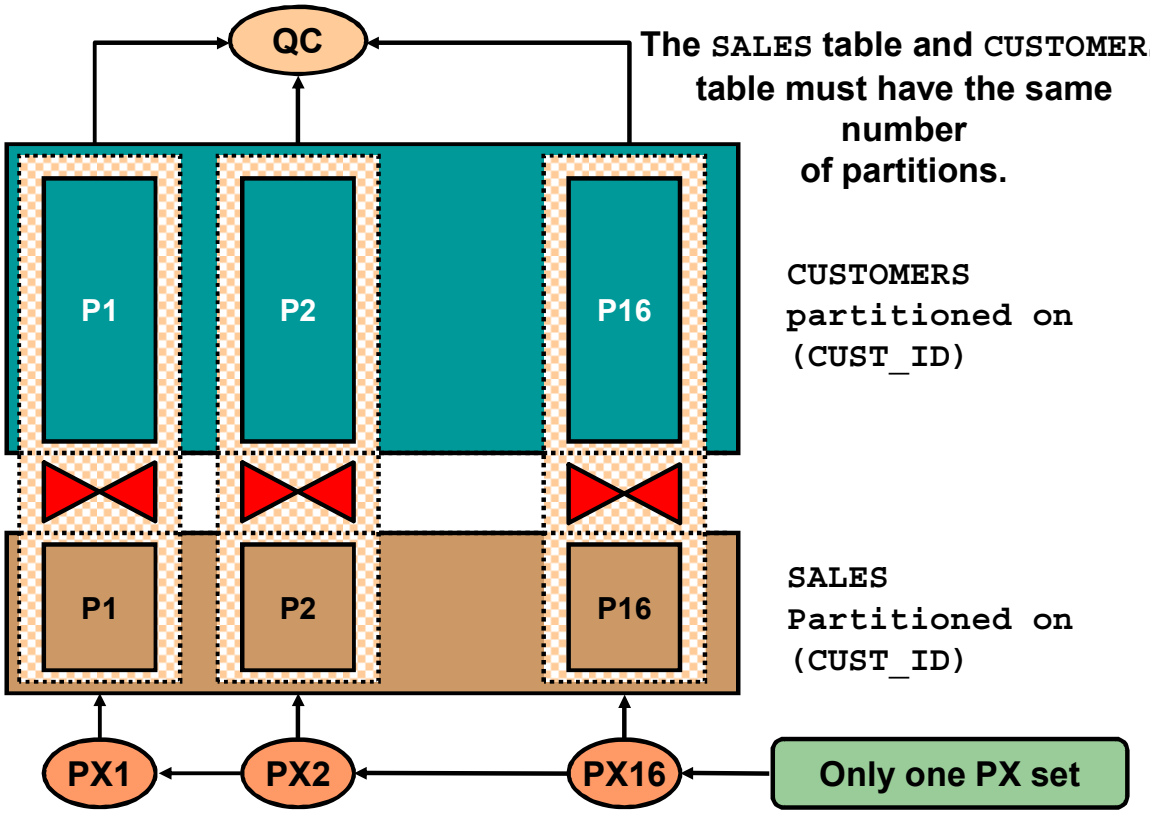
To execute a partial partition-wise join, the database dynamically repartitions the other table based on the partitioning of the reference table. After the other table is repartitioned, the execution is similar to a full partition-wise join.

Parallel joins between nonpartitioned tables require both input tables to be redistributed on the join key. This redistribution operation involves exchanging rows between parallel execution servers.

Note: This is a CPU-intensive operation that can lead to excessive interconnect traffic in Oracle RAC environments.

Partitioning large tables on a join key, either a foreign or primary key, prevents this redistribution every time the table is joined on that key. Of course, if you choose a foreign key to partition the table, which is the most common scenario, then select a foreign key that is involved in many queries.

Example 6: Full Partition-Wise Join



When you execute a full partition-wise join in parallel, the granule of parallelism is a partition, not a block range.

Consequently, the degree of parallelism is limited to the number of partitions. For example, you require at least 16 partitions to set the degree of parallelism of the query to 16.

A full partition-wise join between two four partitions tables equipartitioned on the join key can use at most four PX servers, one per pair of partitions.

Some partition-based operations, such as full partition-wise join, limit the maximum DOP to the number of partitions of the objects involved.

- In serial, this join is performed between pairs of matching hash partitions, one at a time. When one partition pair has been joined, the join of another partition pair begins. The join completes when all partition pairs have been processed.
- In parallel execution, instead of joining one partition pair at a time, partition pairs are joined in parallel by the PX servers. To ensure a good workload distribution, you could have either one of the following:
  - More partitions than the requested DOP
  - Equisize partitions with as many partitions as the requested DOP

One Method to Optimize Parallel Execution on Hash-Partitioned Tables

1. Choose a DOP that is a factor of two (2). **Examples:** 2, 4, 8, 16, 32.
2. Choose a number of partitions that is greater than or equal to 2* DOP.
3. Make sure that the number of partitions is an even number.
4. Choose a stable partition count strategy, which is typically hash, which can be a subpartitioning strategy rather than the main strategy (range - hash is a popular one).
5. Make sure that you do this on the join key between the two large tables you want to join.

Example:

- If DOP = 8 (determined based on concurrency), this indicates that the number of partitions should be equal or greater than 16.
- Number of hash (sub) partitions = 32, which gives each process four partitions to work on. If you have more room on the box, you can easily move the DOP for the query to 16 without repartitioning.

Parallel Execution Example 6: Full Partition-Wise Join

To perform a full PJW, the tables SALES and CUSTOMERS must be equipartitioned.

```
SELECT c.cust_last_name, COUNT(*)
FROM sales s, customers c
WHERE s.cust_id = c.cust_id
  AND s.time_id
     BETWEEN TO_DATE('01-JUL-1999', 'DD-MON-YYYY')
        AND  TO_DATE('01-OCT-1999', 'DD-MON-YYYY')
GROUP BY c.cust_last_name
HAVING COUNT(*) > 100;
```


The slide example shows a query on two tables, SALES and CUSTOMERS, that must be equipartitioned to benefit the parallel full partition-wise join feature. The two tables are joined on their hash-partitioning column, CUST_ID.

Example 6: Full Partition-Wise Join

```
Id|Operation                    |Name     |Rows |Psta|Psto|  TQ  |I-OU|PQ Distr
-------------------------------------------------------------------------------
 0|SELECT STATEMENT             |         |   46|    |    |      |    |
 1| PX COORDINATOR              |         |     |    |    |      |    |
 2|  PX SEND QC (RANDOM)        |:TQ10001 |   46|    |    |Q1,01|P->S|QC(RAND)
 3|   FILTER                    |         |     |    |    |Q1,01|PCWC|
 4|    HASH GROUP BY            |         |   46|    |    |Q1,01|PCWP|
 5|     PX RECEIVE              |         |   46|    |    |Q1,01|PCWP|
 6|      PX SEND HASH           |:TQ10000 |   46|    |    |Q1,00|P->P|HASH
 7|       HASH GROUP BY         |         |   46|    |    |Q1,00|PCWP|
 8|        PX PARTITION HASH ALL|         |59158|  1 | 16 |Q1,00|PCWC|
 9|         HASH JOIN           |         |59158|    |    |Q1,00|PCWP|
10|          TABLE ACCESS FULL  |CUSTOMERS|55500|  1 | 16 |Q1,00|PCWP|
11|          TABLE ACCESS FULL  |SALES    |59158|  1 | 16 |Q1,00|PCWP|
```

Note:

- The line PX BLOCK ITERATOR in previous plans indicated that block range granules were used.
- The line PX PARTITION indicates that partition granules are used.

In the slide example, the two tables are joined on their hash-partitioning columns. This enables the use of a full partition-wise join on their hash-partitioning column, `CUST_ID`. This is translated as `PX PARTITION HASH ALL` row source in the plan table output.

Example

In the following example, `EMP_COMP` and `DEPT_HASH` are also joined on their hash-partitioning columns. This enables the use of a full partition-wise join.

```
CREATE TABLE dept_hash
   PARTITION BY HASH(department_id)
   PARTITIONS 8
   PARALLEL 8
   AS SELECT /*+ PARALLEL */ * FROM departments;
```

```
CREATE TABLE hr.emp_comp
   PARTITION BY RANGE (department_id)
   (PARTITION p1 values less than (1),
    PARTITION p2 values less than (20),
    PARTITION p3 values less than (maxvalue))
   PARALLEL 2
   AS SELECT * FROM hr.employees;

EXPLAIN PLAN FOR
SELECT /*+ PQ_DISTRIBUTE(e, NONE, NONE) ORDERED */
       e.last_name, d.department_name
FROM emp_comp e, dept_hash d
WHERE e.department_id = d.department_id;
```

```
-------------------------------------------------------------------------
| Id|Operation                   |Name      |Pstart|Pstop|PQ Distrib|
-------------------------------------------------------------------------
|  0|SELECT STATEMENT            |          |      |     |          |
|  1| PX COORDINATOR             |          |      |     |          |
|  2|  PX SEND QC (RANDOM)       |:TQ10000  |      |     | QC (RAND)|
|  3|   PX PARTITION HASH ALL    |          |    1 |   3 |          |
|* 4|    HASH JOIN               |          |      |     |          |
|  5|     PX PARTITION RANGE ALL |          |    1 |   5 |          |
|  6|      TABLE ACCESS FULL     |EMP_COMP  |    1 |  15 |          |
|  7|     TABLE ACCESS FULL      |DEPT_HASH |    1 |   3 |          |
-------------------------------------------------------------------------
```

The PX PARTITION HASH row source appears on top of the join row source in the plan table output whereas the PX PARTITION RANGE row source appears over the scan of EMP_COMP. Each parallel server performs the join of an entire hash partition of EMP_COMP with an entire partition of DEPT_HASH.

Operation Values

- PX PARTITION RANGE [ALL]: The partition row source iterates on all RANGE partitions.
- PX PARTITION HASH [ALL]: The partition row source iterates on all HASH partitions.
- If ALL is not specified, the partition boundaries are provided by the values of PARTITION_START and PARTITION_STOP .

Parallel DML

- When to use PDML
  - Refreshing tables in a data warehouse system
  - Creating intermediate summary tables
  - Updating historical tables
  - Running batch jobs
- Decision to parallelize
  - The table being updated or deleted has a `PARALLEL` specification.
  - The `PARALLEL` hint is specified in the DML statement.
  - An `ALTER SESSION ENABLE|FORCE PARALLEL DML` statement has been issued previously during the session.
- Precedence: DOP is determined by the same rules as for the queries.

Parallel DML is useful in a decision support system (DSS) environment where the performance and scalability of accessing large objects are important. Parallel DML complements parallel query in providing you with both querying and updating capabilities for your DSS databases. Parallel DML statements have different locking, transaction, and disk space requirements and may not be suitable for all application models. Parallel DML is disabled at the session level by default.

The overhead of setting up parallelism makes parallel DML operations not feasible for short OLTP transactions. However, parallel DML operations can speed up batch jobs running in an OLTP database.

Note: For `UPDATE` and `DELETE` operations, only the target table to be modified (the only reference object) is involved. Thus, the `UPDATE` or `DELETE PARALLEL` hint specification takes precedence over the parallel declaration specification of the target table.

An `ALTER SESSION FORCE PARALLEL DML PARALLEL` *n* forces a specific degree of parallelism to be in effect, overriding any `PARALLEL` clause associated with subsequent statements.

Restrictions: Parallel processing is not supported for `UPDATE` , `DELETE`, and `MERGE` with temporary tables.

Example 7: Parallel UPDATE Without PDML

```
SQL> ALTER SESSION DISABLE PARALLEL DML;
SQL> EXPLAIN PLAN FOR
  2    UPDATE /*+ PARALLEL (sales,2) */ sales
  3    SET quantity_sold=quantity_sold+100;
```

```
PLAN_TABLE_OUTPUT
-----------------------------------------------------------------
| Id  | Operation              | Name     | Rows  | Bytes |
-----------------------------------------------------------------
|   0 | UPDATE STATEMENT       |          |    14 |    98 |
|   1 |  UPDATE                | SALES    |       |       |
|   2 |   PX COORDINATOR       |          |       |       |
|   3 |    PX SEND QC (RANDOM) | :TQ10000 |    14 |    98 |
|   4 |     PX BLOCK ITERATOR  |          |    14 |    98 |
|   5 |      TABLE ACCESS FULL | SALE     |    98 |       |
-----------------------------------------------------------------
```

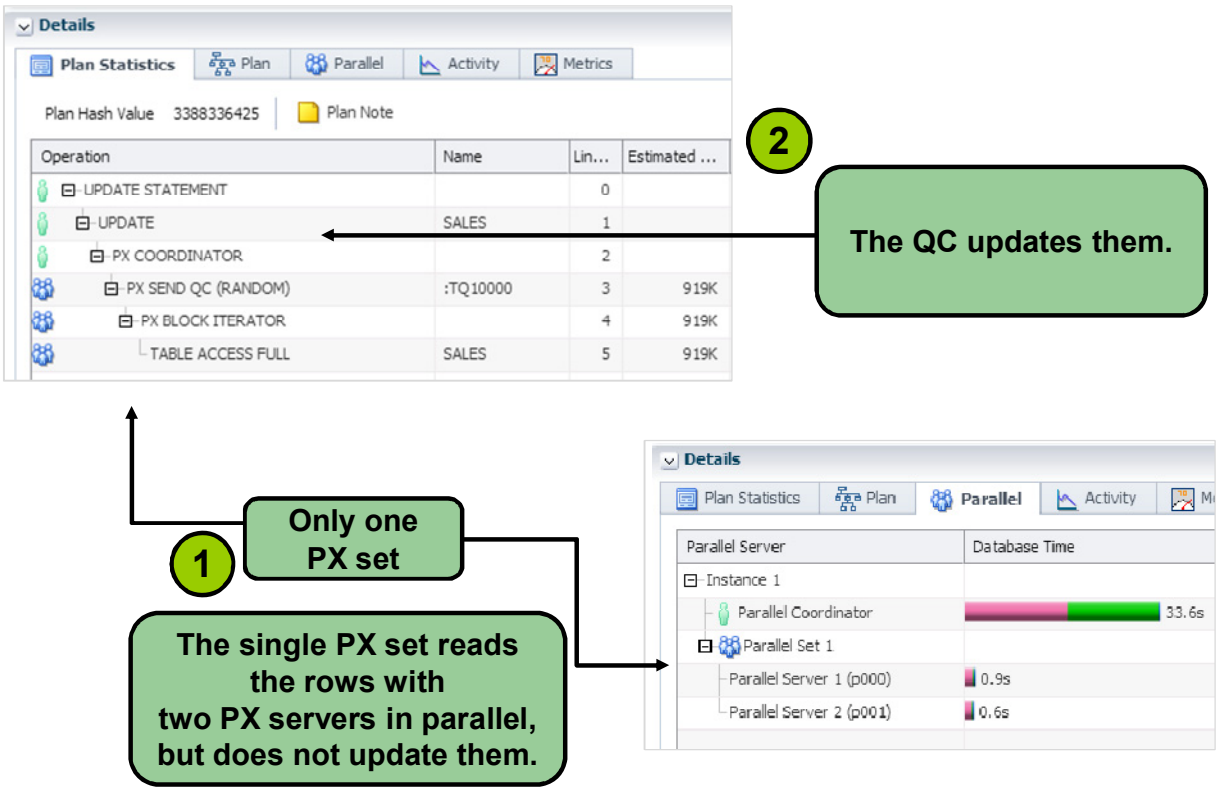
PX set 1

The slide example shows an UPDATE statement with a hint PARALLEL but within a disabled parallel session.

How do we interpret the execution plan?

The UPDATE operation is not parallelized because it is performed outside of the PX coordinator scope. Only the Table Access Full on the SALES table is parallelized.

Example 7: Parallel UPDATE Without PDML



You can access the SQL Monitoring feature in Enterprise Manager Cloud Control 12*c* by navigating to the database home page and clicking the Performance menu item. Scroll down and click **SQL Monitoring**.

The statement shows the execution plan with the QC retrieving rows from the PX server set that performed the Table Access Full on the `SALES` table.

The `UPDATE` is performed after the QC has retrieved all the rows to `UPDATE`.

Example 8: Parallel UPDATE with PDML

```
SQL> ALTER SESSION ENABLE PARALLEL DML;
SQL> EXPLAIN PLAN FOR
  2    UPDATE /*+ PARALLEL (sales,2) */ sales
  3    SET quantity_sold=quantity_sold+100;
```

```
-----------------------------------------------------------------
| Id  | Operation                | Name       | Rows  | Bytes |
-----------------------------------------------------------------
|   0 | UPDATE STATEMENT         |            |    14 |    98 |
|   1 |  PX COORDINATOR          |            |       |       |
|   2 |   PX SEND QC (RANDOM)    | :TQ10001   |    14 |    98 |
|   3 |    INDEX MAINTENANCE     | SALES      |       |       |
|   4 |     PX RECEIVE           |            |    14 |    98 |
|   5 |      PX SEND RANGE       | :TQ10000   |    14 |    98 |
|   6 |       UPDATE             | SALES      |       |       |
|   7 |        PX BLOCK ITERATOR |            |    14 |    98 |
|   8 |         TABLE ACCESS FULL| SALES      |    14 |    98 |
-----------------------------------------------------------------
```

PX set 2

PX set 1

Note: The DBMS_PARALLEL_EXECUTE package enables the user to incrementally update table data in parallel as an alternative to parallel DML.

The DBMS_PARALLEL_EXECUTE package is an alternative to parallel DML and enables the user to incrementally update table data in parallel, in two high-level steps:

1. Group sets of rows in the table into smaller-sized chunks.
2. Run a user-specified statement on these chunks in parallel, and commit when finished processing each chunk.

This technique is recommended whenever you are updating a large amount of data. Its advantages are:

- You lock only one set of rows at a time, for a relatively short time, instead of locking the entire table.
- You do not lose work that has been performed if something fails before the entire operation finishes.
- You reduce rollback space consumption.
- You improve performance.

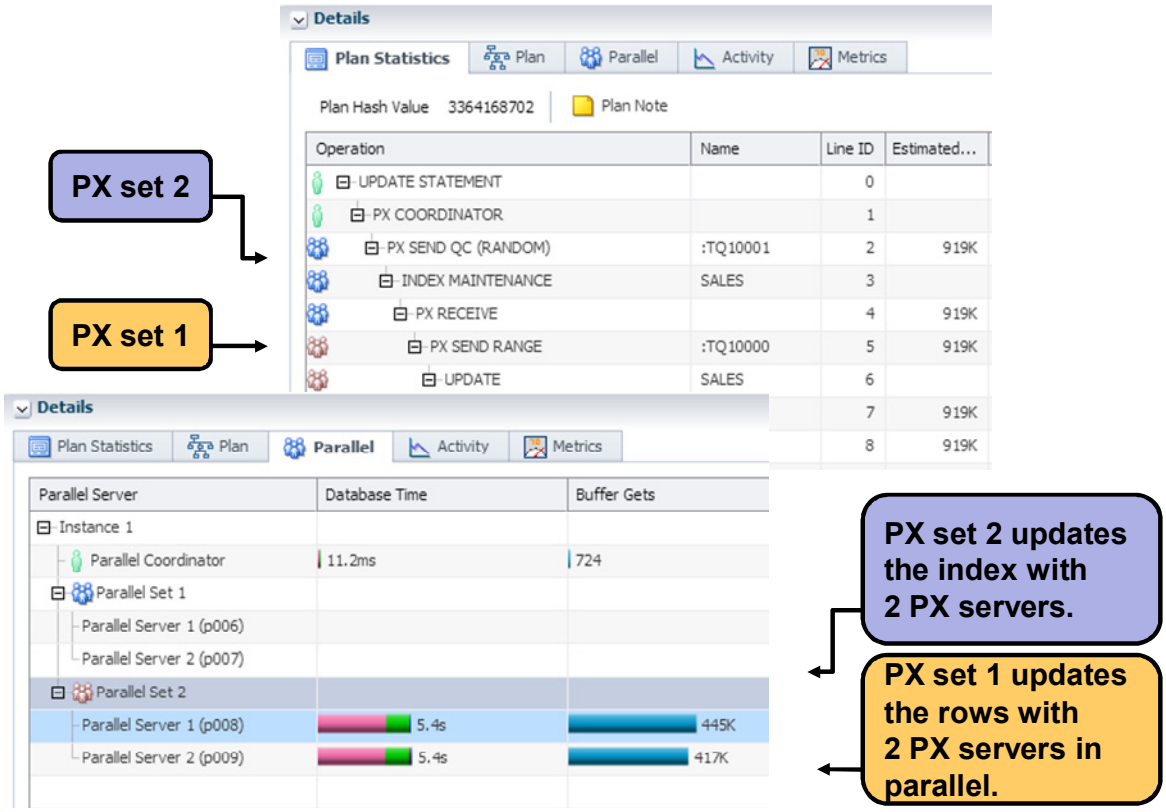
This package introduces the notion of parallel execution task. This task groups the various steps associated with the parallel execution of a PL/SQL block, which typically updates table data.

This example shows the most common usage of this package. After calling the `RUN_TASK` procedure, it checks for errors and reruns in case of an error.

```
DECLARE
  l_sql_stmt VARCHAR2(1000);
  l_try NUMBER;
  l_status NUMBER;
BEGIN
  -- Create the TASK
  DBMS_PARALLEL_EXECUTE.CREATE_TASK ('mytask');

  -- Chunk the table by ROWID
  DBMS_PARALLEL_EXECUTE.CREATE_CHUNKS_BY_ROWID('mytask', 'HR',
                                        'EMPLOYEES', true, 100);
   -- Execute the DML in parallel
  l_sql_stmt := 'update /*+ ROWID (dda) */ EMPLOYEES e
                 SET e.salary = e.salary + 10
                 WHERE rowid BETWEEN :start_id AND :end_id';
  DBMS_PARALLEL_EXECUTE.RUN_TASK('mytask', l_sql_stmt,
DBMS_SQL.NATIVE, parallel_level => 10);

  -- If there is an error, RESUME it for at most 2 times.
  L_try := 0;
  L_status := DBMS_PARALLEL_EXECUTE.TASK_STATUS('mytask');
  WHILE(l_try < 2 and L_status !=DBMS_PARALLEL_EXECUTE.FINISHED)
  LOOP
    L_try := l_try + 1;
    DBMS_PARALLEL_EXECUTE.RESUME_TASK('mytask');
    L_status := DBMS_PARALLEL_EXECUTE.TASK_STATUS('mytask');
  END LOOP;

  -- Done with processing; drop the task
  DBMS_PARALLEL_EXECUTE.DROP_TASK('mytask');
END;
/
```

Note: For more information, see the My Oracle Support document titled "*Using DBMS_PARALLEL_EXECUTE to Update Large Tables in Parallel (Doc ID 1066555.1).*"

Example 8: Parallel `UPDATE` with PDML



The statement shows the execution plan with two PX server sets. One PX server set is collecting and updating rows while the other is updating the index.

Each PX server set has two PX servers to work in parallel.

Note: The work described in an execution plan is performed from the inner level to the outer level. In the slide diagrams here and on the preceding pages, PX set 1 was arbitrarily labeled as the inner set of PX processes and PX set 2 was labeled as the outer set of PX processes. In Enterprise Manager, the Monitored SQL Execution tool has arbitrarily labeled them in a manner opposite to that done earlier. This is not significant.

Example 9: PDML INSERT

- In the following direct-load insert operation, the query and load portions of the operation are combined into each query server according to the PQ_DISTRIBUTE hint.

```
SQL> ALTER SESSION ENABLE PARALLEL DML;
SQL> EXPLAIN PLAN FOR
  2    INSERT /*+ APPEND PARALLEL (sales_target,2)
                  PQ_DISTRIBUTE(sales_target, NONE) */
  3    INTO sales_target
  4    SELECT * FROM sh.sales;
```

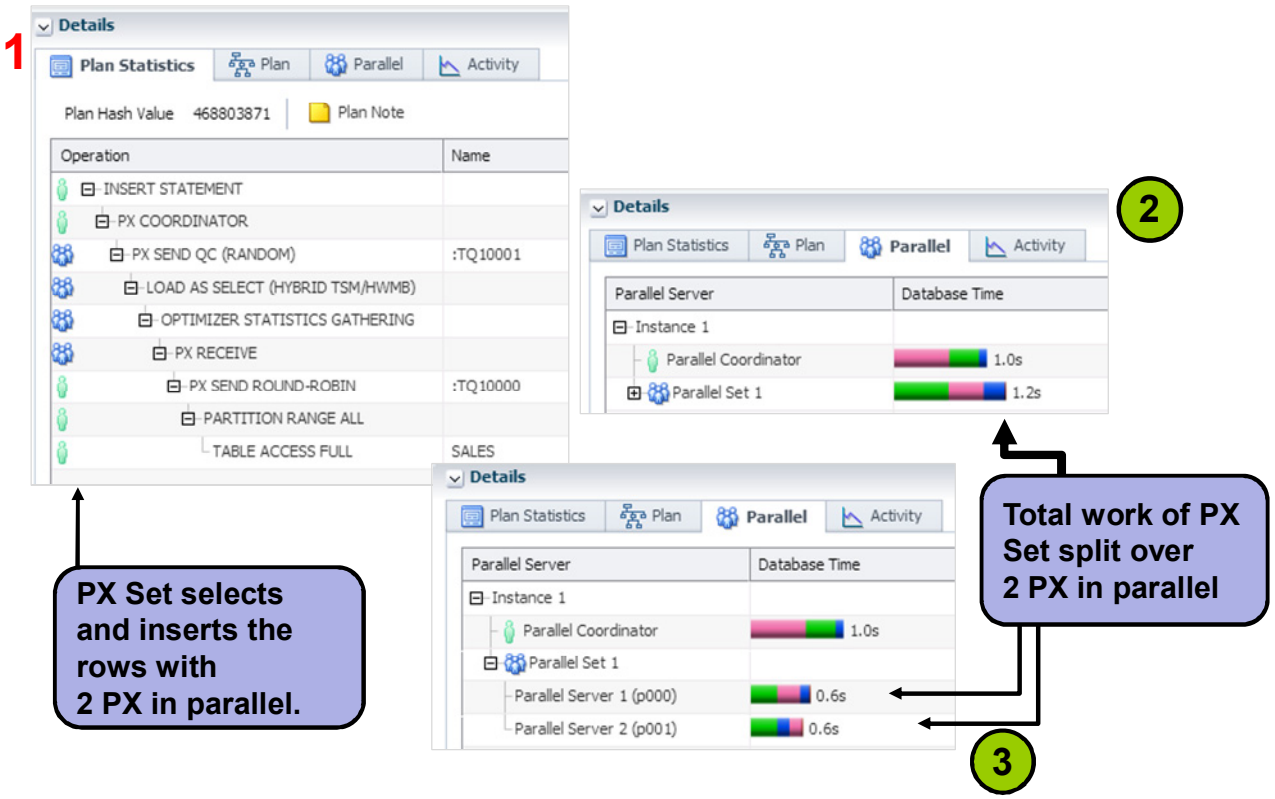
- PQ_DISTRIBUTE hint: Controls the distribution of rows to direct how rows should be distributed between the producer (query) and the consumer (load) servers.

Parallel DML Direct-Path INSERT Considerations

- **Space Considerations:** Direct-path INSERT requires more space than conventional-path INSERT. Parallel direct-path INSERT into nonpartitioned tables requires even more space, because it creates a temporary segment for each degree of parallelism. This behavior can vary depending on the hint that was used. If the nonpartitioned table is not in a locally managed tablespace in automatic segment-space management mode, you can modify the values of the NEXT and PCTINCREASE storage parameters and the MINIMUM EXTENT tablespace parameter to provide sufficient (but not excess) storage for the temporary segments. After the direct-path INSERT operation is complete, you can reset these parameters to settings more appropriate for serial operations.
- **Index Maintenance:** Oracle Database performs index maintenance at the end of direct-path INSERT operations on tables (partitioned or nonpartitioned) that have indexes. This index maintenance is performed by the parallel execution servers for parallel direct-path INSERT or by the single process for serial direct-path INSERT. You can avoid the performance impact of index maintenance by making the index unusable before the INSERT operation and then rebuilding it afterward.

Note: When executing in parallel, the APPEND hint is the default option for the INSERT … INTO … SELECT … FROM form.

Example 9: PDML `INSERT`



`INSERT` operation ensures atomicity of the transaction, even when run in parallel mode (not SQL*Loader). If errors occur during parallel direct-path loads with SQL*Loader, the load completes, but some indexes could be marked `UNUSABLE` at the end of the load. Parallel direct-path `INSERT`, in contrast, rolls back the statement if errors occur during index update.

- Logging Mode: Direct-path `INSERT` lets you choose whether to log redo and undo information during the insert operation.
- `PQ_DISTRIBUTE` distribution values for load: `NONE` (no distribution). That is, the query and load operation are combined into each query server. All servers will load all partitions. This lack of distribution is useful to avoid the overhead of distributing rows where there is no skew. Skew can occur due to empty segments or a predicate in the statement that filters out all rows evaluated by the query. If skew occurs due to using this method, then use either `RANDOM` or `RANDOM_LOCAL` distribution instead.

  Note: Use this distribution with care. Each partition loaded requires a minimum of 512 KB per process of PGA memory. If you also use compression, then approximately 1.5 MB of PGA memory is consumed per server.

- `PARTITION`: This method uses the partitioning information of `tablespec` to distribute the rows from the query servers to the load servers. Use this distribution method when it is not possible or desirable to combine the query and load operations, when the number of partitions being loaded is greater than or equal to the number of load servers, and the input data will be evenly distributed across the partitions being loaded—that is, there is no skew.
- `RANDOM`: This method distributes the rows from the producers in a round-robin fashion to the consumers. Use this distribution method when the input data is highly skewed.
- `RANDOM_LOCAL`: This method distributes the rows from the producers to a set of servers that is responsible for maintaining a given set of partitions. Two or more servers can be loading the same partition, but no servers are loading all partitions. Use this distribution method when the input data is skewed and combining query and load operations is not possible due to memory constraints.

Parallel DDL

- PDDL for nonpartitioned tables and indexes:
  - CREATE INDEX
  - CREATE TABLE AS SELECT
  - ALTER INDEX REBUILD
- PDDL for partitioned tables and indexes:
  - CREATE INDEX
  - CREATE TABLE AS SELECT
  - ALTER TABLE MOVE/SPLIT/COALESCE PARTITION
  - ALTER INDEX REBUILD/SPLIT PARTITION

```
SQL> ALTER SESSION ENABLE PARALLEL DDL;
SQL> EXPLAIN PLAN FOR
  2    CREATE TABLE summary (C1, AVGC2, SUMC3) PARALLEL 5
  4    AS SELECT prod_id, AVG(quantity_sold), SUM(amount_sold)
  5    FROM SALES GROUP BY (prod_id);
```

ORACLE

The CREATE TABLE AS SELECT (CTAS) statement lets you execute the query from another table or set of tables in parallel, and create operations in parallel. This can be extremely useful in the creation of summary or rollup tables.

Note

- Clustered tables cannot be created and populated in parallel.
- PDDL is enabled by default for a session and it is not necessary to enable it before doing a PDDL statement, as shown in the slide, unless for some reason the default value has been overridden.

Example 10: PDDL CTAS

```
CREATE TABLE customers (c NUMBER, c2 date) PARALLEL 8
AS SELECT /*+ PARALLEL 4 */* FROM other_customers;
```

```
CREATE /*+ PQ_DISTRIBUTE(o_sales, PARTITION) */ TABLE o_sales
PARALLEL 16 PARTITION BY HASH (prod_id) PARTITIONS 512
AS SELECT * FROM sales;
```

- The optimizer uses the partitioning of target_table O_SALES to distribute the rows.

```
|Id |Operation                    |Name      |Pstart|Pstop|IN-OUT|PQ Distrib
-------------------------------------------------------------------------------
| 0 |CREATE TABLE STATEMENT       |          |      |     |      |
| 1 | PX COORDINATOR              |          |      |     |      |
| 2 |  PX SEND QC (RANDOM)        |:TQ10001  |      |     | P->S | QC (RAND)
| 3 |   LOAD AS SELECT            |O_SALES   |      |     | PCWP |
| 4 |    PX RECEIVE               |          |      |     | PCWP |
| 5 |     PX SEND PARTITION (KEY)|:TQ10000  |      |     | P->P | PART(KEY)
| 6 |      PX BLOCK ITERATOR      |          | 1    |   28| PCWC |
| 7 |       TABLE ACCESS FULL     |SALES     | 1    |   28| PCWP |
-------------------------------------------------------------------------------
```

ORACLE

PARALLEL Hint: SELECT

- Can be parallelized only if one of the following conditions is satisfied:
  - The query includes a parallel hint specification (PARALLEL or PARALLEL_INDEX).
  - The CREATE part has a PARALLEL clause .
  - The schema objects referred to in the query have a PARALLEL declaration associated with them.
- At least one table specified in the query requires either a full table scan or an index range scan spanning multiple partitions.
- Gets a DOP determined by one of the following rules:
  - The query part uses the values specified in the PARALLEL clause of the CREATE part.
  - If the PARALLEL clause is not specified, the default DOP is calculated (SUM(CPU_COUNT for each instance) * PARALLEL_THREADS_PER_CPU) for the system.
  - If the CREATE is serial, then the DOP is determined by the query.

  Note that any values specified in a hint for parallelism are ignored.

PARALLEL Hint: CREATE TABLE

- Can be parallelized only by a PARALLEL clause or an ALTER SESSION FORCE PARALLEL DDL statement
- Parallelizes the scan operation also if possible. The scan operation cannot be parallelized if, for example, the following occurs:
  - The SELECT clause has a NO_PARALLEL hint.
  - The operation scans an index of a nonpartitioned table.
- Gets a DOP for the CREATE operation, and for the SELECT operation, by the PARALLEL clause of the CREATE statement, unless it is overridden by an ALTER SESSION FORCE PARALLEL DDL statement. If the PARALLEL clause does not specify the DOP, the DOP is calculated (SUM(CPU_COUNT for each instance) * PARALLEL_THREADS_PER_CPU) for the system.

When the CREATE operation is not parallelized, the SELECT can be parallelized if one of the following statements is true:

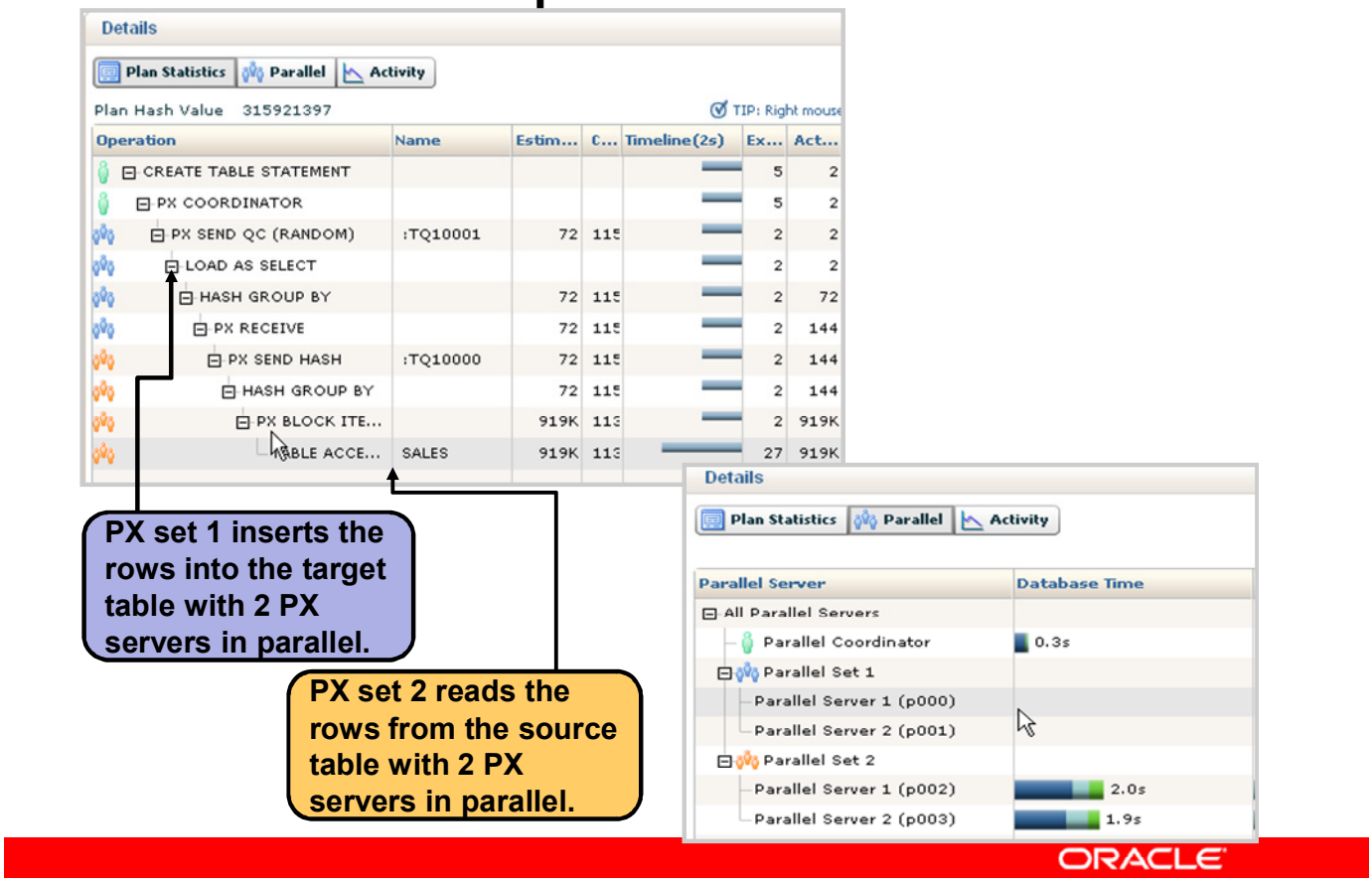
- The SELECT has a PARALLEL hint.
- The selected table (or partitioned index) has a parallel declaration.

PQ_DISTRIBUTE Hint

The hint lets you control the distribution of rows to direct how rows should be distributed between the producer (query) and the consumer (load).

The example in the slide specifies the PARTITION distribution method.

Example 10: PDDL CTAS

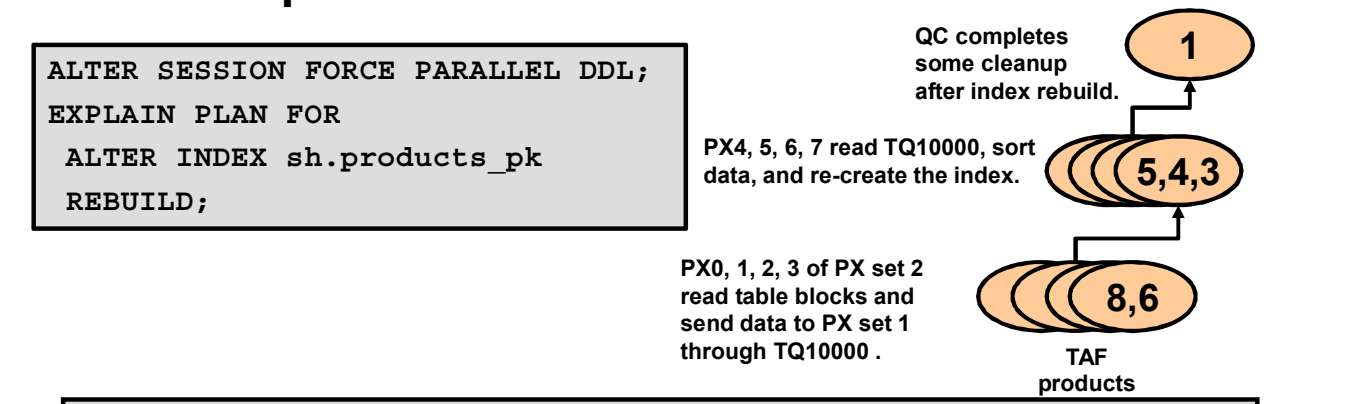


The slide statement shows the execution plan of a `CREATE TABLE AS SELECT` with two PX server sets. The PX server set 1 is collecting rows from `SALES` table with two PX servers in parallel while the other set is inserting the rows into the target table.

Each PX server set has two PX servers to work in parallel, as shown in the Parallel tab.

Example 11: PDDL ALTER INDEX REBUILD

```
ALTER SESSION FORCE PARALLEL DDL;
EXPLAIN PLAN FOR
 ALTER INDEX sh.products_pk
 REBUILD;
```



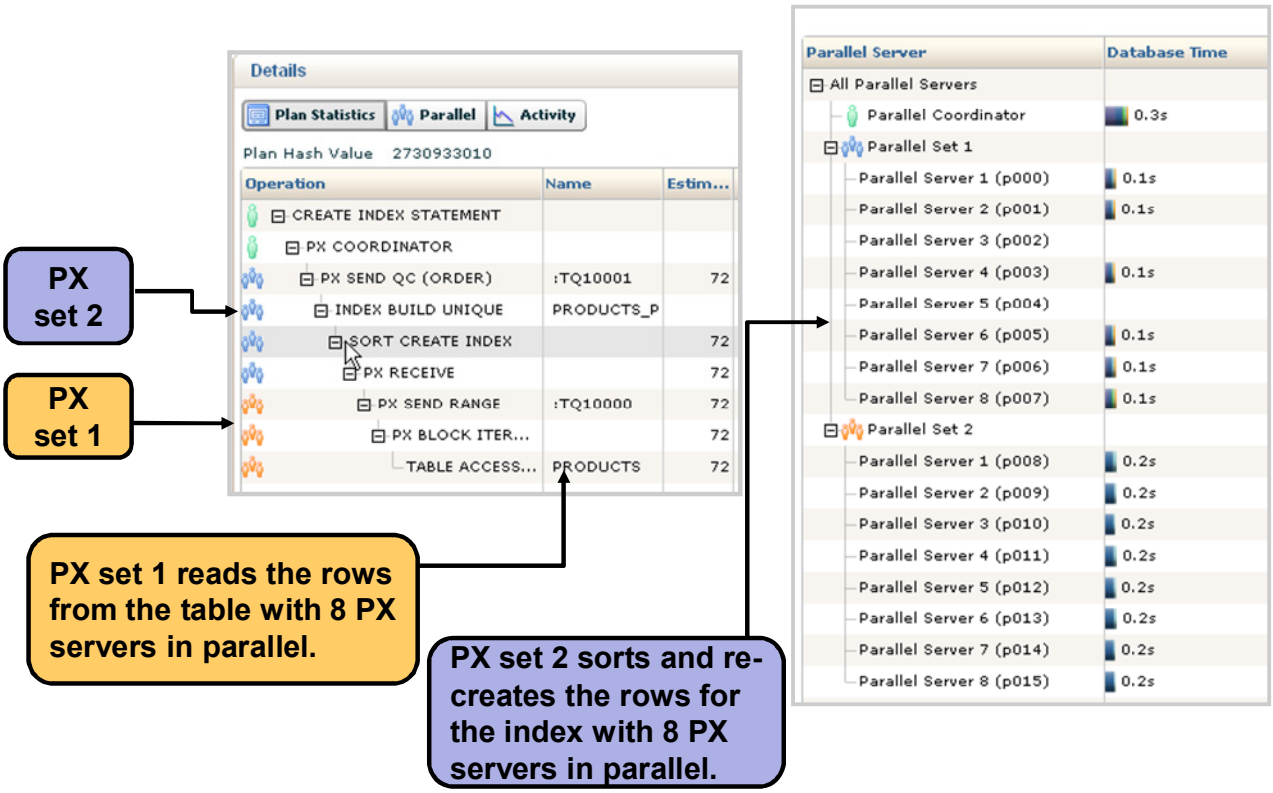
Id	Operation	Name	TQ	IN-OUT	PQ Distrib
0	ALTER INDEX STATEMENT				
1	PX COORDINATOR				
2	PX SEND QC (ORDER)	:TQ10001	Q1,01	P->S	QC (ORDER)
3	INDEX BUILD UNIQUE	PRODUCTS_PK	Q1,01	PCWP	
4	SORT CREATE INDEX		Q1,01	PCWP	
5	PX RECEIVE		Q1,01	PCWP	
6	PX SEND RANGE	:TQ10000	Q1,00	P->P	RANGE
7	PX BLOCK ITERATOR		Q1,00	PCWC	
8	TABLE ACCESS FULL	PRODUCTS	Q1,00	PCWP	

PX set 2

PX set 1

The scan operation for `ALTER INDEX ... REBUILD` (nonpartitioned), `ALTER INDEX ... REBUILD PARTITION`, and `CREATE INDEX` has the same parallelism as the `REBUILD` or `CREATE` operation and uses the same DOP. If the DOP is not specified for `REBUILD` or `CREATE`, the default DOP is calculated (SUM `(CPU_COUNT` for each instance`)` * `PARALLEL_THREADS_PER_CPU`) for the system.

Example 11: PDDL `ALTER INDEX REBUILD`



The slide statement shows the execution plan of a `ALTER INDEX REBUILD` with two PX server sets. The PX server set 1 is collecting rows from the `PRODUCTS` table with eight PX servers in parallel while the other set is creating the index `PRODUCTS_P`.

Each PX server set has eight PX servers to work in parallel, as shown in the Parallel tab.

Summary

In this lesson, you should have learned to:

- Set the manual DOP implementation
- Explain a single parallel table scan
- Explain parallel execution with `GROUP BY`
- Explain parallel execution with `ORDER BY`
- Describe parallel join execution
- Describe parallel join execution with bloom filters
- Explain parallel execution and the advantages of partition-wise join
- Describe bulk DML and parallel processing
- Describe DDL statements and parallel processing
- Explain how indexes rebuild in parallel

ORACLE

5

Simplified Auto DOP

Objectives

After completing this lesson, you should be able to:

- Describe the benefits of using Auto DOP
- Explain the parameters of Auto DOP
- Explain when to use the Auto DOP
- Explain how to use the Auto DOP

The Generic Problem

- Parallel servers are a limited resource.
  - The limit is specified by `PARALLEL_MAX_SERVERS`.
  - Too many concurrent parallel statements can overwhelm the system.
- When there are no more parallel servers:
  - Critical statements may run serially
  - DBAs are forced to underutilize the system or manually schedule large queries during off hours
  - And when parallel servers free up, there is no way to boost the DOP of running statements

The slide lists the issues that appear when DBAs have to work manually with parallelization.

How to Enhance Parallel Execution

- Challenges in earlier releases of Oracle Database
  - It is difficult to determine the ideal DOP for each table without manual tuning.
  - One DOP does not fit all queries touching an object.
  - Not enough PX server processes can result in statement running serial.
  - Too many PX server processes can overwhelm the system.
  - They only use I/O resources.
- From Oracle Database 11*g* Release 2 and above, Oracle:
  - Automatically decides whether a statement executes in parallel and which DOP to use
  - Can execute immediately or is queued
  - Takes advantage of aggregated cluster memory or not

ORACLE

Releases Prior to Oracle 11*g* Release 2

Oracle provides various mechanisms to enable parallelism for a given SQL command. All of these mechanisms are manual and require intimate knowledge of both the SQL and the workload.

It is a burden for the DBA to decide on the degree of parallelism (DOP) of objects such as tables and indexes. This attribute is used to indicate the DOP of any query touching the objects. For a DBA, it is, therefore, very hard to settle on a good DOP that performs well in any circumstances for any queries on any object sizes. Therefore, this DOP is at best a compromise.

Because of the difficulty of enabling parallelism and the systemwide impact of specifying a DOP of an object (it affects all the statements touching it), the DBA might restrict the use of parallelism to very narrow and highly controlled workloads.

From Oracle Database 11*g* Release 2 and above:

With automatic parallelism, the database server compensates for wrong or missing user settings for parallel execution, ensuring more optimal resource consumption and overall system behavior.

Note: This feature automatically determines the DOP for the parallel query execution client. Other clients such as parallel recovery and parallel replication are not affected.

Enabling Auto Degree of Parallelism

Instance or session parameter `PARALLEL_DEGREE_POLICY`

- `MANUAL`
  - The DBA manually specifies all aspects of parallelism.
  - No new features are enabled.
- `LIMITED`
  - Auto DOP restricted for queries with tables decorated with PARALLEL (if an explicit DOP is specified, use that one)
  - No Statement Queuing, no In-Memory Parallel Execution
- `AUTO`
  - All qualifying statements are subject to executing in parallel or not.
  - DOP is automatically computed.
  - DOP set on tables is ignored.
  - Statements can be queued.
  - In-memory PX is available.

`MANUAL` Mode

`MANUAL` mode exists for backward compatibility. This is the default.

In this mode, it is the responsibility of a DBA to specify the degree of parallelism with object properties' name–value pairs, if a statement is to be executed in parallel. Statements touching objects with parallel properties' name–value pairs are executed with a DOP derived directly from the properties' name–value pairs.

`LIMITED` Mode

This mode enables automatic degree of parallelism for those statements that access tables or indexes decorated explicitly with the `PARALLEL` clause, but statement queuing and In-Memory Parallel Execution are disabled.

Tables and indexes that have a degree of parallelism explicitly specified with a value use that degree of parallelism.

`AUTO` Mode

It enables automatic degree of parallelism, statement queuing, and in-memory parallel execution.

New PARALLEL_DEGREE_POLICY Value

Instance or session parameter PARALLEL_DEGREE_POLICY

- ADAPTIVE
  - All qualifying statements are subject to executing in parallel or not.
  - DOP is automatically computed.
  - DOP set on tables is ignored.
  - Statements can be queued.
  - In-memory PX is available.
  - Performance feedback is enabled (New).

Adaptive Mode

This value enables automatic degree of parallelism, statement queuing and in-memory parallel execution, similar to the AUTO value. In addition, performance feedback is enabled. Performance feedback helps to improve the degree of parallelism automatically chosen for repeated SQL statements. After the initial execution of a statement, the degree of parallelism chosen by the optimizer is compared to the degree of parallelism computed based on the actual execution performance. If it varies significantly, then the statement is marked for re-parse and the initial execution performance statistics (for example, CPU-time) are provided as feedback for subsequent executions. The optimizer uses the initial execution performance statistics to better determine a degree of parallelism for subsequent executions.

Auto DOP Requirements

1. Require optimizer statistics and the cost of scan operations
   - No statistics:
     - Cannot appropriately decide between parallelizing and not parallelizing
   - Can lead to:
     - Incorrect decision based on incorrect costing based on incorrect information
     - Too high or too low a DOP
     - Too much queuing if DOP overestimated
2. Require hardware characteristics including I/O calibration
   - No I/O statistics:
     - Automatic DOP uses default value of 200MB/s.
     - Automatic DOP default value may be inaccurate.
   - Solution: Run I/O calibration with Resource Manager.

When `PARALLEL_DEGREE_POLICY` is set to `AUTO`, Oracle Database determines whether the statement should run in parallel based on the cost of the operations in the execution plan and the hardware characteristics. The hardware characteristics include I/O calibration statistics, so these statistics must be gathered; otherwise Oracle Database does not use the automatic degree policy feature.

If I/O calibration is not run to gather the required statistics, automatic DOP will continue and use a default value of 200MB/s. This value may not be the best value for the hardware. In previous releases before Oracle Database 12*c,* it would result in an error and the explain plan output would include the following text in its notes:

```
automatic DOP: skipped because of IO calibrate statistics are
missing
```

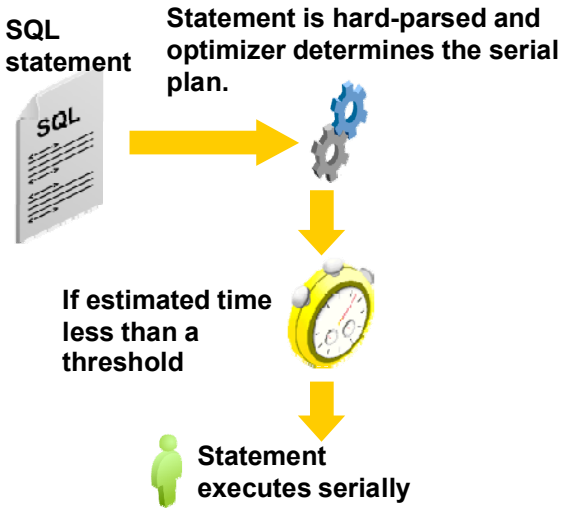
I/O calibration statistics can be gathered with the PL/SQL `DBMS_RESOURCE_MANAGER.CALIBRATE_IO` procedure. I/O calibration is a one-time action if the physical hardware does not change.

Note: For more information about this subject, see the MyOracle Support bulletin titled "*Automatic Degree of Parallelism in11.2.0.2* (Doc ID 1269321.1)." This document is still relevant for Oracle Database 12*c*.

Auto DOP Algorithm

Enhancement addressing:

- Difficult to find the ideal DOP for each table without manual tuning
- One DOP not necessarily relevant for all queries touching an object



Note: The threshold is set in instance parameter `PARALLEL_MIN_TIME_THRESHOLD` (default = 10s).

This diagram illustrates the high-level design of the automatic DOP determination.

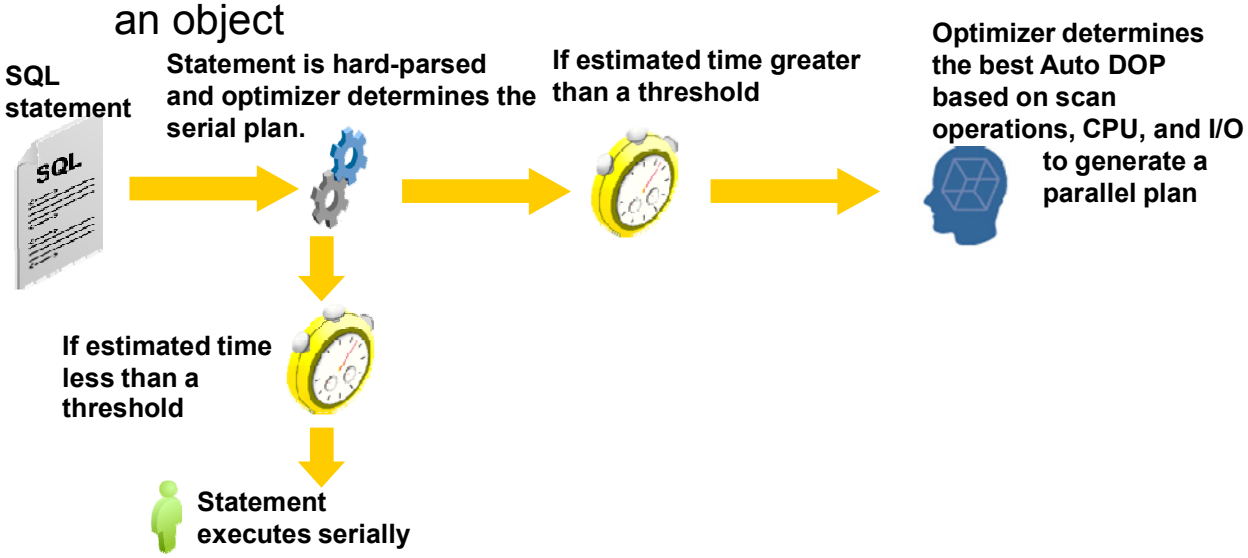
In automatic degree of parallelism, a serial plan is first compiled. During the serial plan compilation, the cost of the serial plan is obtained, in terms of estimated execution elapsed time.

If the serial elapsed time is found to be smaller than the threshold, determined by the parameter `PARALLEL_MIN_TIME_THRESHOLD`, the statement is executed in serial.

Auto DOP Algorithm

Enhancement addressing:

- Difficult to find the ideal DOP for each table without manual tuning
- One DOP not necessarily relevant for all queries touching an object



Note: The threshold is set in instance parameter `PARALLEL_MIN_TIME_THRESHOLD` (default = 10s).

This diagram illustrates the high-level design of the automatic DOP determination.

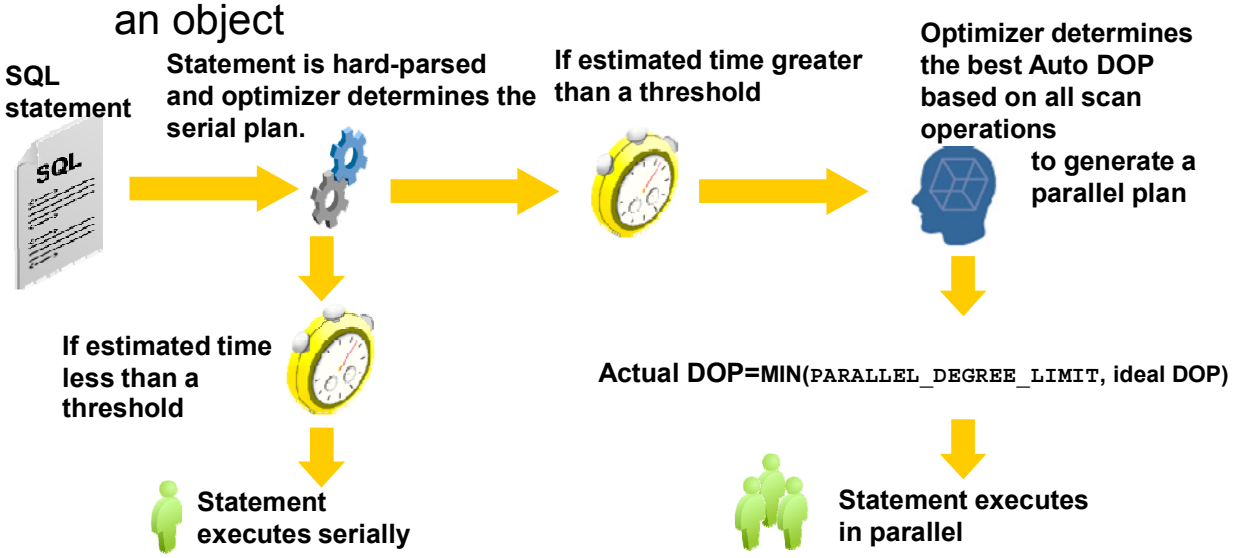
If the serial elapsed time is found to be greater than the threshold, a parallel plan is compiled with the best DOP computed between 2 and the degree limit set by the parameter `PARALLEL_DEGREE_LIMIT`.

The optimizer automatically determines the DOP based on the resource required for all scan operations (full table scan, index fast full scan, and so on) and considers both CPU and I/O.

Auto DOP Algorithm

Enhancement addressing:

- Difficult to find the ideal DOP for each table without manual tuning
- One DOP not necessarily relevant for all queries touching an object



Note: The threshold is set in instance parameter `PARALLEL_MIN_TIME_THRESHOLD` (default = 10s).

This diagram illustrates the high-level design of the automatic DOP determination. However, the optimizer caps the actual DOP for a statement with the default Auto DOP (`sum(CPU_COUNT for each instance)` * `PARALLEL_THREADS_PER_CPU`), to ensure that parallel processes do not flood the system.

Controlling Auto DOP Without Queuing

- PARALLEL_DEGREE_POLICY
  - Controls whether Auto DOP will be used
  - Set to MANUAL by default to disable Auto DOP
  - Set to LIMITED to enable Auto DOP without Queuing or In-Memory PX
- PARALLEL_MIN_TIME_THRESHOLD
  - Controls which statements are candidates for parallelism
  - Default is AUTO, by default set to 10 seconds
- PARALLEL_DEGREE_LIMIT
  - Controls maximum DOP per statement
  - Set to CPU, by default meaning DEFAULT DOP

New Instance Parameters	Allowable Values	Default
PARALLEL_DEGREE_POLICY	MANUAL,LIMITED,AUTO,ADAPTIVE	MANUAL
PARALLEL_MIN_TIME_THRESHOLD	Any number > 0, AUTO	AUTO
PARALLEL_DEGREE_LIMIT	Any number > 1, CPU, IO	CPU

Using PARALLEL_MIN_TIME_THRESHOLD

When PARALLEL_DEGREE_POLICY is left at MANUAL, PARALLEL_MIN_TIME_THRESHOLD and PARALLEL_DEGREE_LIMIT do not have any effect on the compilation or execution of a statement, unless a hint is specified in the statement.

One new concept related to simplified parallelism is *minimum threshold for parallelism*. Even if a SQL statement is deemed a candidate for automatic determination of parallelism, does the parallel overhead justify the cost of this SQL going parallel?

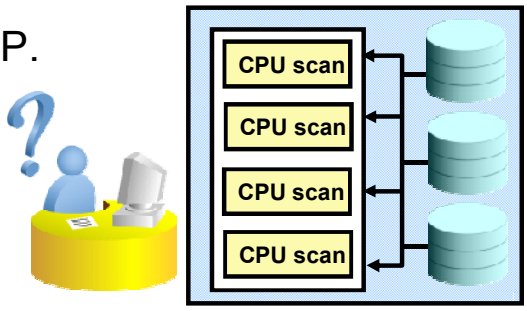
The PARALLEL_MIN_TIME_THRESHOLD parameter limits parallelism to only SQL statements for which it is worth the effort. Any SQL that has the computed elapsed time below this threshold will execute in serial.

This new parameter indicates the threshold (expressed in seconds) that limits the scope of automatic parallel execution. Only if the estimated serial execution elapsed time of a statement is greater than the threshold will the statement be a candidate for automatic parallel execution. It accepts any numerical value greater than 0 or a keyword AUTO, set to 10 seconds by default.

Note: PARALLEL_MIN_PERCENT works as in past releases if the estimated elapsed time exceeds PARALLEL_MIN_TIME_THRESHOLD.

Using PARALLEL_DEGREE_LIMIT

- The maximum degree of parallelism for a statement is capped by the default DOP.
- In some cases, this DOP might be too high.
- PARALLEL_DEGREE_LIMIT enables you to specify a maximum DOP.



Maximum DOP?

New Parameters	Allowable Values	Default
PARALLEL_DEGREE_LIMIT	Any number > 1, CPU, IO	CPU

Another concept related to simplified parallelism is "*maximum degree of parallelism*." By default, the maximum degree of parallelism for a statement is capped by the default DOP. In some cases, this DOP might be too high. There is a need to have a limit on the maximum degree of parallelism for all parallel statements in the system to prevent overloading of the system. The PARALLEL_DEGREE_LIMIT parameter allows you to set the maximum DOP.
The allowable values for the parameter are any numerical value greater than or equal to 1, CPU, or IO.
Users are also allowed to set the limit to a fixed value.
When PARALLEL_DEGREE_LIMIT is set to:

- IO: The maximum degree of parallelism that the optimizer can use is limited by the I/O capacity of the system. The value is calculated by dividing the total system throughput by the maximum I/O bandwidth per process. You must run the DBMS_RESOURCE_MANAGER.CALIBRATE_IO procedure on the system in order to use the IO setting. This procedure will calculate the total system throughput and the maximum I/O bandwidth per process.
- CPU (the default value for this parameter): The threshold is calculated as follows:

```
threads_per_cpu * total number of CPUs in the cluster
```

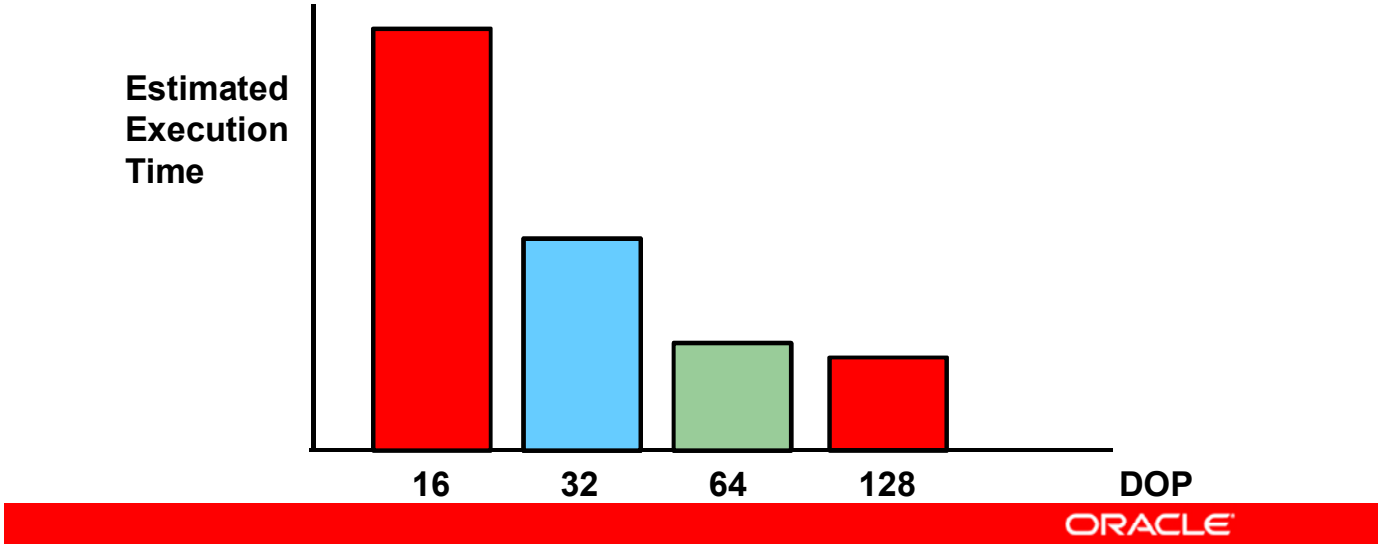
Remark: With PARALLEL_DEGREE_LIMIT defaulted to CPU, Oracle still defaults a maximum DOP for intraparallelism operations.

Note: If PARALLEL_DEGREE_POLICY is set to MANUAL (the default), the maximum DOP a statement can have is limited by PARALLEL_MAX_SERVERS, as in previous releases. If PARALLEL_DEGREE_POLICY is set to AUTO or LIMITED, the maximum DOP a statement can have is limited by PARALLEL_DEGREE_LIMIT, which is by default DEFAULT_DOP. It is still possible, however, to use a hint to force a greater DOP for a statement.

Optimal DOP Determination

Assume that the optimizer cost for scan operations estimates a DOP of 32:

- The optimizer tests whether, with twice the resources, the DOP executes twice as fast.
- If so, the optimizer chooses the higher DOP.
- If not, the optimizer keeps the original DOP.



The degree of parallelism is derived from the elapsed time of the most costly operation during the parallel compilation.

To derive the DOP, the optimizer uses a unit of work that can be performed by a PX server. This unit is expressed in seconds. (The default is 10s.)

To have access to another PX server, an operation must require at least that unit of work above the threshold. As an example, if the unit is 10s and the threshold is also 10s and the estimated elapsed time of the operation is 43s, the ideal degree of parallelism will be 4.

The optimizer still uses response time to compute the best parallel plan. The DOP is derived from the expected response time of a single operation. The unit of work that a PX server should have to efficiently cover for its cost is the quotient used to derive the DOP.

DOP = Elapsed Time(operation) / parallel time unit

However, the optimizer caps the actual DOP for a statement with the `PARALLEL_DEGREE_LIMIT` defined.

All Auto DOP Parameters Switched On

1. After Auto DOP is switched ON,
2. For a SINGLE statement,
3. If the estimated serial execution time is higher than a system set THRESHOLD,
4. WITHOUT knowing anything about workload, concurrency, or other system factors,
5. Derive an Auto DOP based on optimizer cost ESTIMATES for SCAN operations only.
6. Then CAP that value to the ultimate value
7. And now run the statement with the resulting DOP.

```
PARALLEL_DEGREE_POLICY=
AUTO | LIMITED | ADAPTIVE
```

```
PARALLEL_MIN_TIME_THRESHOLD=
AUTO | value
```

```
PARALLEL_DEGREE_LIMIT=
CPU | IO | value
```

Note: However, Auto DOP does *not queue* anything.

ORACLE

After you set some or all Auto DOP parameters, Oracle will choose which statements are to be parallelized or not, and which DOP to use.

Examples

Example 1

- PARALLEL_DEGREE_POLICY is set to AUTO.

```
SQL> CREATE TABLE sales (c1 number, c2, date …) PARALLEL 64;
SQL> SELECT   sum(amount_sold)
  2  FROM     sales GROUP BY prod_id;
```

- Table SALES has a DEGREE of 64.
- The statement is executed with a DOP determined by Oracle and not the table decoration.

```
-----------------------------------------------------
| Id  | Operation                | Name     | Rows  |
-----------------------------------------------------
|   0 | SELECT STATEMENT         |          |    72 |
|   1 |  PX COORDINATOR          |          |       |
....
|   8 |         TABLE ACCESS FULL| SALES    |   918K|
-----------------------------------------------------
Note
-----
   - automatic DOP: Computed Degree of Parallelism is 2
```

ORACLE

If PARALLEL_DEGREE_POLICY=AUTO, a query on a table without any hint does not use the parallel DEGREE set for the table.

Examples

Example 2

- PARALLEL_DEGREE_POLICY is set to AUTO.
  - Issue: Too many statements are parallelized and consume available resources over others.
  - Solutions (choose one):
    - Increase PARALLEL_MIN_TIME_THRESHOLD.
    - Set PARALLEL_DEGREE_POLICY to LIMITED.

If PARALLEL_DEGREE_POLICY=AUTO and if there are too many statements running in parallel and consuming too many parallel resources, you have two solutions:

- Increase PARALLEL_MIN_TIME_THRESHOLD so that Oracle can run more statements in serial.
- Set PARALLEL_DEGREE_POLICY=LIMITED so that Oracle can parallelize only statements on objects having a DEGREE set.

Examples

Example 3

- PARALLEL_DEGREE_POLICY is set to LIMITED.

```
SQL> CREATE TABLE sales (c1 number, c2, date …) PARALLEL 8;
SQL> SELECT   sum(amount_sold)
  2  FROM     sales GROUP BY customer_id;
```

- Auto DOP is not involved.
- The statement is executed with a DOP determined by the table decoration, which is 8.

```
-------------------------------------------------------------
| Id  | Operation             | Name     | Rows  | Bytes |
-------------------------------------------------------------
|   0 | SELECT STATEMENT      |          |   918K|    25M|
|   1 |  PX COORDINATOR       |          |       |       |
|   2 |   PX SEND QC (RANDOM)| :TQ10000 |   918K|    25M|
|   3 |    PX BLOCK ITERATOR  |          |   918K|    25M|
|   4 |     TABLE ACCESS FULL| SALES    |   918K|    25M|
-------------------------------------------------------------
11 rows selected.
```

- No note of stating “automatic DOP: Computed Degree of Parallelism is 2”

ORACLE

If PARALLEL_DEGREE_POLICY=LIMITED, any statement can be parallelized, as long as the objects involved in the statement have a DEGREE set.

Examples

Example 4

- PARALLEL_DEGREE_POLICY is set to LIMITED.
- PARALLEL_DEGREE_LIMIT is set to CPU.

```
SQL> CREATE TABLE sales (c1 number, c2, date …) PARALLEL;
SQL> SELECT   sum(amount_sold)
  2  FROM     sales GROUP BY customer_id;
```

- The statement is executed with a calculated DOP capped with the PARALLEL_DEGREE_LIMIT.

```
-------------------------------------------------------------
| Id  | Operation            | Name     | Rows  | Bytes |
-------------------------------------------------------------
|   0 | SELECT STATEMENT     |          |   918K|    25M|
|   1 |  PX COORDINATOR      |          |       |       |
|   2 |   PX SEND QC (RANDOM)| :TQ10000 |   918K|    25M|
|   3 |    PX BLOCK ITERATOR |          |   918K|    25M|
|   4 |     TABLE ACCESS FULL| SALES    |   918K|    25M|
-------------------------------------------------------------
Note
-----
   - automatic DOP: Computed Degree of Parallelism is 2
```


If PARALLEL_DEGREE_POLICY=LIMITED, any statement can be parallelized, as long as the objects involved in the statement have a DEGREE set. In the slide example, the DEGREE set is DEFAULT. The DOP used is the one based on PARALLEL_DEGREE_LIMIT.

Other Instance Parameters

- When PARALLEL_FORCE_LOCAL is set to TRUE, it restricts the allocation of parallel server processes to the node to which the query coordinator is connected in a RAC environment.
  - Parallel statements are executed within a particular instance to avoid any interconnection with other instances.
  - If a user connects to a RAC service that encompasses more than one RAC node, PARALLEL_FORCE_LOCAL restricts the allocation of parallel server processes to whichever node the initial connection was mapped.

New Parameters	Allowable Values	Default
PARALLEL_FORCE_LOCAL	TRUE, FALSE	FALSE

- PARALLEL_IO_CAP_ENABLED is deprecated and remapped to PARALLEL_DEGREE_LIMIT set to IO.

PARALLEL_FORCE_LOCAL is used in RAC environments to restrict the allocation of PX servers from nodes other than the one that the query coordinator works on.

Parallel Hints Are Now at the Statement Level

- The scope of parallel hints is now at the *statement* level, superseding parallelism specified at table and object level.
- The following settings apply to the `PARALLEL_DEGREE_POLICY` initialization parameter:
  - `MANUAL` (default): Hints beginning with `PARALLEL` indicate the degree of parallelism for a specified *object*.
  - `LIMITED` or `AUTO`: The scope of the `PARALLEL` hint is the *statement*, not an object.

```
/*+ PARALLEL                  */
/*+ PARALLEL(AUTO)            */
/*+ PARALLEL(MANUAL)          */
/*+ PARALLEL(4)               */
/*+ NO_PARALLEL               */
```

ORACLE

Parallel execution hints instruct the optimizer about whether and how to parallelize operations.

In Oracle Database 12*c* Release 1, the scope of parallel hints is now at the *statement* level, superseding whatever parallelism might be specified at table and object level. It depends on the `PARALLEL_DEGREE_POLICY` value.

- `PARALLEL or PARALLEL (DEFAULT)`: The statement is always run parallel, and the database computes the degree of parallelism, which can be 2 or greater.
- `PARALLEL (AUTO)`: The database computes the degree of parallelism, which can be 1 or greater. If the computed degree of parallelism is 1, then the statement runs serially.
- `PARALLEL (MANUAL)`: The optimizer is forced to use the parallel settings of the objects in the statement.
- `PARALLEL (integer)`: The optimizer uses the degree of parallelism specified by an integer. Auto DOP is not involved.

EXPLAIN PLAN Enhancements

- EXPLAIN PLAN is modified to show the computed degree of parallelism that the optimizer used.
- Additional notes provide detail:
  - Computed degree of parallelism is <*DOP*>.
  - Computed degree of parallelism is <*DOP*> derived from scan of <object name>.
  - Computed degree of parallelism is <*DOP*> because of degree limit.
  - Computed degree of parallelism is 1 because of parallel threshold.
  - Computed degree of parallelism is 1 because of parallel overhead.
  - Degree of parallelism is <*DOP*> because of hint.

In Oracle Database 12*c* Release 1, the EXPLAIN PLAN output is enhanced to show the computed degree of parallelism that the optimizer used.

This computed degree of parallelism is stored by the optimizer as a statement-level annotation (similar to other statement-level annotation such as “dynamic sampling used” or “cost-based transformation used”) and exposed as a note of the EXPLAIN PLAN.

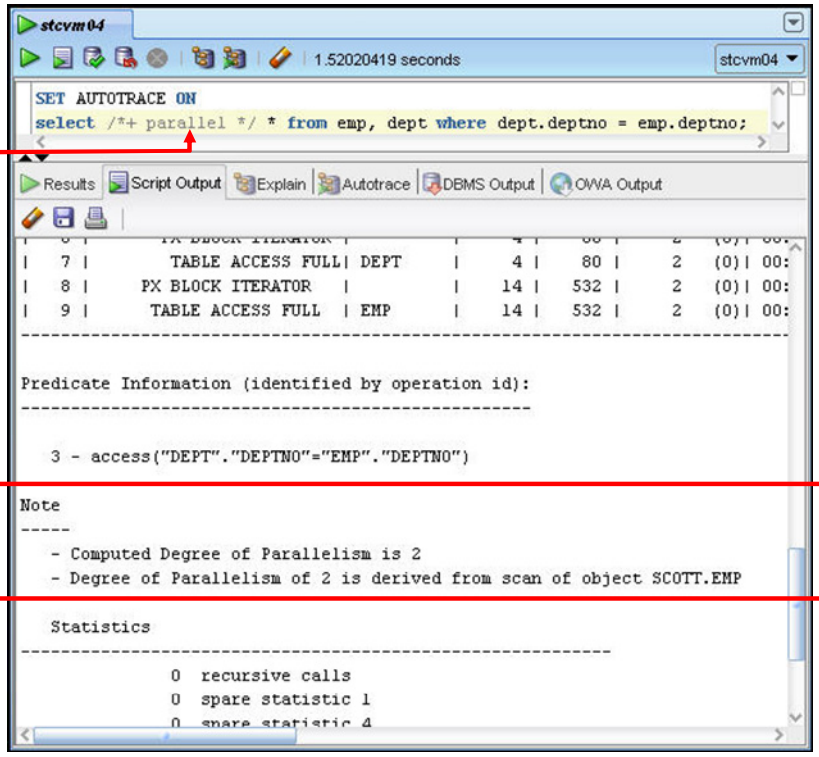
Note: Any statement about DOP computed value applies only at the moment of the EXPLAIN PLAN, due to real-time system conditions. Because of this, actual DOP is known only during run time, not parse time.

If you run the SQL a few seconds later, you could get a different DOP due to changes in system resources (number of PX processes, number of users connected, and so on).

The actual DOP may be different than the estimated DOP reported at EXPLAIN PLAN time.

Enhanced EXPLAIN PLAN Example

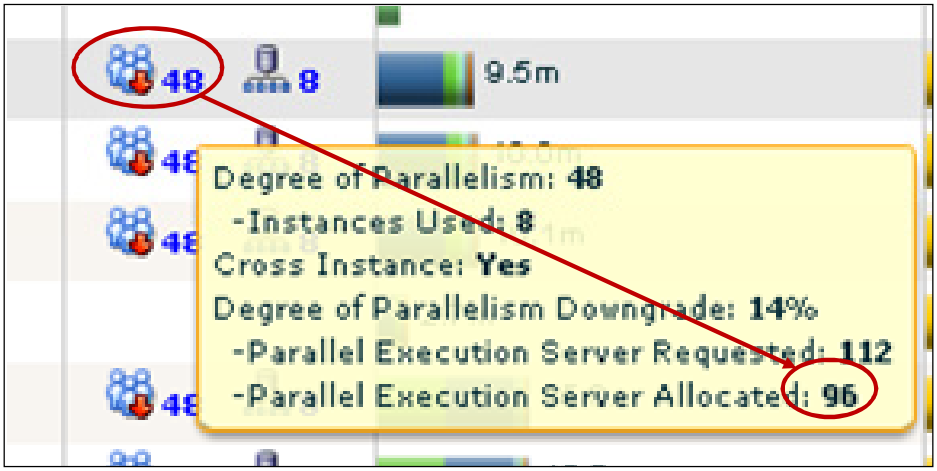
Parallel hint



In this example using SQL Developer, the note in the `EXPLAIN PLAN` output shows the computed degree of parallelism used by the optimizer. Autotrace is enabled, and the note appears in the Script Output tab after the script is executed in SQL Developer.

In this example, the computed degree of parallelism is 2. It has been derived from a scan of the `SCOTT.EMP` table.

A Workload with Auto DOP



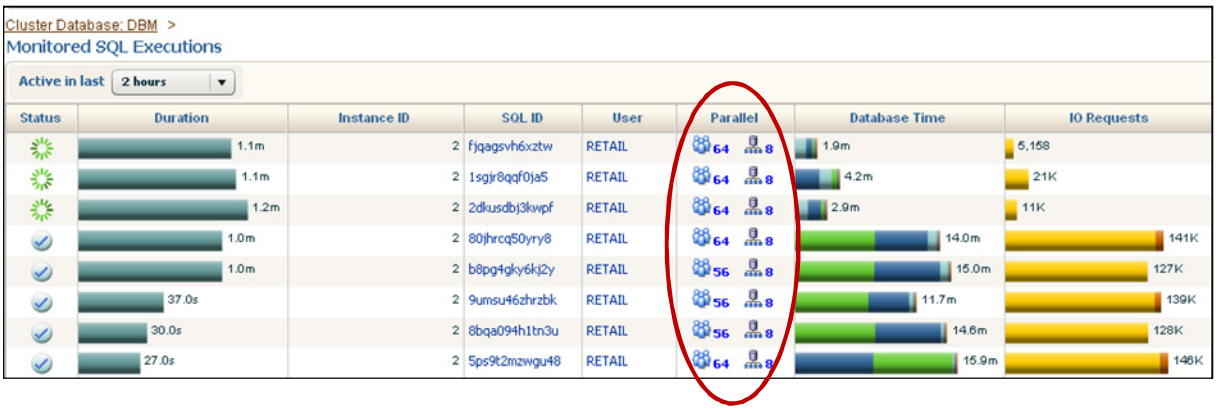
- `PARALLEL_DEGREE_POLICY` is set to `LIMITED` or `AUTO`.
- The DOP shown in EM is 48: number of PX servers per set.
- The Parallel Execution Server Allocated is 96: number of PX servers allocated for consumers and producers.
- The Parallel Execution Server Requested was 112 due to the desired DOP of 56, but could not be allocated.

DOP and Parallel Execution Servers

The number of parallel execution servers allocated is smaller than the number of parallel execution servers requested.

The possible reason for this is that the parallel execution servers requested are not available for the moment.

A Workload with Auto DOP



- Lower DOPs give less resource contention, leading to better run times and more predictability.
- No queuing in effect means a potential for overloading the system with parallel processes.

ORACLE

Examples with Manual and Automatic DOP Tuning

- Manual tuning for this demo set the DOP at 48.
- Auto DOP calculated 56 or 64 as the best DOP.
- Run times are very similar with either 48 or 56.

Impact of Auto DOP

- Potential changes in system behavior and query execution:
  - Number of potential statements running
  - Resources required
  - Individual response times
  - Concurrency of statements
- A shift in focus from individual DOP tuning exercises to *managing a workload*:
  - Worry less about individual DOPs.
  - Worry much more about *expected behavior.*

The slide bullets list the potential changes in system behavior and query execution after you have turned on Auto DOP.

Preventing Extreme DOPs

Set `PARALLEL_DEGREE_LIMIT`.

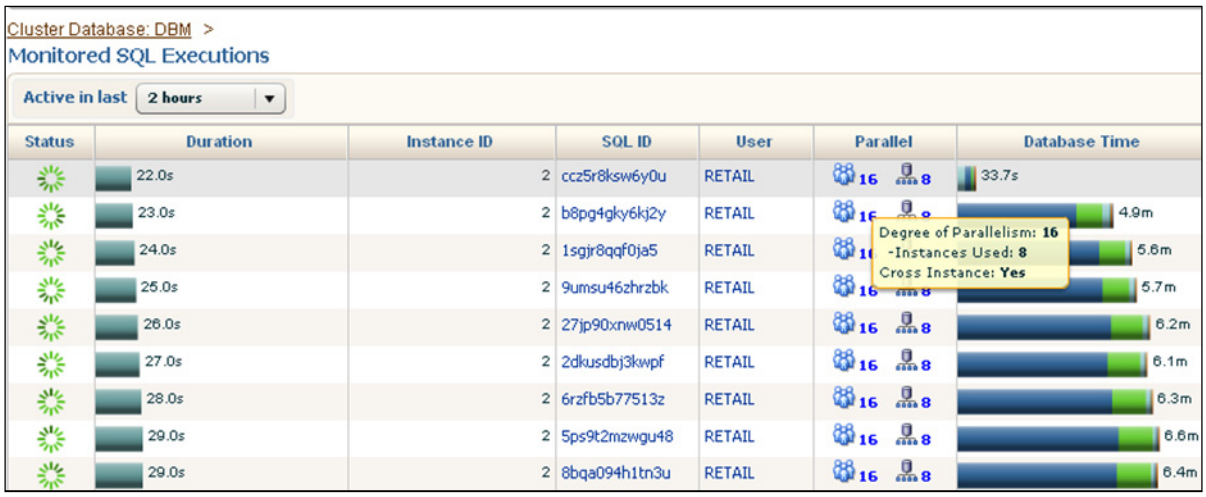
- The DBA caps the maximum degree *any* statement can run with on the system.
- If set to the default value (`CPU`), which equals default DOP:
  - No statement ever runs at a higher DOP than Default DOP
  - It is a good safety net for Auto DOP reaching high levels of DOP
- If set to a value (16), more statements can run with fewer PX servers than fewer statements with more PX servers.
- *Interesting:* When `PARALLEL_DEGREE_POLICY` is set to `LIMITED`, there is no statement queuing.
  - Pros: A simple way to increase concurrency and create predictable run times
  - Cons: Servers not fully utilized at low load, inflexible, and unable to prioritize queries

ORACLE

This is based on “rather have 6 statements running at 16 than 3 statements at 32.”

Note: EM will not show a downgrade for capped DOPs.

View the Degree Limit in Enterprise Manager



- PARALLEL_DEGREE_POLICY=AUTO
- PARALLEL_DEGREE_LIMIT=16
- Rather than 56 or 64, DOP is 16.

Note: There is no indication of downgrades on the statements.

ORACLE

If PARALLEL_DEGREE_POLICY=AUTO and PARALLEL_DEGREE_LIMIT=16, any statement can be parallelized with a maximum DOP of 16. Even if the statement could have benefited available PX servers and used a DOP greater than this limit, the DOP used is the one based on PARALLEL_DEGREE_LIMIT. Oracle, in this case, does not show any downgrade in DOP on the statement.

Summary

In this lesson, you should have learned to:

- Describe the benefits of using Auto DOP
- Explain the parameters of Auto DOP
- Explain when to use the Auto DOP
- Explain how to use the Auto DOP

6

Statement Queuing

Objectives

After completing this lesson, you should be able to:

- Understand why and when to use statement queuing
- Understand concurrency between parallelized statements
- Describe the parameters to set
- Use Database Resource Manager to manage parallel statement queuing

No Statement Queuing Before Auto DOP

- Before Auto DOP, guarantee of a minimal DOP for parallel statements could result in statements not running
- No statement queuing
- `PARALLEL_MIN_PERCENT`: Parallel operations do not execute unless adequate resources are available.
  - The statement executes with at least the minimum PX servers.
  - The statement does not execute: The requested DOP cannot be satisfied by the system at a given time.

```
ORA-12827 : insufficient parallel query
slaves available
```


`PARALLEL_MIN_PERCENT` Parameter

This parameter enables users to claim an acceptable DOP, depending on the application in use.

Setting this parameter to values other than 0 (zero) causes Oracle Database to return an error when the requested DOP cannot be satisfied by the system at a given time. For example, if you set `PARALLEL_MIN_PERCENT` to 50, which translates to 50 percent, and the DOP is reduced by 50 percent or greater because of the adaptive algorithm or because of a resource limitation, then Oracle Database returns `ORA-12827`.

The default value of 0 means that no minimum percentage of processes has been set.

To control parallel execution, the best practice is to use Resource Manager.

Why Use Statement Queuing?

With the introduction of the Auto DOP capability:

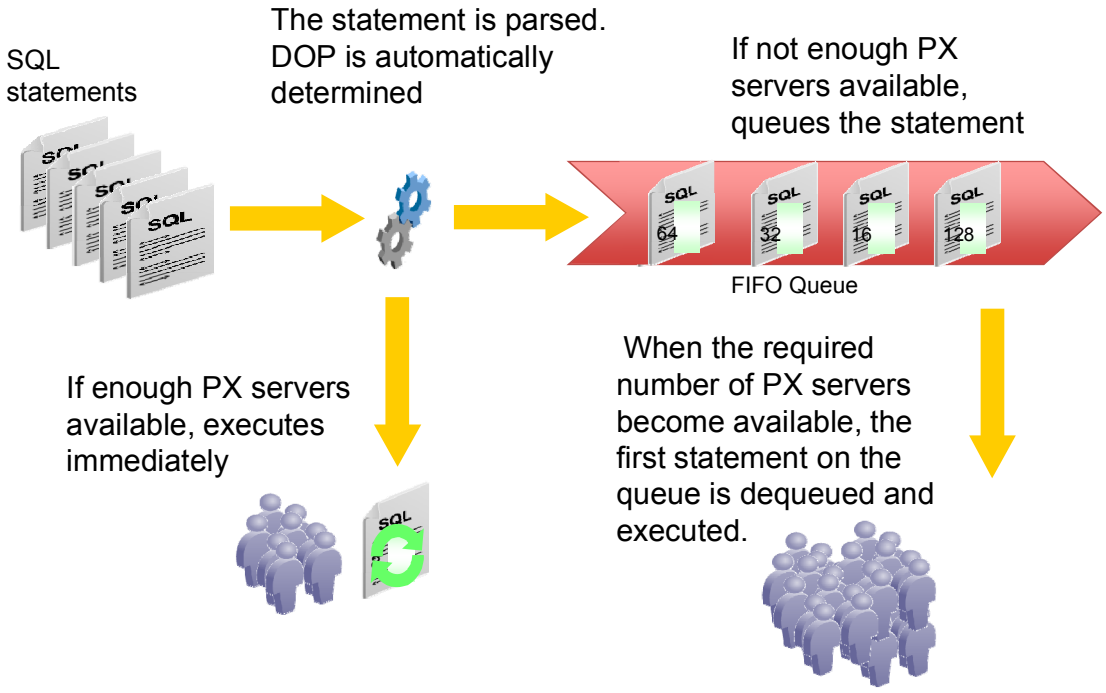
- More statements run in parallel.
  - Possible to exhaust all parallel PX processes
  - Potential system overwhelming due to too many processes
- Parallel statement queuing
  - Oracle automatically decides whether a statement can execute immediately.
  - It prevents serializing parallel queries when parallel servers are not available.
  - It prevents the system from being overwhelmed.

ORACLE

With the introduction of the Auto DOP capability, Oracle automatically decides whether a statement should execute:

- In parallel or not
- With which computed DOP
- Immediately or queued until more system resources are available
- Taking advantage of the aggregated cluster memory or not (In-Memory PQ)

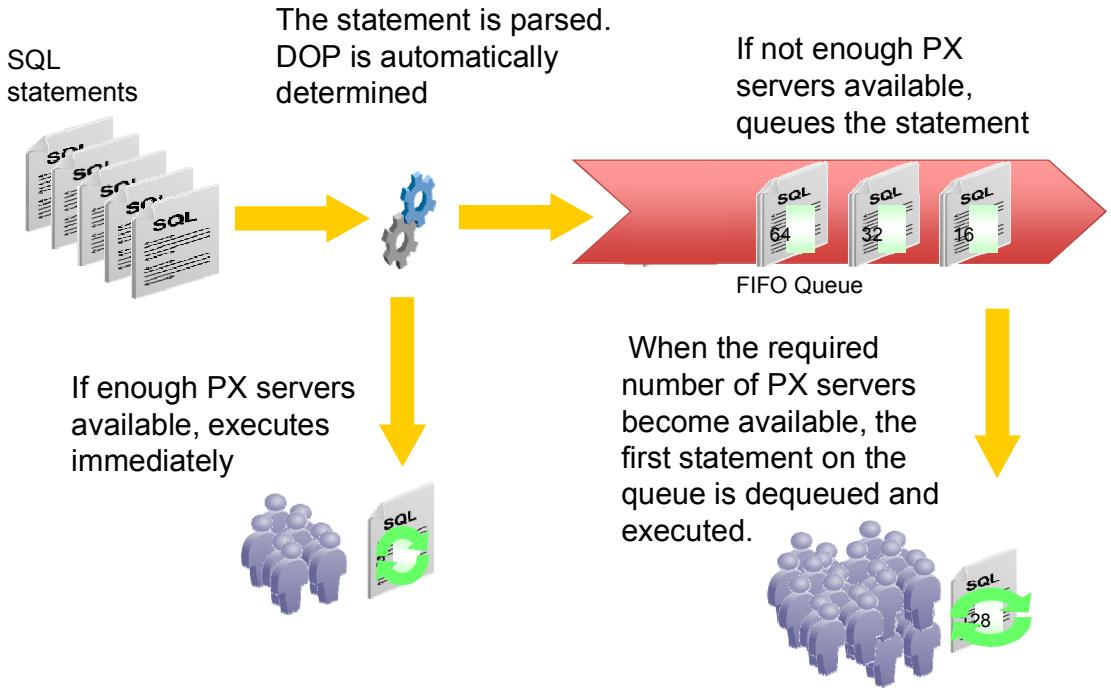
How Statement Queuing Works



1. SQL Statement is issued. Oracle automatically determines whether it runs in parallel and, if so, which DOP it gets.
2. Oracle checks whether there are enough PX servers to execute the query.
   - If there are, it executes immediately.
   - If there are not, then the query is queued in a FIFO method.
3. Everyone waits in the queue for the statement at the top of the queue to get all of his requested PX servers even if there are enough PX servers for other statements in the queue to start running.
4. When enough PX servers are available, the statement is dequeued and allowed to execute.

In the slide example, the first statement that will be dequeued requires 128 PX servers.

How Statement Queuing Works



1. SQL Statement is issued. Oracle automatically determines whether it runs in parallel and, if so, which DOP it gets.
2. Oracle checks whether there are enough PX servers to execute the query.
   - If there are, it executes immediately.
   - If there are not, then the query is queued in a FIFO method.
3. Everyone waits in the queue for the statement at the top of the queue to get all of his requested PX servers even if there are enough PX servers for other statements in the queue to start running.
4. When enough PX servers are available, the statement is dequeued and allowed to execute.

In the slide example, the first statement in the queue was dequeued.

The next in the FIFO queue waits for the 16 PX servers requested.

Statement Queuing Setting

Statement queuing ensures that each statement is allocated the necessary PX server resources.

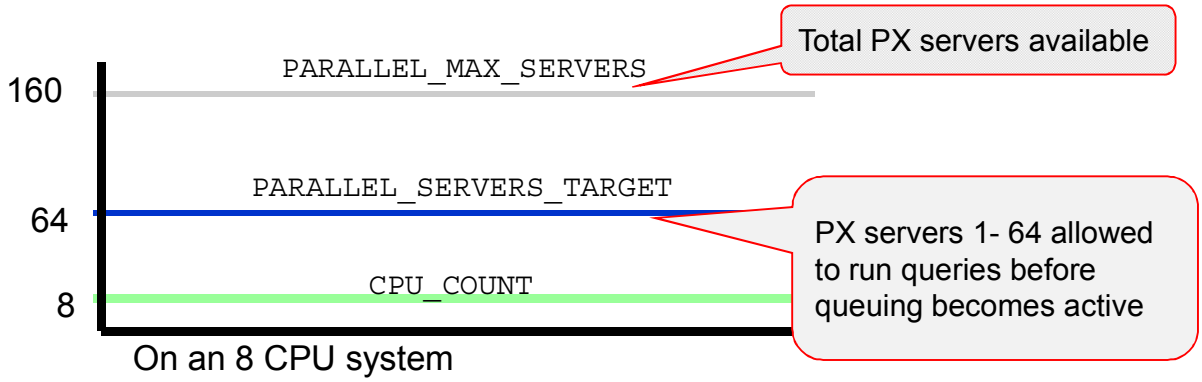
- Enabled only when `PARALLEL_DEGREE_POLICY` is set to AUTO
- Enforced with a queue FIFO policy
- Defined by `PARALLEL_SERVERS_TARGET` indicating how many PX servers are allowed to run queries in parallel before queuing becomes active
  - Default value:
    `4 * PARALLEL_THREADS_PER_CPU * CPU_COUNT * ACTIVE_INSTANCE_COUNT`
  - Is a soft limit, whereas `PARALLEL_MAX_SERVERS` is the hard limit

ORACLE

When `PARALLEL_DEGREE_POLICY` is set to `AUTO`, statements that require parallel execution are queued if the number of parallel processes currently in use on the system equals or is greater than `PARALLEL_SERVERS_TARGET`.

This is not the maximum number of parallel server processes allowed on a system (that is controlled by `PARALLEL_MAX_SERVERS`). However, `PARALLEL_SERVERS_TARGET` and parallel statement queuing are used to ensure that each statement that requires parallel execution is allocated the necessary parallel server resources and the system is not flooded with too many parallel server processes.

Statement Queuing Setting



Example:

- PARALLEL_DEGREE_POLICY = AUTO
- The DBA wants to ensure fewer parallel statements run before queuing starts on the system. What should he do?
- Possible solutions:
  - Decrease the value of PARALLEL_SERVERS_TARGET
  - Increase the value of PARALLEL_DEGREE_LIMIT

ORACLE

Example

Decreasing PARALLEL_SERVERS_TARGET leads to fewer statements starting to execute in parallel and fewer resources consumed.

Increasing PARALLEL_DEGREE_LIMIT leads to fewer statements running a bit faster.

Statement Queuing Setting

- You can switch all these parameters per SESSION:

```
SQL> ALTER SESSION SET parallel_degree_policy = AUTO;
SQL> ALTER SESSION SET parallel_degree_limit = 32;
SQL> ALTER SESSION SET parallel_servers_target = 64;
```

- You can hint to queue or not to queue.

```
SQL> SELECT /*+ NO_STATEMENT_QUEUING */ * FROM t1;
SQL> SELECT /*+ STATEMENT_QUEUING */ * FROM t1;
```


How Parallel Statement Queuing Works

You can set the PARALLEL_DEGREE_POLICY, PARALLEL_DEGREE_LIMIT, and PARALLEL_SERVERS_TARGET to either of the following levels:

- Instance level
- Session level

NO_STATEMENT_QUEUING Hint

This hint enables a statement to bypass the parallel statement queue even though PARALLEL_DEGREE_POLICY is set to AUTO. For example:

```
SQL> SELECT /* NO_STATEMENT_QUEUING */
  2         emp.last_name, dpt.department_name
  3  FROM   employees emp, departments dpt
  4  WHERE  emp.department_id = dpt.department_id;
```

STATEMENT_QUEUING Hint

When PARALLEL_DEGREE_POLICY is not set to AUTO, this hint enables a statement to be delayed and to only run when parallel processes are available to run at the requested degree of parallelism. This ensures that the statement runs in parallel.

Statement Queuing Monitoring

- V$SQL_PLAN_MONITOR has a new status value for queued statements:

```
SELECT s.sql_id, s.sql_text
FROM   V$SQL_MONITOR m, V$SQL s
WHERE  m.status = 'QUEUED'
AND    m.sql_id = s.sql_id;
```

- EM SQL Monitoring also displays queuing statements.
- Two new wait events
  - PX Queuing: Statement queue
    - Indicates the first query in the queue
  - enq: JX-SQL statement queue
    - Indicates the statement is queued (but not the first in line to go)

ORACLE

V$SQL_PLAN_MONITOR View

The column status has a new possible value: QUEUED. This means that the statement waits to acquire the requested PX servers.

When to Use Statement Queuing

- Parallel statement queuing is beneficial when it:
  - Allows for higher DOPs per statement without overwhelming the system
  - Allows a set of queries to run at roughly the same aggregate time by allowing the optimal DOP to be used all the time
- Parallel statement queuing is harmful when it:
  - Adds delay to your execution time if your statement is queued, making elapse times more unpredictable
  - Struggles with complex mixed workloads
- Solution: Find the optimal queuing point based on the desired concurrency.

ORACLE

First of all, you should decide whether you want queuing and how aggressive you want to apply it in your system or in a more subtle method within your group of consumers.
That is because queuing comes with both upsides and downsides.

- The upside? Each statement runs with a computed or set DOP. Downgrade is impossible.

  That means that a statement:
  - Runs faster than if you run it with a lower DOP (provided the DOP calculation is reasonably well done)
  - Does not overwhelm the system by using too many resources
- The downside is an element of unpredictability: If the statement runs in 20 seconds today, it may run in 30 seconds tomorrow due to 10 seconds in the queue.
  In mixed workloads, where short-running and long-running statements execute concurrently, long-running statements may prevent queued short-running ones from starting rapidly and may delay them for a long time.
  In 11.2.0.1, the single queue is suitable for less complex mixed workloads.
  In 11.2.0.2, you can use Database Resource Manager for more fine-grained control over statement queuing. You can set different queues along with consumer groups: a consumer group executing long-running statements having its own queue and another consumer group executing short-running statements having its own queue.

Statement Queuing

- Consider :
  - `PARALLEL_DEGREE_POLICY = AUTO`
  - `PARALLEL_SERVERS_TARGET = 32`
  - `PARALLEL_DEGREE_LIMIT = CPU`
  - `PARALLEL_MAX_SERVERS = 102`
- Does the first statement run with DOP 48, or does it queue?

What happens when a statement requires a DOP and PX servers that exceed the value of `PARALLEL_SERVERS_TARGET`? Does it queue endlessly?

Statement Queuing

- Consider :
  - `PARALLEL_DEGREE_POLICY = AUTO`
  - `PARALLEL_SERVERS_TARGET = 32`
  - `PARALLEL_DEGREE_LIMIT = CPU`
  - `PARALLEL_MAX_SERVERS = 102`
- Does the first statement run with DOP 48, or does it queue?
  - It does *not* queue, because the first statement is allowed to break the barrier.
  - It cannot exceed the `PARALLEL_MAX_SERVERS` or it will be downgraded.
  - Each subsequent parallel statement will queue until this one finishes.

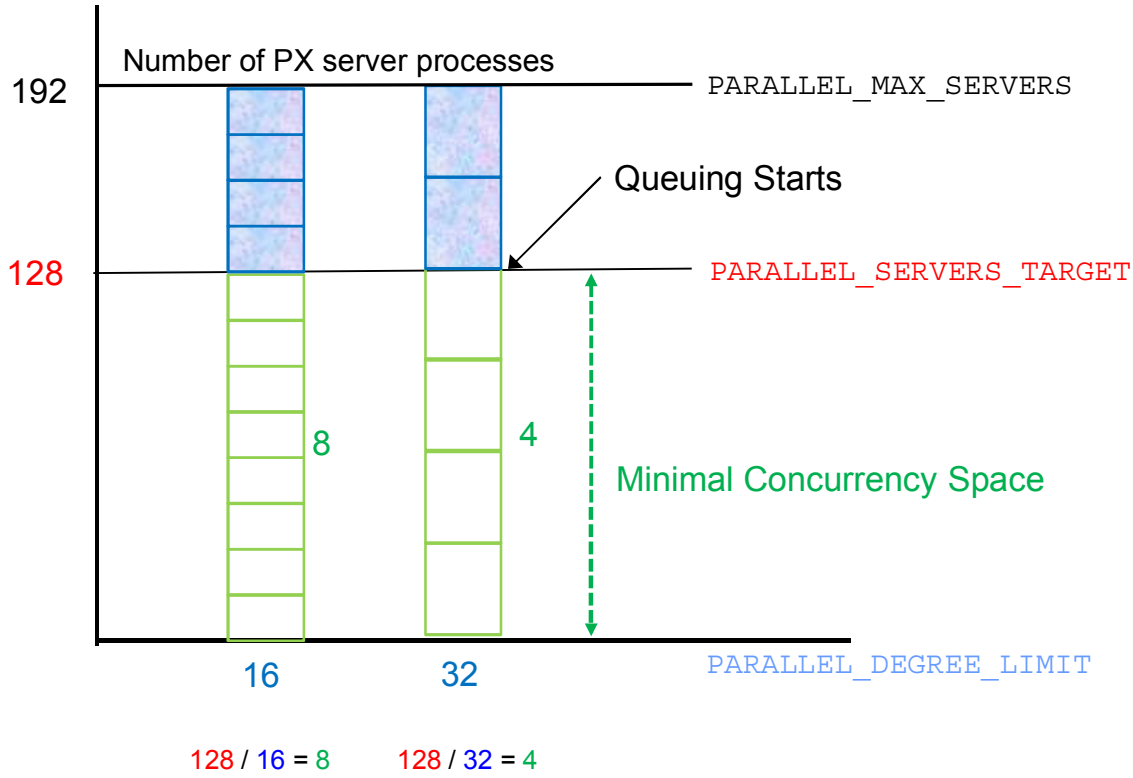
ORACLE

Statement Queuing Parameters

What happens when a statement requires a DOP and PX servers that exceed the value of `PARALLEL_SERVERS_TARGET`? Does it queue endlessly?

It starts running in parallel as long as it is the first statement in the queue.

Statement Queuing and Concurrency



The minimal concurrency is the minimal number of parallel statements that can run before queuing becomes active.

In the first slide example, consider the following:

- `PARALLEL_DEGREE_LIMIT` is set to 16
- `PARALLEL_SERVERS_TARGET` is set to 128

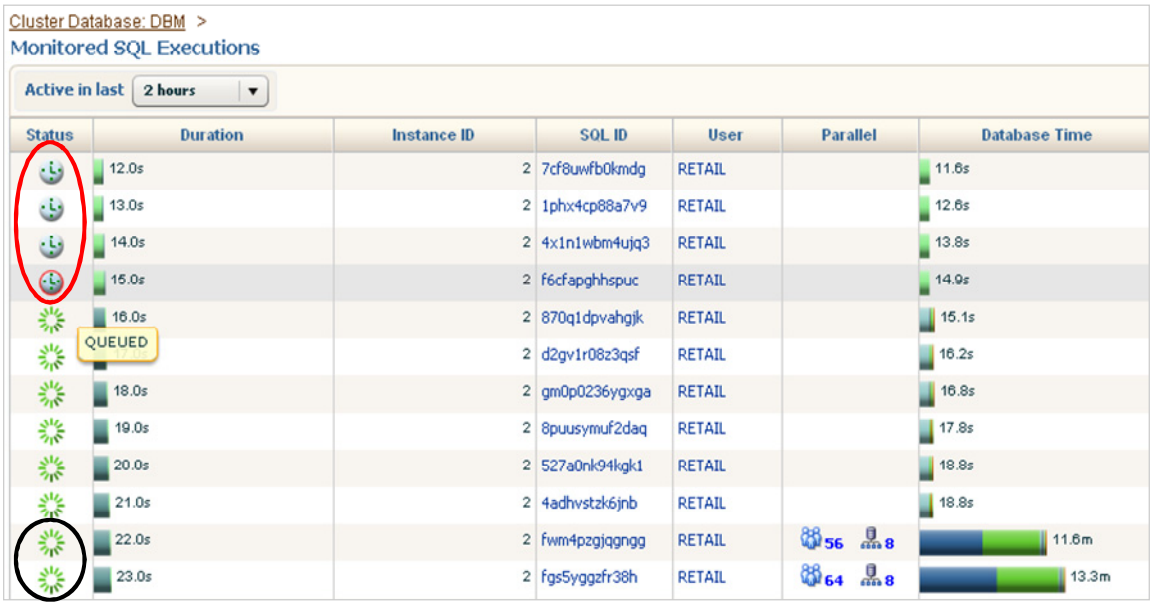
If several concurrent statements require a DOP of 16, the first 8 statements will execute with a DOP of 16, provided they need only 16 PX server processes each. The following statements requiring a parallel execution will not get any available PX servers and will start queuing.

In the second slide example, consider the following:

- `PARALLEL_DEGREE_LIMIT` is set to 32
- `PARALLEL_SERVERS_TARGET` is set to 128

If several concurrent statements require a DOP of 32, the first 4 statements will execute with a DOP of 32, provided they need only 32 PX server processes each. The following statements requiring a parallel execution will not get any available PX servers and will start queuing.

Statement Queuing and Concurrency



Two statements run before queuing becomes active.

In the slide example, only 2 statements run concurrently in parallel. One is using a DOP of 56 and the other one a DOP of 64. Therefore, both statements are consuming 120 PX servers at least or 240 if they require producers and consumers. Because the `PARALLEL_SERVERS_TARGET` is set to 128, all other subsequent parallel statements start queuing.

Statement Queuing and Concurrency

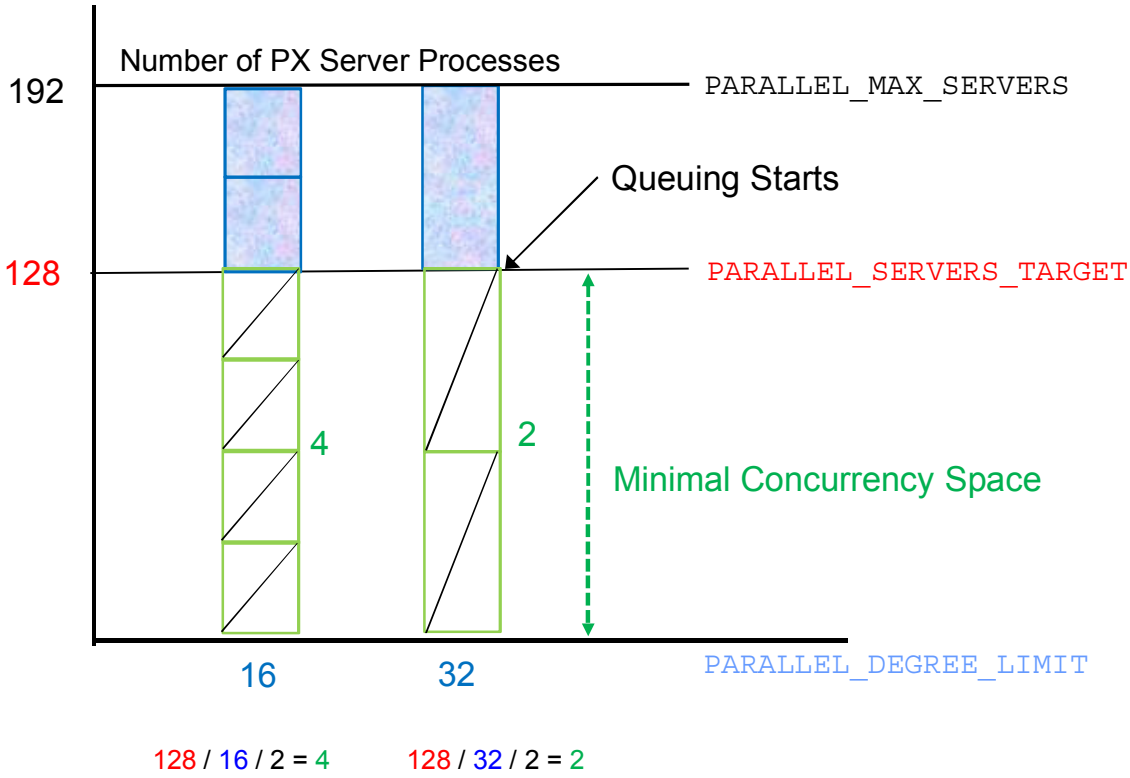
Calculating minimal concurrency:

- Minimal concurrency is the minimal number of parallel statements that can run before queuing becomes active.

$$\text{minimal concurrency} = \frac{\texttt{PARALLEL\_SERVERS\_TARGET}}{\texttt{PARALLEL\_DEGREE\_LIMIT}}$$

The slide shows the formula to find the minimal concurrency in case the statements require only one PX set each.

Statement Queuing and Concurrency: 2



In the first slide example, consider the following:

- `PARALLEL_DEGREE_LIMIT` is still set to 16.
- `PARALLEL_SERVERS_TARGET` is still set to 128.

If several concurrent statements require a DOP of 16 with PX sets (producers and consumers), only four statements will execute with a DOP of 16, as they will request 32 PX server processes each. The following statements requiring a parallel execution will not get any available PX servers and will start queuing.

In the second slide example, consider the following:

- `PARALLEL_DEGREE_LIMIT` is set to 32.
- `PARALLEL_SERVERS_TARGET` is set to 128.

If several concurrent statements require a DOP of 32 with PX sets (producers and consumers), only two statements will execute with a DOP of 32, as they will request 64 PX server processes each. The following statements requiring a parallel execution will not get any available PX servers and will start queuing.

Statement Queuing and Concurrency: 2

- Consider another situation:
  - Number of active instances: 8
  - `PARALLEL_DEGREE_POLICY = AUTO`
  - `PARALLEL_SERVERS_TARGET = 128` per node `(1024)`
  - `PARALLEL_DEGREE_LIMIT = CPU`
  - The sum of all DOPs is 744.
- Why does queuing start after 8 statements?

ORACLE

In the slide example, consider the following:

- `PARALLEL_DEGREE_LIMIT` is set to CPU. Assume that the default DOP is computed to 64.
- `PARALLEL_SERVERS_TARGET` is set to 128 per node. As we have 8 nodes, this makes (128 * 8=) 1024 PX servers usable.

If several concurrent statements require a DOP of 64 with PX sets (producers and consumers), only 8 statements will execute with a DOP of 64, as they will request 128 PX server processes each: 8 (statements) * 64 (DOP) * 2 (sets) = 1024 PX servers. Any following statement requiring a parallel execution will not get any available PX servers and will start queuing.

Statement Queuing and Concurrency: 2

- Consider another situation:
  - Number of active instances: 8
  - `PARALLEL_DEGREE_POLICY = AUTO`
  - `PARALLEL_SERVERS_TARGET = 128` per node `(1024)`
  - `PARALLEL_DEGREE_LIMIT = CPU`
  - The sum of all DOPs is 744.
- Why does queuing start after 8 statements?
  - `PARALLEL_SERVERS_TARGET` is a number of PX server processes.
  - DOP is a degree of parallelism that may require for each statement twice the DOP PX servers, due to producers and consumers.

ORACLE

In the slide example, `PARALLEL_SERVERS_TARGET` means 1024 PX servers are usable. But the sum of the DOPs of 744 means that a minimum of 744 PX servers are necessary, provided that each statement requests only one PX set.

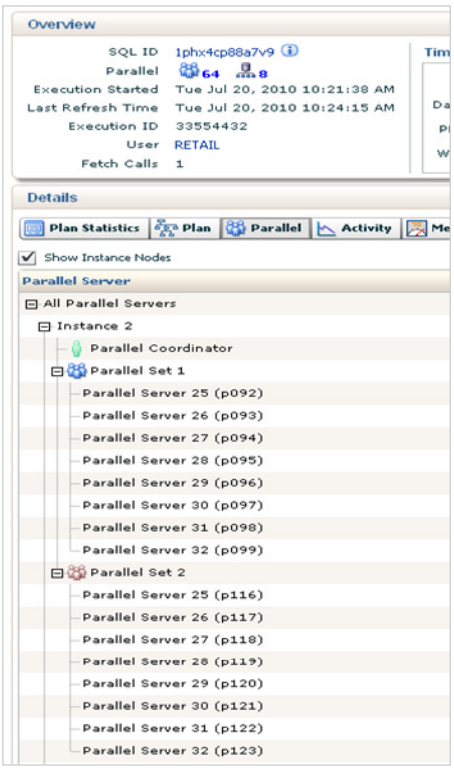
DOP Versus Processes

EM shows a DOP of 64 and the number of instances of 8.

- 8 PX producers and
  8 PX consumers processes
  per node:
  16 * 8 (instances) = 128 processes

or

- 2 * DOP = 2 * 64 = 128 processes



ORACLE

The DOP of 64 may mean that 64 PX servers requested or 128 PX servers requested.

Statement Queuing and Concurrency: 2

Consider that statements always or most of the time need producers and consumers (not PWJ all the time, which requires only one PX set):

- Minimal concurrency is the minimal number of parallel statements that can run before queuing becomes active:

$$\text{minimal concurrency} = \frac{\texttt{PARALLEL\_SERVERS\_TARGET}}{\texttt{PARALLEL\_DEGREE\_LIMIT}} / 2$$

- This is a more conservative method.

ORACLE

The slide shows the formula to find the minimal concurrency in case the statements require two PX sets each.

Workload-Oriented Statement Queuing

Individual SQL statement queuing analysis
Versus
Workload-oriented statement queuing analysis

- To address the last:
  - Leverage services to dispatch PX resources to users
- Use Database Resource Manager to :
  - Cap the DOP per consumer group
  - Set a percentage of PX servers that each consumer group can use before statement queuing becomes active
  - Set a queue timeout per consumer group
- Map DBRM consumer groups to services.
- Reminder: Auto DOP uses an estimated value if I/O Calibrate has not been performed.

ORACLE

Individual SQL Statement Queuing Analysis

When using instance parameter to set the DOP limit, all SQL statements and users are under this unique policy. You can alter the behavior for specific SQL statements with session parameters or hints.

However, a SQL statement may not need to follow the same policy under different workload situations.

Parallel Execution and RAC Services

In a RAC environment, services are used to limit the number of instances that participate in a parallel SQL operation.

Parallel Execution and Database Resource Manager (DBRM)

The parameter `PARALLEL_DEGREE_LIMIT` is used as a system wide safety net.
Cap the maximum DOP as a directive in DBRM for each consumer group.

Calculate the minimal concurrency as a threshold in DBRM for each consumer group.

Database Resource Manager Consumer Group Mappings for Services

Services are integrated with the Resource Manager, which enables you to restrict the resources that are used by the users who connect to an instance by using a service. The Resource Manager enables you to map a consumer group to a service so that users who connect to an instance by using that service are members of the specified consumer group.

Hence, the maximum DOP directive and the minimal concurrency threshold set for each consumer group in DBRM are used for those users who connect through a service mapped to the consumer group to which the user belongs.

Database Resource Manager Queuing

Without DBRM, the single FIFO queue is used systemwide.

- Everyone queues in the same queue.
- There is no priority as to where you are in the queue.
- A statement gets stuck behind the query that requires a large number of resources.
- This easy setup is appropriate for less complex mixed workloads.

With DBRM:

- One queue per consumer group
  - `PARALLEL_TARGET_PERCENTAGE`
  - `PARALLEL_QUEUE_TIMEOUT`
- Priorities between consumer groups' queues: `MGMT_Pn`
- Appropriate for more complex mixed workloads

One Queue Per Consumer Group

In DBRM, plan directives can be set per consumer group to set limits of different resources that can be consumed.

A New Plan Directive

The plan directive, `PARALLEL_TARGET_PERCENTAGE`, sets the maximum percentage of `PARALLEL_SERVERS_TARGET` that a consumer group can use before the statements of the users of the group are queued.

The number of PX servers used by a particular consumer group is counted as the sum of the PX servers used by all sessions in that consumer group.

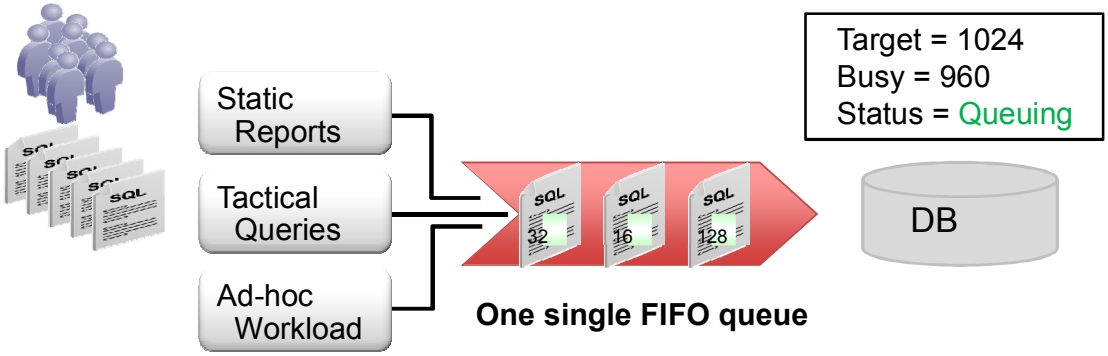
Example: Assume that the total number of available PX servers is 32, as set by the `PARALLEL_SERVERS_TARGET` initialization parameter, and the `PARALLEL_TARGET_PERCENTAGE` directive attribute for the consumer group `MY_GROUP` is set to 50.

This consumer group can use a maximum of 50% of 32, or 16 PX servers.

Remarks

- If the resource plan does not have any management attributes (mgmt_p1, mgmt_p2, and so on), then the parallel statement queue is managed as a first in, first out (FIFO) queue and PARALLEL_TARGET_PERCENTAGE is enforced on top of this. Management attributes control how a parallel statement is selected from the parallel statement queue for execution. You can prioritize the parallel statements of one consumer group over another by setting a higher value for the management attributes of that group.
- The sum of all PARALLEL_TARGET_PERCENTAGE does not have to add up to 100 over the groups in the plan.
- If there is no resource plan active, no management directives are set. Hence, statement queuing works with the single FIFO queue.
- Although parallel server usage is monitored for all sessions, the parallel server directive attributes you set affect only sessions for which parallel statement queuing is enabled (PARALLEL_DEGREE_POLICY is set to AUTO). If a session has the PARALLEL_DEGREE_POLICY set to MANUAL, parallel statements from this session are not queued. However, any parallel servers used by such sessions are included in the count that is used to determine the limit for PARALLEL_TARGET_PERCENTAGE. Even if this limit is exceeded, parallel statements from this session are not queued.

Queuing with a Single Queue

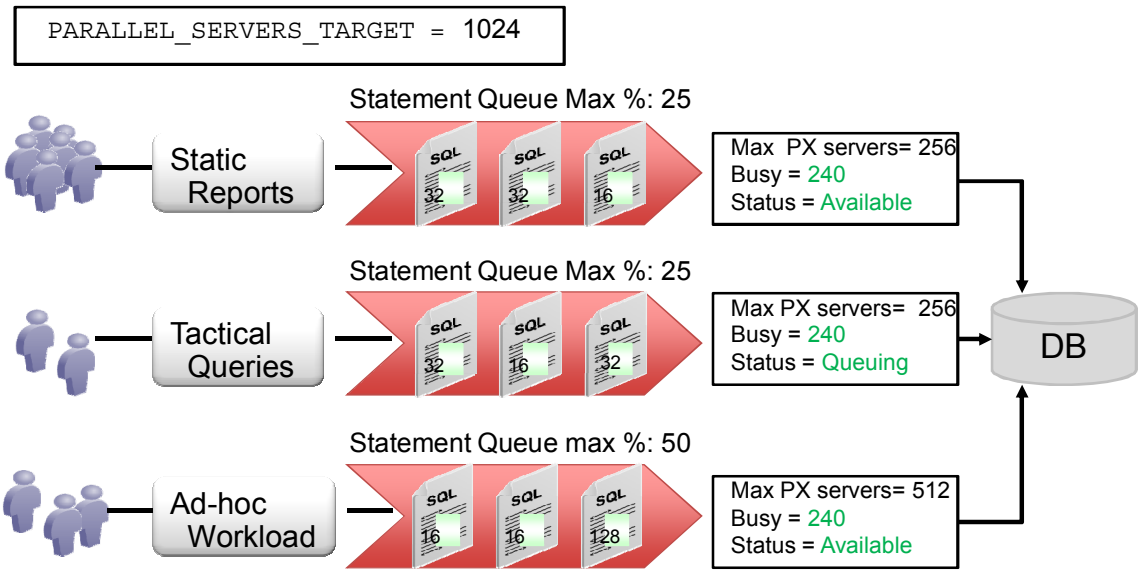


Advantages of the single FIFO queue used systemwide:

- Easy setup
- No DBRM directive to set
- Appropriate for less complex mixed workloads

In the slide example, if the `PARALLEL_SERVERS_TARGET` is set to 1024, and 960 PX servers are already used, the next parallel statement is queuing in the single FIFO queue.

Queuing with DBRM



One FIFO queue per consumer group with DBRM:

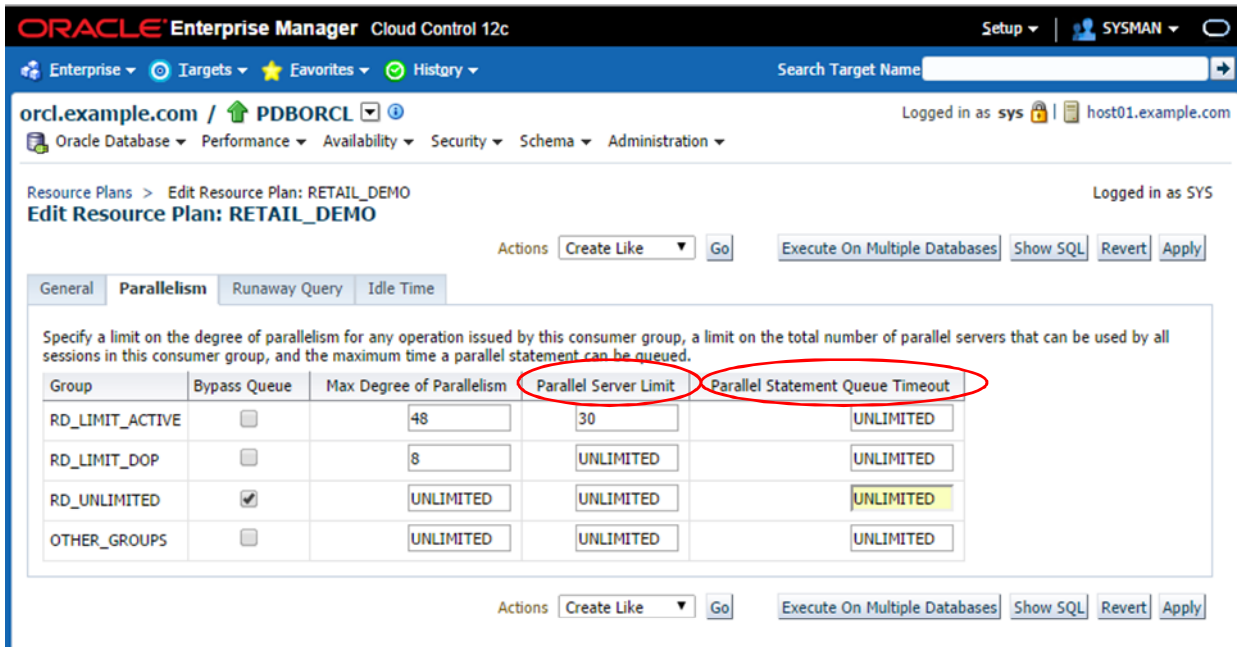
- DBRM directives to set
- Appropriate for more complex mixed workloads

ORACLE

DBRM Queuing

In the slide example, if the `PARALLEL_SERVERS_TARGET` is set to 1024, and different `PARALLEL_TARGET_PERCENTAGE` directive values are set for the three consumer groups, each consumer group has its own FIFO queue.

DBRM and EM

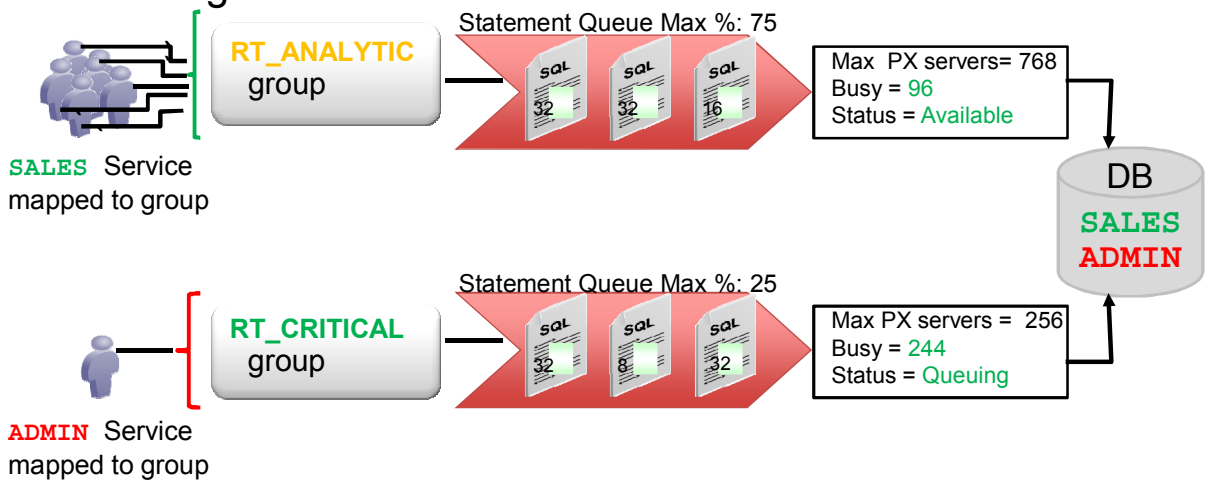


DBRM Queuing

In EM, the `PARALLEL_TARGET_PERCENTAGE` directive in DBRM is displayed with a heading of Parallel Server Limit, and the `PARALLEL_QUEUE_TIMEOUT` directive is displayed with a heading of Parallel Statement Queue Timeout.

Parallel Statement Queuing in RAC

- A great many systems today leverage services to divide a set of resources.
- Services work nicely with both DBRM and with statement queuing, giving you another level of working with resource management.



DBRM Queuing and Services

In the case of an Oracle Real Application Clusters (Oracle RAC) environment, the maximum possible PX servers that a consumer group mapped with a service can use is the sum of (`PARALLEL_SERVER_LIMIT` * `PARALLEL_SERVERS_TARGET` / 100) across Oracle RAC instances within that service.

Services are integrated with Resource Manager, which enables you to restrict the resources that are used by the users who connect to an instance by using a service. Resource Manager enables you to map a consumer group to a service so that users who connect to an instance using that service are members of the specified consumer group.

Hence, the maximum DOP directive and the minimal concurrency threshold set for each consumer group in DBRM are used for those users who connect through a service mapped to the consumer group to which the user belongs.

This allows you to separate the workloads in a physical environment (for example, log on to service `SALES` or `ADMIN`).

Statement queuing is still set within the group. But you must keep in mind that the service has only a limited number of nodes at its disposal and therefore a reduced number of processes in `PARALLEL_SERVERS_TARGET` for the service.

- Each node has its number for PARALLEL_SERVERS_TARGET.
- The aggregate for the whole system is node count * PARALLEL_SERVERS_TARGET.
- For the service, it is node count within service * PARALLEL_SERVERS_TARGET.

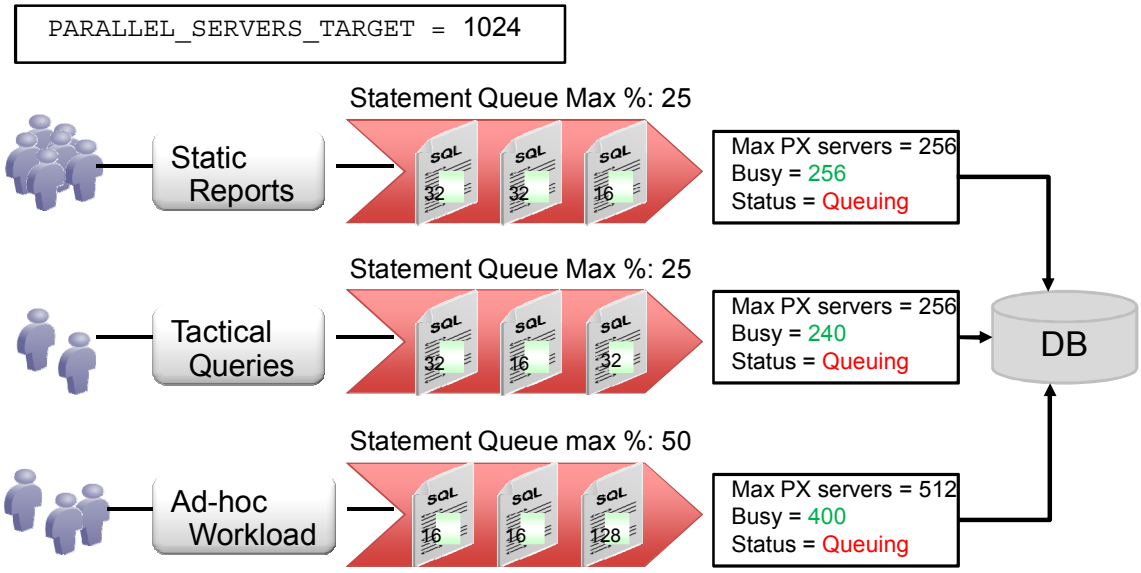
Consider two services created that divide the 8-node RAC system into two services called SALES and ADMIN. You can now layer the groups on top of the services. This allows you to separate the workloads in a physical environment (for example, log on to service SALES and ADMIN) and then manage various consumer groups with these services.

Server Pools

By using server pools, you can choose to change the service “size” and add or revoke a node from a service at specific times. For example, to run a large ETL load on service SALES, you may want to add a node to it.

Cluster reconfiguration: Queue is aware of nodes leaving and joining the cluster and adjusts the limits accordingly.

DBRM Queuing Priority



- All queues are in “Queuing” status.
- Which queue is dequeued first?

Queuing Priorities with `MGMT_Pn` Directive

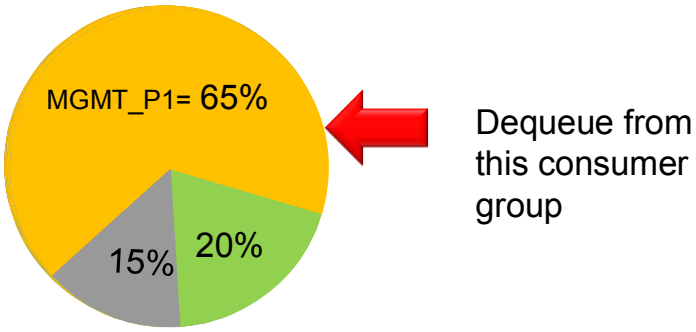
Management attributes control how a parallel statement is selected from the parallel statement queue for execution. You can prioritize the parallel statements of one consumer group over another by setting a higher value for the management attributes of that group.

If your resource plan has management attributes (`MGMT_P1`, `MGMT_P2`, and so on), then a separate parallel statement queue is managed as a First In First Out (FIFO) queue for each management attribute.

If your resource plan does not have any management attributes, then a single parallel statement queue is managed as a FIFO queue.

DBRM Queuing Priorities

- Plan directive `MGMT_Pn` is set for each consumer group.
- The group with the highest `MGMT_Pn` is dequeued first.



Queuing Priorities with `MGMT_Pn` Directive

In the previous slide example, if the `Ad-hoc Workload` consumer group is the one having the highest `mgmt_p1` value, the queuing statement of the `Ad-hoc Workload` consumer group would be the one to be dequeued first.

Limiting Consumer Groups

A single resource consumer group could use all available parallel execution servers. This can be controlled with:

- `PARALLEL_SERVER_LIMIT`: maximum percentage of the parallel execution server pool that a consumer group can use
- `PARALLEL_STMT_CRITICAL`: parallel statements from consumer group bypass the parallel statement queue

It is possible for a single consumer group to launch enough parallel statements to use all of the available parallel execution servers. If this happens when a high-priority parallel statement from a different consumer group is run, then no parallel execution servers are available to allocate to this group. You can avoid such a scenario by limiting the number of parallel execution servers that can be used by a particular consumer group. You can also set the directive `PARALLEL_STMT_CRITICAL` to `BYPASS_QUEUE` for the high-priority consumer group so that parallel statements from the consumer group bypass the parallel statement queue. Default is `NULL`, which means that parallel statements are eligible for queuing.

Use the `PARALLEL_SERVER_LIMIT` directive attribute to specify the maximum percentage of the parallel execution server pool that a particular consumer group can use. The number of parallel execution servers used by a particular consumer group is counted as the sum of the parallel execution servers used by all sessions in that consumer group.

For example, assume that the total number of parallel execution servers is 32, as set by the `PARALLEL_SERVERS_TARGET` initialization parameter, and the `PARALLEL_SERVER_LIMIT` directive attribute for the consumer group `MY_GROUP` is set to 50%. This consumer group can use a maximum of 50% of 32, or 16 parallel execution servers.

Summary

In this lesson, you should have learned how to:

- Use statement queuing
- Set appropriate parameters
- Configure parameters to satisfy concurrency between parallelized statements
- Use Database Resource Manager to manage parallel statement queuing

7

In-Memory Parallel Execution

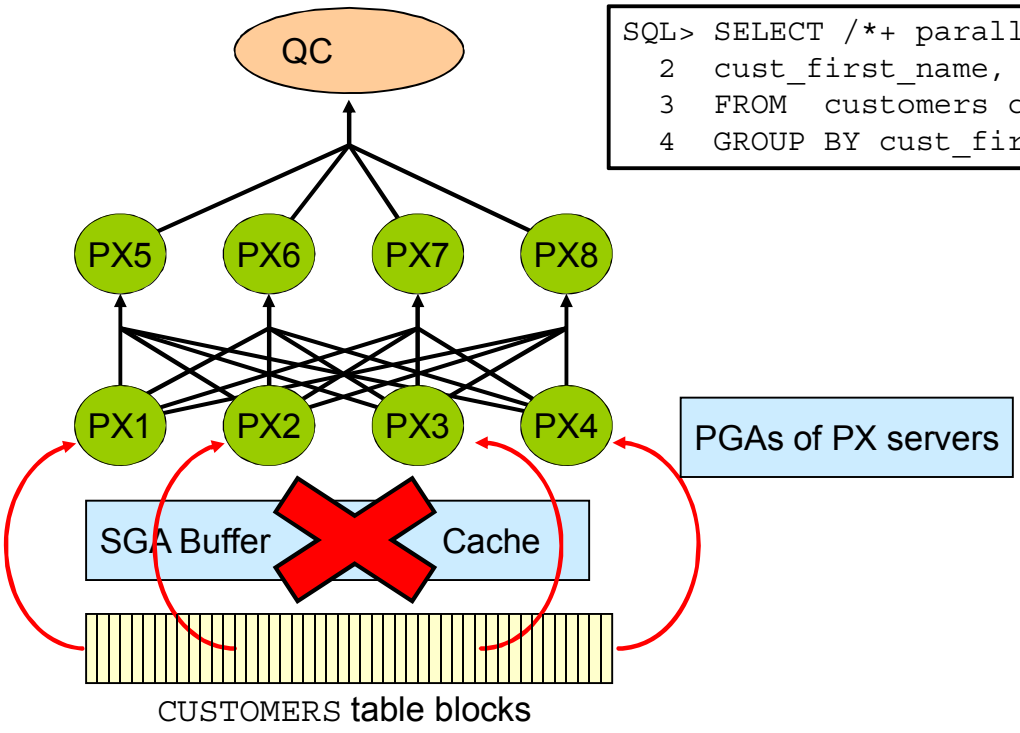
Objectives

After completing this lesson, you should be able to:

- Describe the benefits of using the In-Memory Parallel Execution feature
- Describe why queries benefit from In-Memory Parallel Execution
- Describe when queries benefit from In-Memory Parallel Execution
- Describe how to use In-Memory Parallel Execution
- Describe the enhancement of Parallel Execution in RAC with services

ORACLE

Direct Reads Versus Buffer Cache Reads




```
SQL> SELECT /*+ parallel(c,4)*/
  2  cust_first_name, count(*)
  3  FROM  customers c
  4  GROUP BY cust_first_name;
```


Direct Reads

Traditionally, parallel processing bypassed the database buffer cache for most operations, reading data directly from disk (via direct path I/O) into the PX server's private working space.

Implications of this behavior:

- Avoiding contention
- Allowing asynchronous I/O
- Avoiding scattered reads
- No cache benefits
- Checkpointing dirty buffers belonging to the objects referenced in the SQL statement

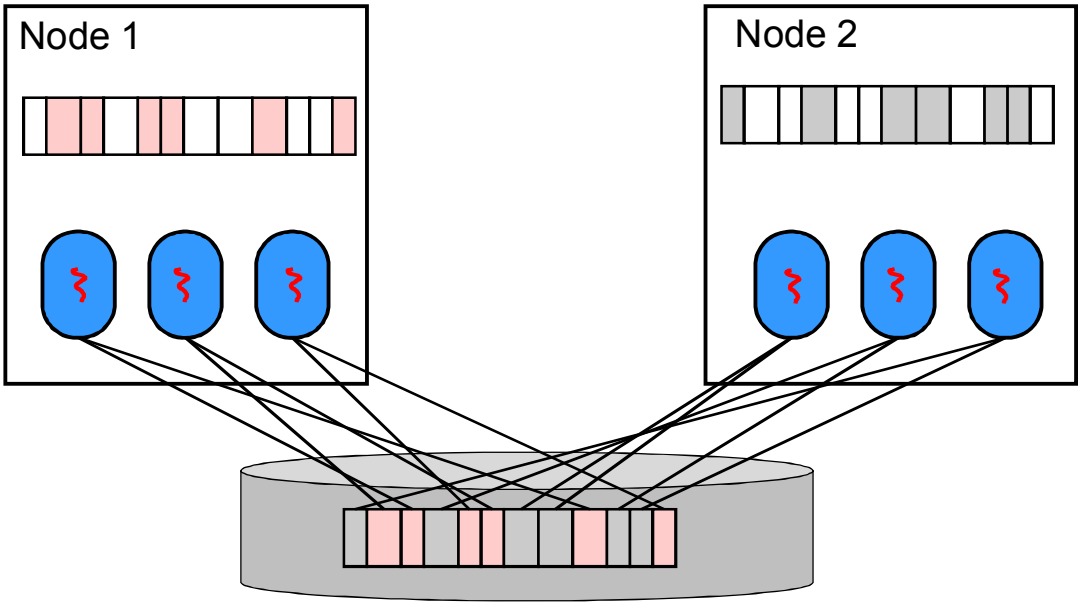
Buffer Cache Reads

There are some situations when data is read into the database buffer cache:

- Objects smaller than 2% of `DB_CACHE_SIZE`.
  But most objects accessed in parallel are larger than this limit.
- Full table scans during parallel `UPDATE/DELETE` because the `DBWR` must update the block read from the buffer cache
- `CACHE` attribute set on the table

Though reading data from memory has lower latency than reading from disks, it is very difficult to cache a large amount of data in the buffer cache. It is expensive to resource a sufficient number of high-volume memory modules and, in addition, the server is not capable of hosting such a large amount of physical memory.

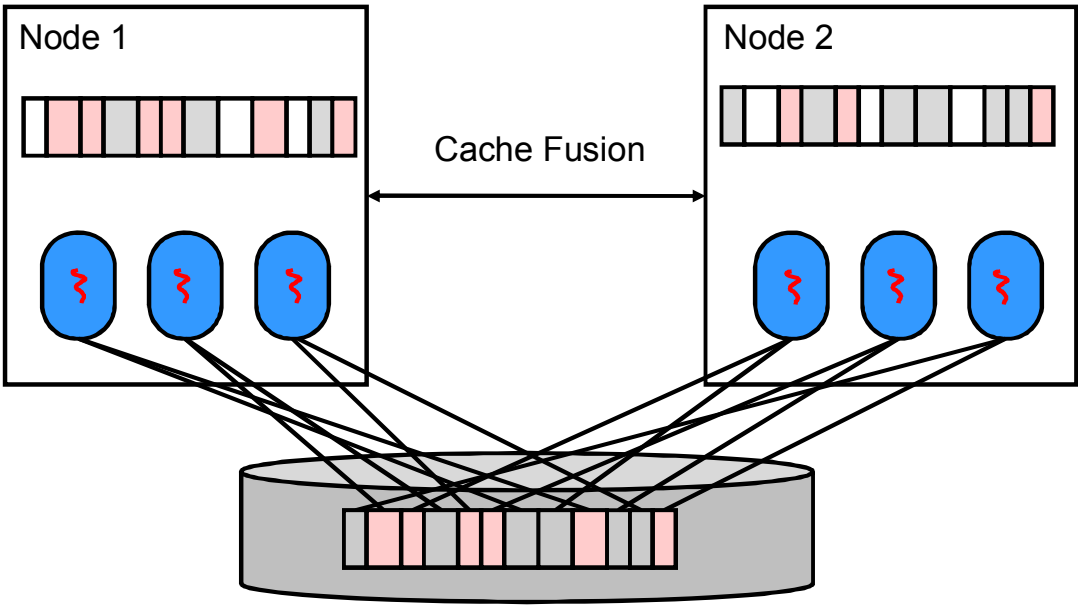
Parallel Execution and the Buffer Cache



If the Parallel Execution of one statement read part of an object into the buffer cache, then subsequent SQL statements on other nodes in the cluster would access that data via Cache Fusion. This behavior could eventually result in a full copy of that table in every buffer cache in the cluster.

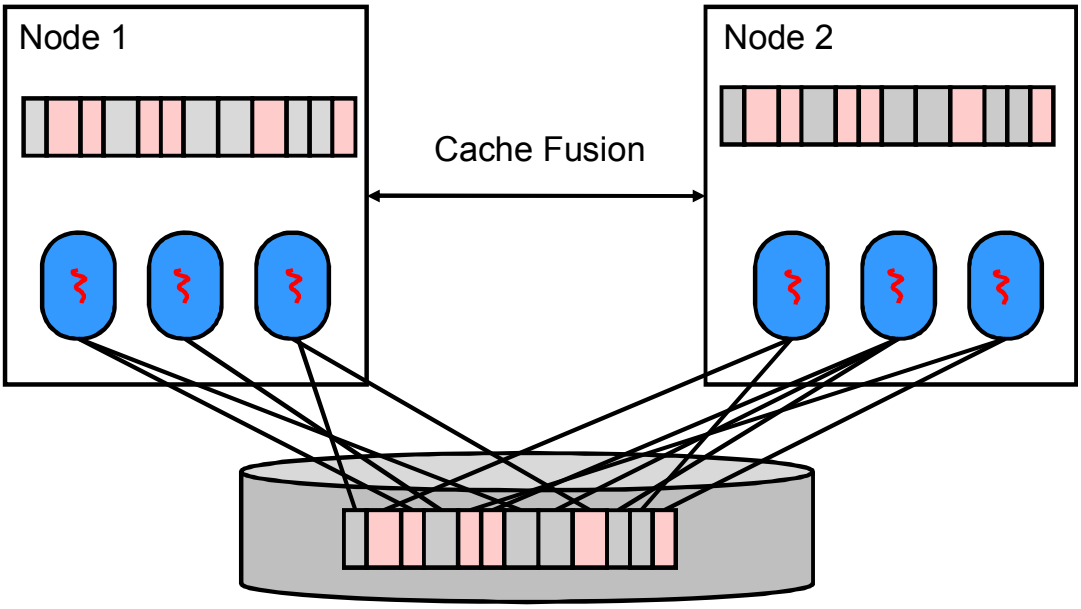
In the slide example, the instance of the first node reads the pink blocks of the table and the instance of the second node reads the gray blocks of the table.

Parallel Execution and the Buffer Cache



In the slide example, the instance of the first node now needs some of the gray blocks of the table and the instance of the second node needs some of the pink blocks of the table. The Cache Fusion becomes active and copies the gray blocks from instance 2 to instance 1 and the pink blocks from instance 1 to instance 2.

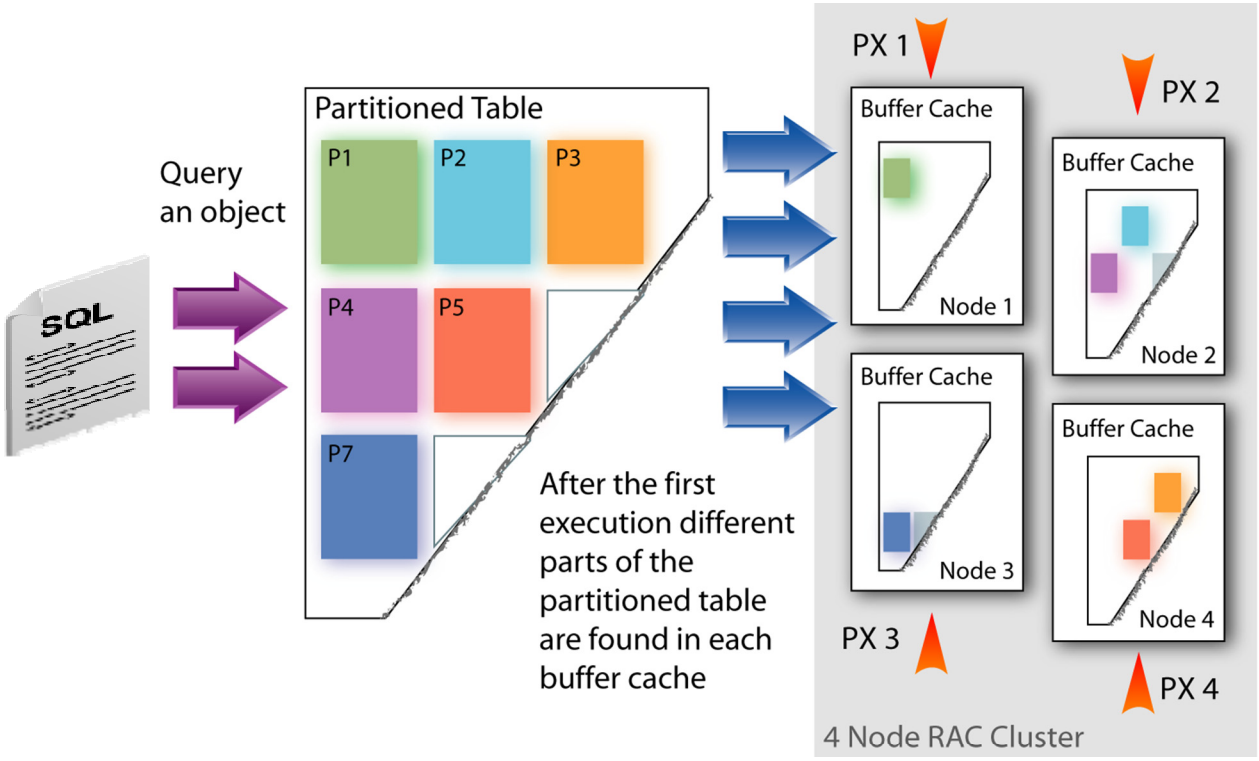
Parallel Execution and the Buffer Cache



In the slide example, the instance of the first node now needs more of the gray blocks of the table and the instance of the second node needs more of the pink blocks of the table. The Cache Fusion continues to copy the gray blocks from instance 2 to instance 1 and the pink blocks from instance 1 to instance 2.

At this stage, a whole copy of the table is in both buffer caches.

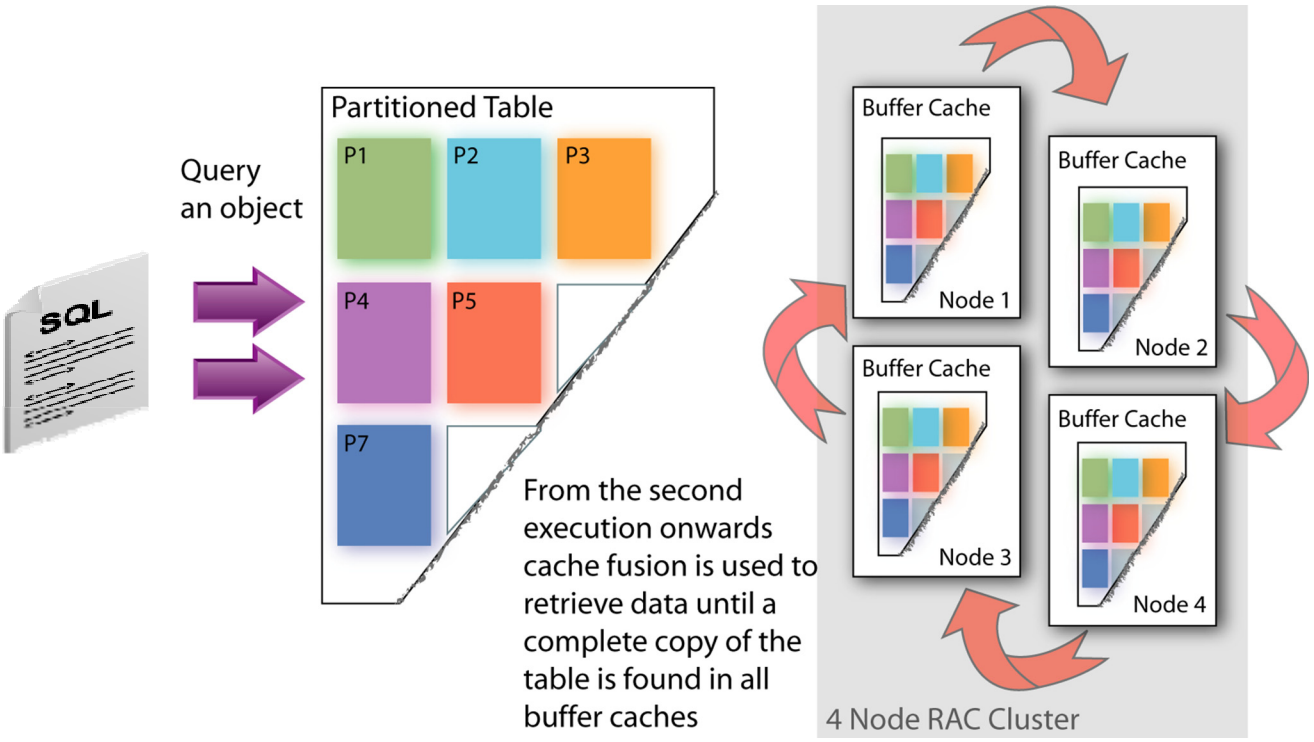
Parallel Execution and the Buffer Cache



ORACLE

In the slide example, the instance of the first node reads partition1 of the table and the instance of the second node reads partition2 of the table, and so on.

Parallel Execution and the Buffer Cache



ORACLE

In the slide example, you can see that when all instances request the other partitions already read by other instances, the Cache Fusion will copy the missing partitions from one buffer cache to the other, which leads to a full copy of the partitioned table on all instances.

Parallel Execution and the Buffer Cache

Disadvantages

- The usable size of the buffer cache is the size of a single instance, not the sum of all instance buffer caches.
- Because buffer cache is small:
  - Blocks are aged out rapidly
  - Uploading blocks in buffer cache consumes resources
- Bringing blocks in the buffer cache through cache fusion is worse than filtering on a remote instance and sending back the result.

Advantages

- It is useful only to parallel execution for metadata (extent map) or very small tables.

ORACLE

The slide explains the pros and cons of such behavior and usage.

In-Memory Parallel Execution

- Why In-Memory Parallel Execution is better
- When In-Memory Parallel Execution gets involved
- How In-Memory Parallel Execution works

Before In-Memory Parallel Execution

Customers had to manually determine which DOP a statement should run with. With no clear guide on how to determine the ideal DOP, customers would generally decide on a DOP for an object (table, index) and all statements accessing that object would use the same DOP. But one DOP may not be suitable for all statements.

The only other alternative was to use a different `PARALLEL` hint for each statement, but many third-party applications do not allow hints to be used and even if you could include a hint for every statement, it would mean a time-consuming tuning exercise every time a new statement was added to the system.

With In-Memory Parallel Execution

If you enabled `PARALLEL_DEGREE_POLICY` to `AUTO`, Oracle automatically decides whether a statement:

- Should execute in parallel and which DOP it should use
- Can execute immediately or if it should be queued until more system resources are available
- Can take advantage of the aggregated cluster memory or not (In-Memory PX)

Why In-Memory Parallel Execution?

- Business Requirement
  - Traditionally, Parallel Execution takes advantage of the I/O capacity of the system
- Evolution
  - Growing imbalance between I/O bandwidth and memory
  - Extremely large memory capacity on typical database servers
- In-Memory Parallel Execution as a solution
  - Leverages the memory capacity of the entire system
  - Uses the aggregated memory of a cluster as a whole
  - Scans data nearly 10 times faster than scanning from disk
  - Enforces access to these objects by local PX servers
  - Places object fragments on RAC nodes

ORACLE

Hardware Evolution

Up until now these parallel server processes typically bypassed the buffer cache and read the necessary data directly from disk. The main reason for this behavior was that the objects accessed by PX servers were large and would not fit into the buffer cache. Any attempt made to read these large objects into the cache would have resulted in reading one batch of blocks after the other, decreasing the performance of table access.

However, as hardware systems have evolved, the memory capacity on a typical database server has become extremely large. An Exadata Database Machine X4-2 Full Rack has an impressive 4 TB of memory for database processing, and an X4-8 has 12 TB of memory for database processing. Suddenly, using the buffer cache to hold a large object does not seem nearly impossible as it once did.

In-Memory PX takes advantage of these larger buffer caches.

Performance

Using In-Memory PX instead of Direct Path Read maximizes CPU utilization and minimizes disk I/O on the storage, which is often the cause of performance bottlenecks in parallel queries with a large amount of data.

When In-Memory Parallel Execution Works

- Decision to use the buffer cache is based on set of heuristics:
  - Ratio between buffer cache size and object size
  - Frequency at which the object is accessed
  - How much the object changes between accesses
- Possible workarounds:
  - Compression
  - Partitioning

Note: If data is not cached yet, response time may be slower than direct path during the time data is loaded into the buffer cache.

Oracle begins by determining whether the working set (group of database blocks) necessary for a query fits into the aggregated buffer cache of the system.

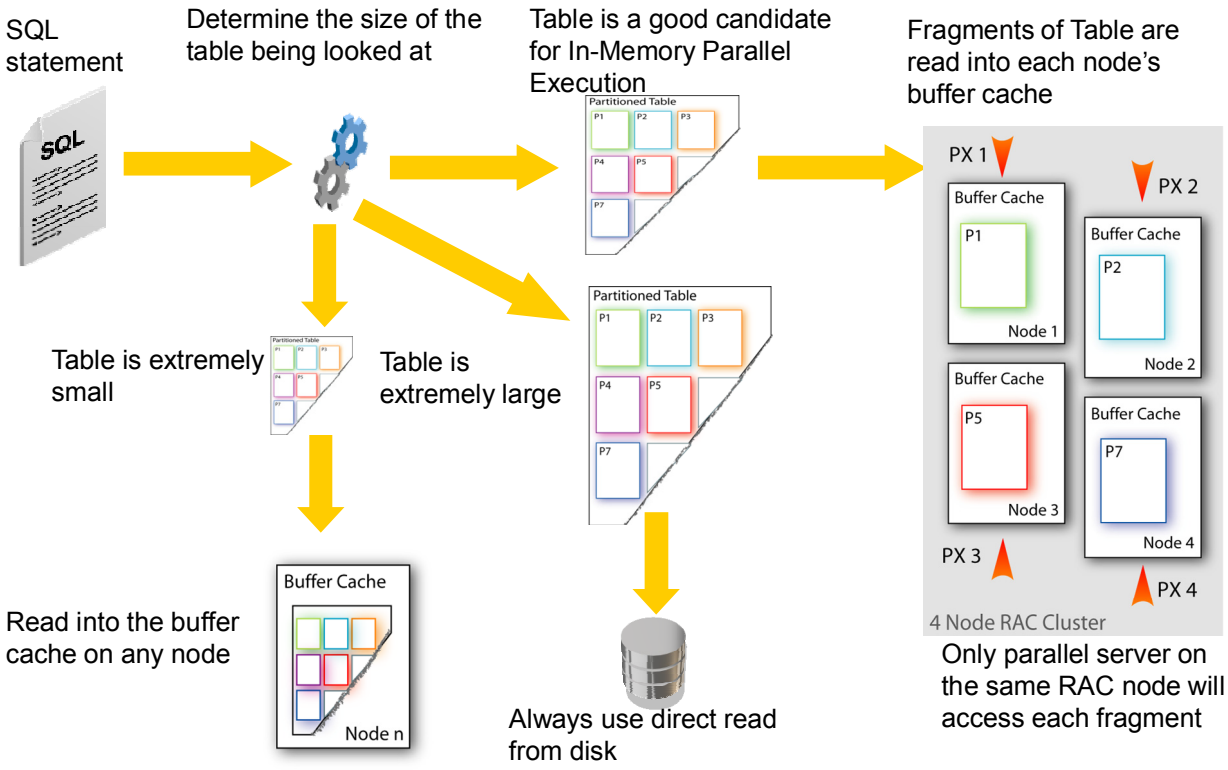
- If the working set does not fit, the objects are accessed via direct path I/O, just as they were before. The object size should not exceed 80% of the buffer cache.
- If the working set fits into the aggregated buffer cache, the blocks are distributed among the nodes and the blocks are associated with that node.

In-Memory PX uses data of database segments (such as tables, indexes) cached on the buffer cache. And it offers a high-speed SQL execution environment, eliminating the issue of storage performance.

If the target data is not cached yet, response time may be slower than direct path read because there can be some performance degradation due to the overhead to load data into the buffer cache from disks.

If PX deals with more data than the capacity of buffer size, it automatically detects it during generation of the SQL execution plan and changes to use direct path read instead.

When In-Memory Parallel Execution Works

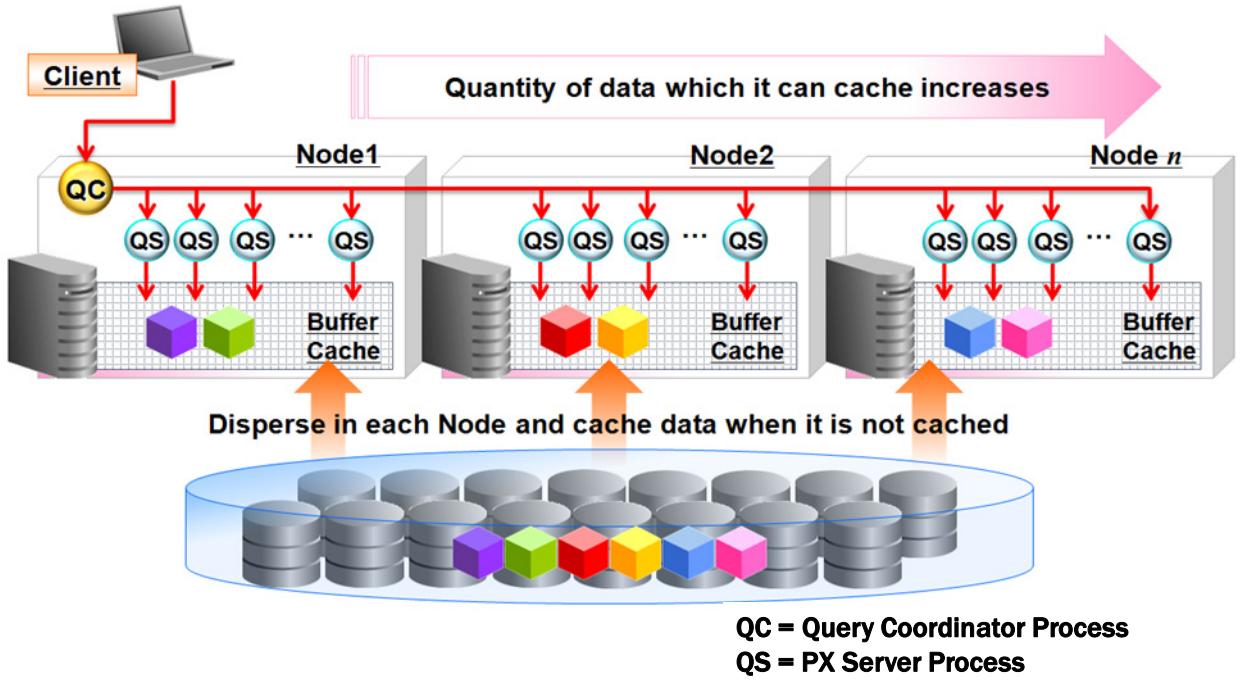


In-Memory PX with Partitioning

When using partitioned tables, there is a greater tendency for the data to be cached in the buffer cache because partition pruning reduces the actual data size to be used. There is no need to scan the irrelevant partitions, increasing the chances that the data fits within 80% of the buffer cache.

How In-Memory Parallel Execution Works

```
SQL> SELECT prod_id,sum(quantity_sold) FROM sales GROUP BY prod_id ;
```



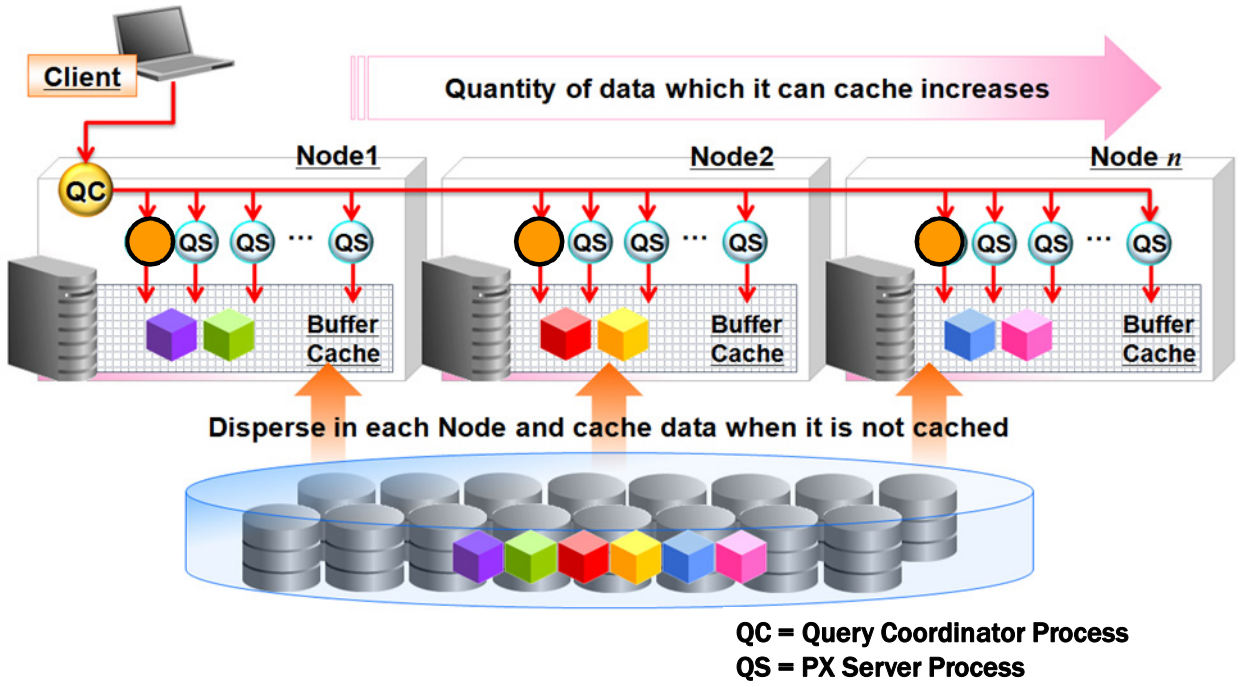
The Buffer Cache Usage

Buffer caches of each database instance are aggregated into one virtual database buffer cache and data (database segment of the table, indexes) can be spread across the buffer caches.

Therefore, a larger amount of data can be cached than a single database instance could handle, and even more can be cached by adding nodes into Oracle RAC.

How In-Memory Parallel Execution Works

```
SQL> SELECT prod_id,sum(quantity_sold) FROM sales GROUP BY prod_id ;
```



Only PX servers on the same RAC node will access each fragment of the object, all fragments being dispersed in all buffer caches.

How In-Memory Parallel Execution Works

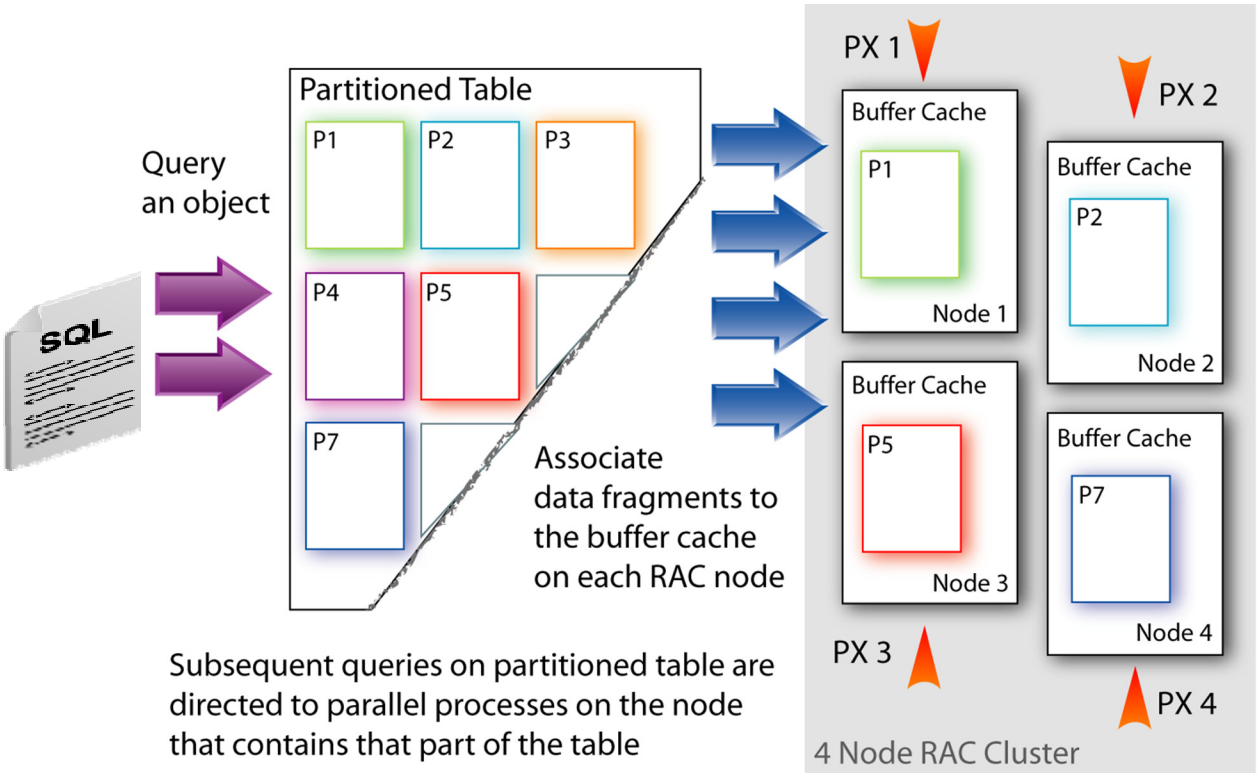
- Different behavior from Cache Fusion
- Fragments of the object are affinitized in the buffer cache on each of the active instances:
  - Based on FileNumber and ExtentNumber unless HASH partitioned
  - Automatically preventing multiple instances reading the same data from disk
  - Allowing only PX process on the same RAC node to access each of the fragments of the object

Different Behaviors

In previous releases, if the Parallel Execution of one statement read part of an object into the buffer cache, then a subsequent SQL statement on other nodes in the cluster would access that data via Cache Fusion. This behavior could eventually result in a full copy of that table in every buffer cache in the cluster.

In-Memory PX is notably different because Cache Fusion will not be used to copy the data from its original node to another node, even if a parallel SQL statement that requires this data is issued from another node. Instead Oracle uses the parallel server process on the same node (that the data resides on) to access the data and will return only the result to the node where the statement was issued.

How In-Memory Parallel Execution Works



ORACLE

In-Memory Parallel Execution and the Buffer Cache

In the slide example, you can see that when all instances request the other partitions already read by other instances, there is no copy of the missing partition from one instance to another.

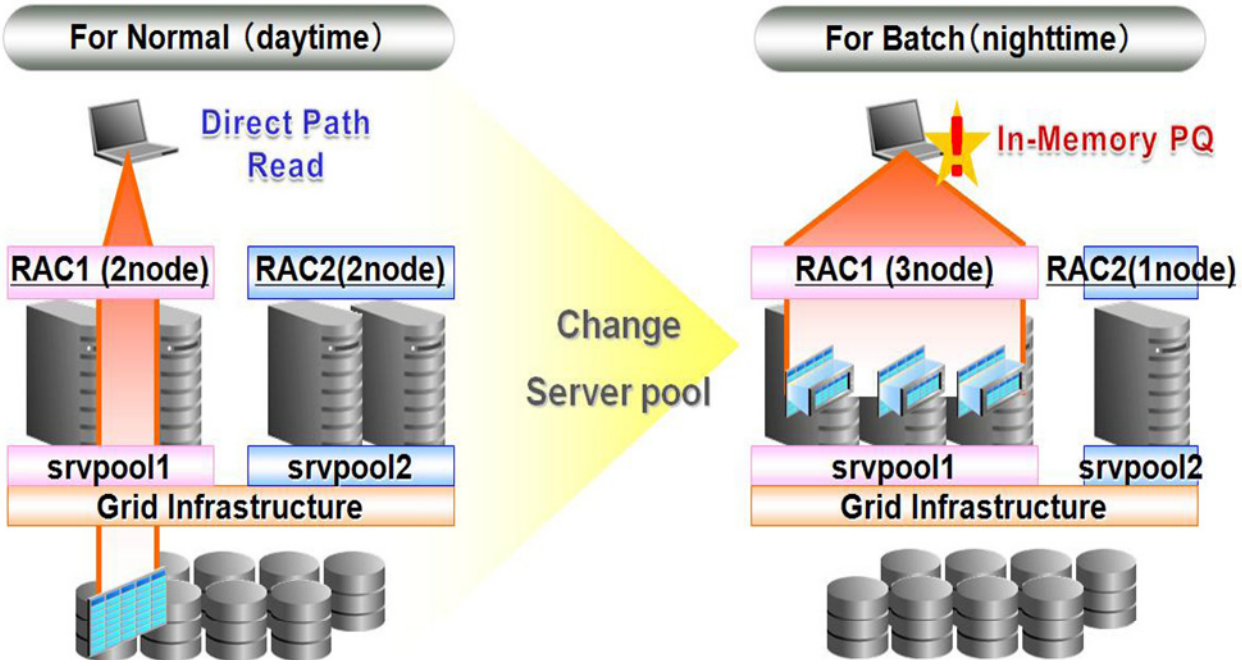
Controlling In-Memory Parallel Execution

Controlled by the `PARALLEL_DEGREE_POLICY` parameter:

- Active only when set to `AUTO`
- No possibility to turn it off with a specific parameter
- No possibility to manually control specific statements with a specific parameter

It is only active when `PARALLEL_DEGREE_POLICY` is set to `AUTO`.

Enhance In-Memory PX Using Server Pools



In-Memory Parallel Execution Using Server Pools

Assume the system environment shown in the slide, where two nodes are assigned to srvpool1 for online transactions in the daytime.

It does not satisfy the buffer cache size to use In-Memory PX with a large amount of data when executing batch processing at night.

Solution:

- Add one node into a srvpool1 temporarily by borrowing one from srvpool2.
- Increase the pool size to the required number of nodes until it satisfies the batch-processing memory requirement .
- Execute batch processing with In-Memory PX.

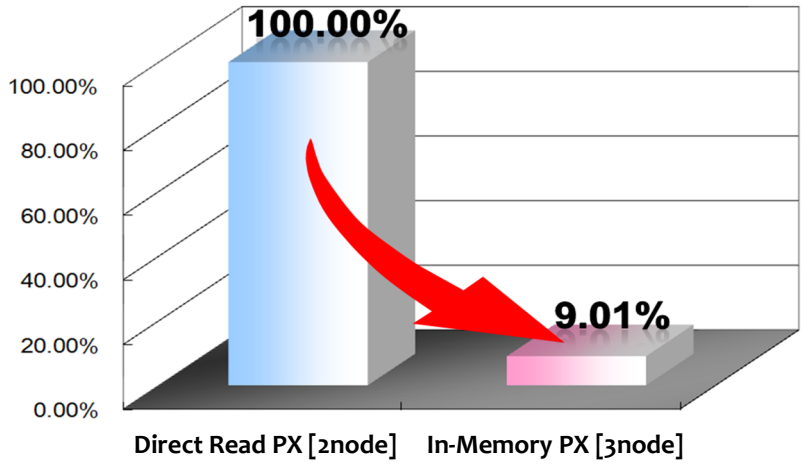
To change the configuration of each server pool, use the `srvctl` command or Oracle Enterprise Manager:

```
$ ### Reducing the nodes in srvpool2
$ srvctl stop instance –d dbname –n nodename
```

```
$ srvctl modify srvpool -g srvpool2 -u 1
$ srvctl status srvpool -g srvpool2
Server Pool Name: srvpool2
Active Servers: 1
$ ### Expanding the nodes in srvpool1
$ srvctl modify srvpool -g srvpool1 -u 3
$ srvctl status srvpool -g srvpool1
```

Enhance In-Memory PX

- Increase the number of nodes (in a server pool).
  - Query performance = 10 times faster than conventional direct-read PX
  - Increase the size of the whole aggregated database cache.
- Load data into buffer cache before executing repetitive SQL.



The slide displays the result of relative response time between executing Table Full Scan queries using direct-read PX with two nodes (left column) and In-Memory PX with three nodes (right column).

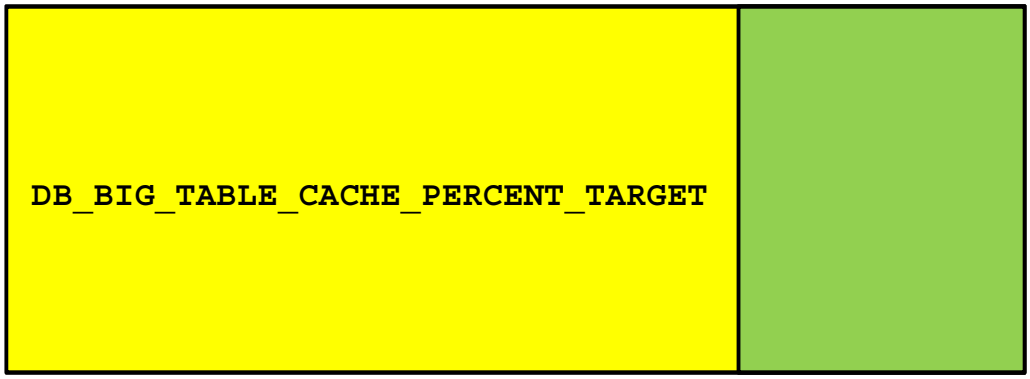
The query performance reaches 10 times faster than conventional direct-path read by increasing the number of nodes in server pool "srvpool1," which enables In-Memory PX.

In addition, increasing the total amount of buffer cache of the target Oracle RAC database, every database segment for the query is cached across the buffer cache of each database instance, making In-Memory PX usable.

As for In-Memory PX, it is supposed that every target data is cached on the buffer cache in advance. The above result is obtained after configuring server pool (2-nodes to 3-nodes) and loading all data into the buffer cache. Executing a query before the target data is loaded into the buffer cache means the response time becomes slower than direct-path read, because there is an overhead for caching.

Enhance In-Memory PX: Automatic Big Table Caching

- Use Automatic Big Table Caching in single instance and Oracle RAC environments.
  - Supports parallel queries only
  - Reserves a percentage of the Buffer Cache for table scans
  - Is an alternative to using least recently used (LRU) mechanism for cached reads or direct reads to disk



Database Buffer Cache

Automatic big table caching integrates queries with the buffer cache to enhance the in-memory query capabilities of Oracle Database, in both single instance and Oracle RAC environments. In Oracle Real Application Clusters (Oracle RAC) environments, this feature is supported only with parallel queries. In single instance environments, this feature is supported with both parallel and serial queries.

The cache section reserved for the big table cache is used for caching data for table scans. Although the big table cache is primarily designed to enhance performance for data warehouse workloads, it also improves performance in Oracle Database running mixed workloads.

Automatic big table caching uses temperature and object-based algorithms to track medium and big tables. Oracle does cache very small tables, but automatic big table caching does not track these tables.

To enable automatic big table caching for serial queries, you must set a value (percentage) for the `DB_BIG_TABLE_CACHE_PERCENT_TARGET` initialization parameter. Additionally, you must set the `PARALLEL_DEGREE_POLICY` initialization parameter to `AUTO` or `ADAPTIVE` to enable parallel queries to use automatic big table caching. In Oracle RAC environments, automatic big table caching is only supported in parallel queries so both settings are required.

Note: Available starting with Oracle Database 12*c* Release 1 (12.1.0.2)

Automatic Big Table Caching Views

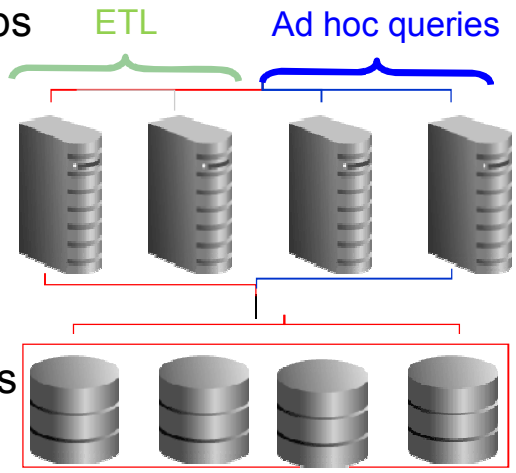
- Use `V$BT_SCAN_CACHE` to determine:
  - Current ratio of the big table cache section to the buffer cache
  - Number of objects tracked by the big table cache
  - Minimum temperature of any object that is allowed
- Use `V$BT_SCAN_OBJ_TEMPS` to determine:
  - Data object number of item being tracked
  - Size of object being tracked
  - Temperature of object being tracked
  - Number of blocks cached for object being tracked
  - Caching policy of object being tracked
    - `MEM_ONLY`: Fully cached in memory
    - `MEM_PART`: Partially cached in memory
    - `DISK`: Not cached in memory
    - `INVALID`: Caching policy not valid

There are two dynamic views dedicated to reporting on the Automatic Big Table Caching feature. The `V$BT_SCAN_CACHE` view shows the parameters and status of the big table cache section. The `V$BT_SCAN_OBJ_TEMPS` view shows the active objects currently tracked by the big table cache.

Controlling Parallel Execution History: Oracle 10*g*

Static Parameters: Create two groups

```
sid1.INSTANCE_GROUPS = ETL,ALL
sid2.INSTANCE_GROUPS = ETL,ALL
sid3.INSTANCE_GROUPS = AHOC,ALL
sid4.INSTANCE_GROUPS = AHOC,ALL
```



Dynamic system/session parameters

```
sid1.PARALLEL_INSTANCE_GROUP=ALL
sid2.PARALLEL_INSTANCE_GROUP=ETL
sid3.PARALLEL_INSTANCE_GROUP=AHOC
sid4.PARALLEL_INSTANCE_GROUP=AHOC
```

ORACLE

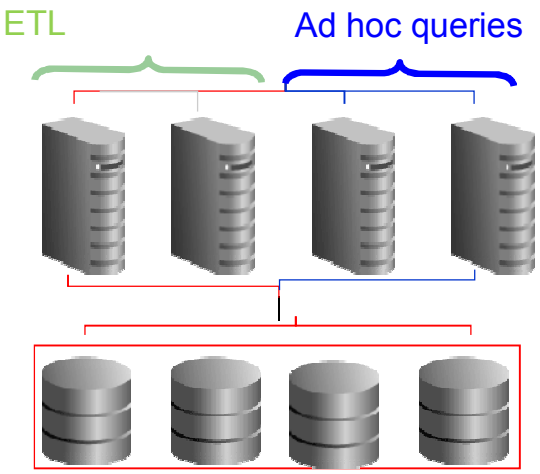
In Oracle Database 10*g*, parallel execution in RAC instances is controlled with two initialization parameters: `INSTANCE_GROUPS` and `PARALLEL_INSTANCE_GROUP`. Both must be used together in Oracle Database 10*g*. In order for parallel execution to occur in a session where you have `PARALLEL_INSTANCE_GROUP` set, the value for `PARALLEL_INSTANCE_GROUP` has to be found in the list of groups specified by `INSTANCE_GROUPS` for that instance. In the example in the slide, a parallel statement started on SID2 runs on SID1 and SID2 with the PX servers of both instances.

Note: These parameters are deprecated in Oracle Database 11*g* and Oracle Database 12*c*. They are retained for backwards compatibility.

Interconnect

By default, in an Oracle RAC environment, a SQL statement executed in parallel can run across all of the nodes in the cluster. For this cross-node or internode parallel execution to perform, the interconnection in the Oracle RAC environment must be sized appropriately because internode parallel execution may result in heavy interconnect traffic. If the interconnection has a considerably lower bandwidth in comparison to the I/O bandwidth from the server to the storage subsystem, it may be better to restrict the parallel execution to a single node or to a limited number of nodes. Internode parallel execution does not scale with an undersized interconnection.

Controlling Parallel Execution History: Oracle 11*g*



Use RAC Services:

- Create two services: ETL, AHOC

```
$ srvctl add service -d database_name -s ETL -r sid1, sid2
```

```
$ srvctl add service -d database_name -s AHOC -r sid3, sid4
```

ORACLE

With Oracle Database11*g*, PX processes are integrated with services so that setting up `INSTANCE_GROUPS` and setting `PARALLEL_INSTANCE_GROUP` are no longer necessary. Instead of instance groups, you can use the service names directly (for example, `ALTER SESSION SET PARALLEL_INSTANCE_GROUP = service_name`). The `INSTANCE_GROUPS` parameter is deprecated in Oracle Database 11*g*; you can still use it, but it is retained for backward compatibility only.

You also do not need to set `PARALLEL_INSTANCE_GROUP` explicitly in order to restrict PX processes to a subset of instances. If you execute a SQL statement in parallel, the default behavior is that parallel processes run only on the instances that offer the service with which you originally connected to the database.

Note: Refer to My Oracle Support note “RAC: How to control parallel execution across instances in Oracle 11*g*.” (Doc ID 1207190.1)

Using PARALLEL_FORCE_LOCAL Parameter

Set to TRUE:

- Controls parallel execution in an Oracle RAC environment
- Forces the degree limit computation to consider a restricted list of instance resources to evaluate the DOP
  - If a user connects to a RAC instance, parallel statements are executed with the PX processes of the instance.
  - If a user connects to a RAC service that encompasses several RAC nodes, parallel statements are executed with the PX processes of the nodes where the initial connection was mapped.
- Is useful when the interconnect is slow

Monitor which nodes can be used: V$PX_INSTANCE_GROUP

Within a cluster, the instance with the highest number of CPUs is used to determine the degree limit.

Example: In a cluster with two 4-CPU machines, one 2-CPU machine, and one 8-CPU machine, 8 is the higher number and is used to compute the degree limit with:

```
threads per cpu * #cpu
```

You can enable this feature by setting PARALLEL_FORCE_LOCAL to TRUE.

The default value is FALSE.

In a RAC environment, data sharing between PX server processes (producers and consumers) is achieved through the interconnect. This can considerably impact the parallel execution processing.

Summary

In this lesson, you should have learned how to:

- Benefit from the use of In-Memory Parallel Execution
- Use In-Memory Parallel Execution
- Use the RAC services to control parallel execution in RAC

8

Parallel Execution and Data Loading

Objectives

After completing this lesson, you should be able to apply parallel execution with the following:

- Data Pump
- SQL*Loader
- External tables

When loading large amounts of data, whether it is an initial or incremental load, the main goal is often to get the data into the database in the most performance-oriented manner. This is certainly true with large systems such as a data warehouse.

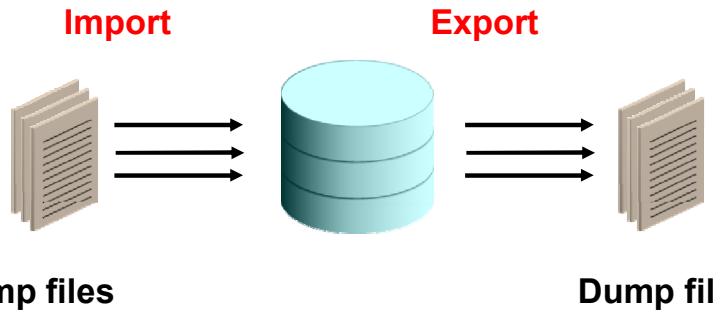
Oracle Database 12*c* supports several embedded facilities for loading data including external tables, SQL*Loader, and Oracle Data Pump.

Although SQL*Loader and Data Pump are mentioned, this lesson focuses on loading data by using the most scalable method, which is with external tables.

Note: For information about Oracle's data loading facilities, see the *Oracle Database Utilities12c Release 1 (12.1)* documentation guide.

Parallel Processing with Oracle Data Pump

- When you use Data Pump, the `PARALLEL` clause specifies whether direct loads can operate in multiple concurrent sessions to do the following:
  - Load data into the database
  - Unload data to external files



The Data Pump utilities are designed especially for very large databases. If your site has large quantities of data versus metadata, then you should experience a significant increase in performance compared to the original Export and Import utilities.

With Data Pump, you can:

- Import data by using the `impdp` utility
- Export data by using the `expdp` utility

Data Pump Import

Import is a utility for loading an export dump file set into a database. You can also use Import to load a destination database directly from a source database with no intervening files, which enables operations to run concurrently, minimizing total elapsed time.

Data Pump Export

Export is a utility for unloading data and metadata into a set of operating system files called a dump file set. The dump file set is made up of one or more binary files that contain table data, database object metadata, and control information.

Using `expdp` to Export in Parallel

- For example, a full database Export might have up to three parallel processes.

```
$ expdp hr FULL=y DUMPFILE=dpump_dir1:full1%U.dmp,
  dpump_dir2:full2%U.dmp
  FILESIZE=2G PARALLEL=3 LOGFILE=dpump_dir1:expfull.log
  JOB_NAME=expfull
```


Full Database Export

In this example:

- Up to three parallel processes will be used in the database export, as specified in the `PARALLEL` clause. (Because this is a full database export, all data and metadata in the database will be exported.)
- Dump files such as `full101.dmp`, `full201.dmp`, and `full102.dmp` will be created in a round-robin fashion in the directories pointed to by the `dpump_dir1` and `dpump_dir2` directory objects. (For best performance, these directories should be on separate I/O channels.)
- Each dump file will be up to 2 GB in size, as necessary. Initially, up to three files will be created. More files will be created if needed.
- The job and master table will have a name of `expfull`. The log file will be written to `expfull.log` in the `dpump_dir1` directory.

Using a Substitution Variable

The dump file names can be specified by using the substitution variable, `%U`. If `%U` is used, the export or import utility examines each file that matches the template to locate all files that are part of the dump file set. The `%U` expands to a 2-digit incrementing integer starting with 01.

Using `impdp` to Import in Parallel

- For example, you might perform an import that has up to three parallel processes.

```
$ impdp hr DIRECTORY=dpump_dir1 LOGFILE=parallel_import.log
  JOB_NAME=imp_par3 DUMPFILE=par_exp%U.dmp PARALLEL=3
```

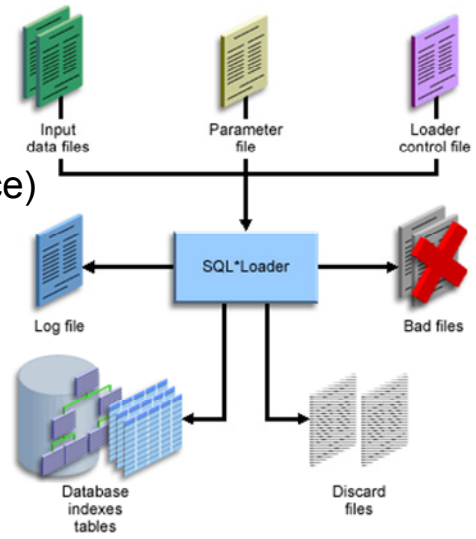
The code example shows a parallel import that uses the `impdp` utility:

- Up to three parallel processes will be used in the import, specified by the `PARALLEL` clause.
- The following three files are identified as the import source, referenced by the substitution variable in the `DUMPFILE` clause: `par_exp01.dmp`, `par_exp02.dmp`, and `par_exp03.dmp`.
- The job will have a name of `imp_par3`, and the log file will be written to `parallel_import.log` in the `dpump_dir1` directory.

Note: Parallel processes can read multiple dump files or even chunks of the same dump file concurrently. Therefore, data can be loaded in parallel even if there is only one dump file, as long as that file is large enough to contain multiple file offsets.

Parallel Processing with SQL*Loader

- Loads data from:
  - Flat files into tables of an Oracle database
  - Multiple data files during the same load session
- Can load data by using either:
  - Conventional (flexible) load
  - Direct path (better performance)



ORACLE

When you have a large amount of data to load quickly, a direct path load that uses SQL*Loader can quickly load and index large amounts of data. It can also load data into either an empty or nonempty table.

SQL*Loader permits multiple, concurrent sessions to perform a *parallel direct path load* into the same table, or into the same partition of a partitioned table. This approach improves the performance of a direct path load, given the available resources on your system.

Using PARALLEL Clause with SQL*Loader

- An example of a parallel direct path load is as follows:

```
$ sqlldr USERID=scott CONTROL=load1.ctl DIRECT=TRUE PARALLEL=true
$ sqlldr USERID=scott CONTROL=load2.ctl DIRECT=TRUE PARALLEL=true
$ sqlldr USERID=scott CONTROL=load3.ctl DIRECT=TRUE PARALLEL=true
```

- Degree of parallelism depends on the number of files used in SQL*Loader.

A popular and efficient way to load data into a data warehouse or data mart is to perform a parallel load by using SQL*Loader with the DIRECT and PARALLEL options.

When you perform a parallel load in this way:

- SQL*Loader creates temporary segments for each concurrent session and then merges the segments upon completion.
- The segment created from the merge is then added to the existing segment in the database above the segment's high-water mark.
- The last extent used of each segment for each loader session is trimmed of any free space before being combined with the other extents of the SQL*Loader session.

Code Example

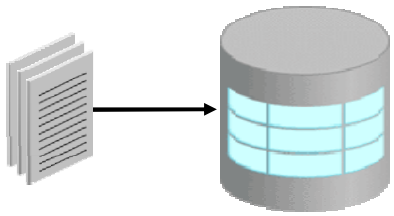
This method of data loading is enabled by setting both the DIRECT and the PARALLEL parameters to true, as shown in the code example.

Note: Be aware that parallelism is user managed. Setting the PARALLEL parameter to true allows only multiple, concurrent, direct-path load sessions. The degree of parallelism obtained is limited by the number of files used by SQL*Loader.

Parallel Processing with External Tables

External tables are tables where the data is stored outside the database in text files and binary files.

- External tables enable you to access data in external sources as if it were in a table in the database.
- Data can be queried directly and in parallel without first requiring the external data to be loaded in the database.
- An external table describes how the external data must be presented to the database.
- The metadata for an external table is created by using a `CREATE TABLE` statement.



ORACLE

Oracle external table functionality enables you to use external data as a virtual table that can be queried directly and in parallel without requiring the external data to be first loaded in the database.

The main difference between external tables and regular tables is that an external table is a read-only table whose metadata is stored in the database but whose data is stored in files outside the database. Other considerations include:

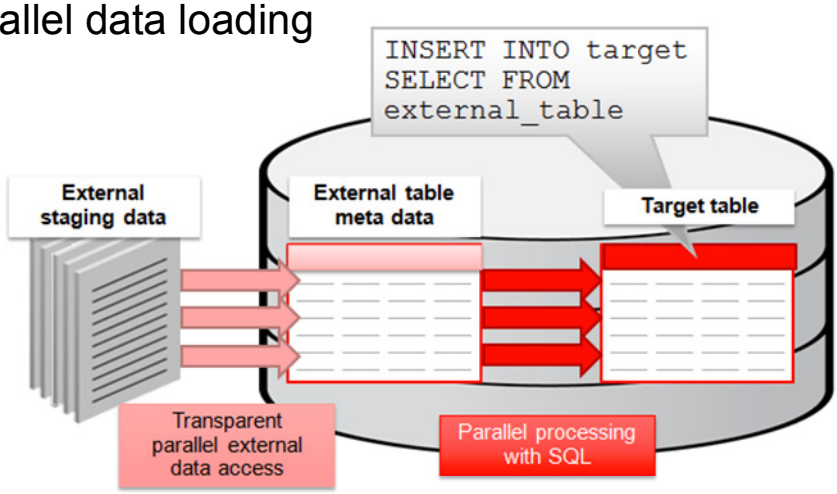
- The database uses the metadata describing external tables to expose their data as if they were relational tables, so you must always have the external data accessible when using an external table.
- Because the data resides outside the database in a non-Oracle format, you cannot create any indexes on the data either.

External tables also enable the pipelining of the loading phase with the transformation phase. The transformation process can be merged with the loading process without any interruption of the data streaming. It is no longer necessary to stage the data inside the database for further processing inside the database, such as comparison or transformation.

For example, the conversion functionality of a conventional load can be used for a direct-path `INSERT INTO` statement in conjunction with the `SELECT` from an external table.

Goals of Parallel Loading with External Tables

- Access of external staging files that will perform and scale
- Proper setup of the target database to perform highly scalable and parallel operations
- Optimal parallel data loading



ORACLE

Best Practice for Parallel Data Loading

For large systems such as data warehouse environments, Oracle's best performing approach to loading data is through the use of flat files and external tables. External files can be queried directly and in parallel by using the full power of SQL, PL/SQL, and Java. The diagram above illustrates the architecture and the components of external tables.

To ensure performant and scalable data loading by using external tables, each individual component and operation should be designed to avoid bottlenecks in the data flow from the external staging files into the target table. The goal is to:

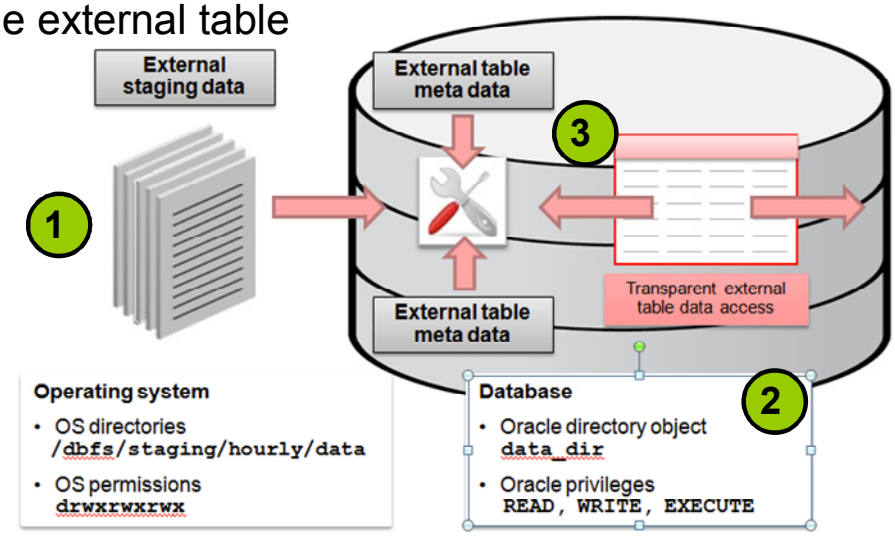
- Provide access to the external staging files
- Set up the target database so that it will perform highly scalable and parallel operations
- Enable optimal parallel data loading for your business

The remainder of this lesson will provide an overview of these three main areas. For details about each of these topics, refer to the Oracle White Paper**:** *Performant and scalable data loading with Oracle Database 11g*.

Preparing for Data Loading

You must:

1. Provide access to external data (staging) files
2. Create a Directory object
3. Create the external table



To load data, you must provide the following:

1. External data files (also known as staging files)
   - Staging files must be in a known format that is accessible from the machine where the database instance runs.
   - To guarantee optimal performance, the staging files should not be placed on the same disks that are used by RDBMS, because reading the external data files will compete with the data insertion of the database.

There are two objects in the database:

2. **A Directory object:** Creates a logical pointer to a location outside the database where the external data files reside
3. **An external table:** Holds the metadata of both the data format in the external files and the internal database representation of this "table"

Note: Oracle Database 12*c* allows an external program to be specified as part of the external table definition. For example, an unzip program can be invoked as a preprocessor before consuming the data. While the database is accessing the compressed external files, the decompression will take place outside the database, without the need to restage the data in an uncompressed format before consuming it.

Creating the External Table

Use normal CREATE TABLE syntax with an ORGANIZATION EXTERNAL clause and a PARALLEL clause.

```
CREATE TABLE sales_external (…)
ORGANIZATION EXTERNAL
( TYPE ORACLE_LOADER
  DEFAULT DIRECTORY data_dir1
  ACCESS PARAMETERS
  ( RECORDS DELIMITED BY NEWLINE
    BADFILE bad_dir: 'sh%a_%p.bad'
    LOGFILE log_dir: 'sh%a_%p.log'
    FIELDS TERMINATED BY '|'
    MISSING FIELD VALUES ARE NULL )
    LOCATION (data_dir1:'sales_1.csv', data_dir1:'sales_2.csv'))
PARALLEL
REJECT LIMIT UNLIMITED;
```


An external table is created by using the standard CREATE TABLE syntax except that it requires an additional clause, ORGANIZATION EXTERNAL. ORGANIZATION EXTERNAL, which provides options that enable access to the external data files. To enable parallel execution, you must also include the PARALLEL clause.

The code example includes the following:

- TYPE: Defines the type of the access driver used for reading the external staging files. This can be either ORACLE_LOADER or ORACLE_DATAPUMP.
- DEFAULT DIRECTORY: Defines the directories objects that contain the staging files and defines the location for the log files and bad files for the load
- ACCESS PARAMETERS: Define the behavior of the access driver, including a description of the external data format (such as record delimiters), field separators, and default value for missing fields
- LOCATION: Defines the location and names of the external files, using the directory object : file name format
- The PARALLEL clause (separate from the ORGANIZATION EXTERNAL clause): Is used to specify the DOP. If no number is specified, the default DOP is used.

Note: If the staging files were compressed, the PREPROCESSOR option would be specified within the ORGANIZATION EXTERNAL clause.

Loading Data in Parallel

A sample CTAS statement could look like this:

```
CREATE TABLE <target_table>
PARALLEL 8
AS SELECT /*+ PARALLEL(64) */ *
   FROM <external_table>;
```


After ensuring that a database environment provides sufficient parallelism for your operation, as discussed previously in this seminar, you have to ensure that your load is actually running in parallel.

Code Example

The CTAS command will run with a degree of parallelism of 64, assuming the system provides 64 PX server processes and the user session is not constrained to a lower degree of parallelism.

Note

If you use Auto DOP, the DOP will be automatically derived by Oracle based on the more expensive operation of the statement: the `SELECT` or the `INSERT (CREATE TABLE)` portion.

If you are using manual DOP, you have to ensure that the desired degree of parallelism will be used by the `SELECT` and/or the `INSERT` portion of the statement. The `SELECT` portion is derived by following the rules for parallelizing a SQL operation13, and the degree of parallelism for the `INSERT (CREATE TABLE)` portion is derived by the parallel declaration in the `CREATE TABLE` command or the parallel setting of the target table, respectively.

For the overall statement, the higher degree of parallelism for both the `SELECT` and `INSERT` branch is chosen.

Unloading Data in Parallel

A sample CTAS statement could look like this:

```
CREATE TABLE inventories_xt3
 ORGANIZATION EXTERNAL
 (
   TYPE ORACLE_DATAPUMP
   DEFAULT DIRECTORY def_dir1
   LOCATION ('inv_xt1.dmp', 'inv_xt2.dmp', 'inv_xt3.dmp')
 )
 PARALLEL 3
 AS SELECT * FROM inventories;
```


When the ORACLE_DATAPUMP access driver is used to export data, the dump file must be on a disk large enough to hold all the data being written. If there is insufficient space for all of the data, then an error is returned for the CREATE TABLE AS SELECT statement.

One way to alleviate the problem is to create multiple files in multiple directory objects (assuming those directories are on different disks) when executing the statement.

Code Example

The code example uses this approach to export data in parallel to an external table:

- Three files are created by specifying multiple locations in the LOCATION clause.
- The PARALLEL clause is used to specify up to three processes.
- Each parallel I/O server process that is created to populate the external table writes to its own file. Therefore, the number of files in the LOCATION clause should match the DOP because each I/O server process requires its own file.

Note

- Any extra files that are specified will be ignored.
- If there are not enough files for the degree of parallelization specified, then the DOP is lowered to match the number of files in the LOCATION clause.

When the ORACLE_DATAPUMP access driver is used to load data from an external table, consider the following:

- The degree of parallelization is not linked to the number of files in the LOCATION clause when reading from ORACLE_DATAPUMP external tables.
- However, the external tables functionality treats each data file specified on the LOCATION clause as a single granule. To make the best use of parallel processing with the PREPROCESSOR clause, the data to be loaded should be split into multiple files (granules). This is because external tables dynamically limit the degree of parallelism to the number of data files present.
- For example:
  - If you specify a degree of parallelism of 12, but have only 8 data files, then in effect the degree of parallelism is 8, because 8 PX server processes will be busy and 4 will be idle.
  - The best practice is not to have any idle PX server processes. So if you do specify a degree of parallelism, then ideally it should be no larger than the number of data files so that all PX server processes are kept busy.

Summary

In this lesson, you should have learned how to apply parallel execution with the following:

- Data Pump
- SQL*Loader
- External tables

9 Diagnose, Troubleshoot, and Trace Parallel Execution

Objectives

After completing this lesson, you should be able to:

- Diagnose whether parallel execution works
- Troubleshoot situations when parallel processing does not proceed as desired
- Trace parallel executions to provide information to Oracle Customer Support

Diagnosing and Troubleshooting Tools

Diagnostic tools to troubleshoot parallel executions include:

- Enterprise Manager SQL Monitoring
- `EXPLAIN PLAN`
- `V$PQ_SESSTAT` view
- `_PX_TRACE` event
- Wait events
  - PX slave connection
  - PX slave release
  - PX create server
  - PX Send Wait
  - Parallel Query Statement Queuing
    - PX Queuing: statement queue
    - enq: JX - SQL statement queue
- For more information, refer to “Master Note Parallel Execution Wait Events (Doc ID 1097154.1).”

ORACLE

Parallel Execution Wait events

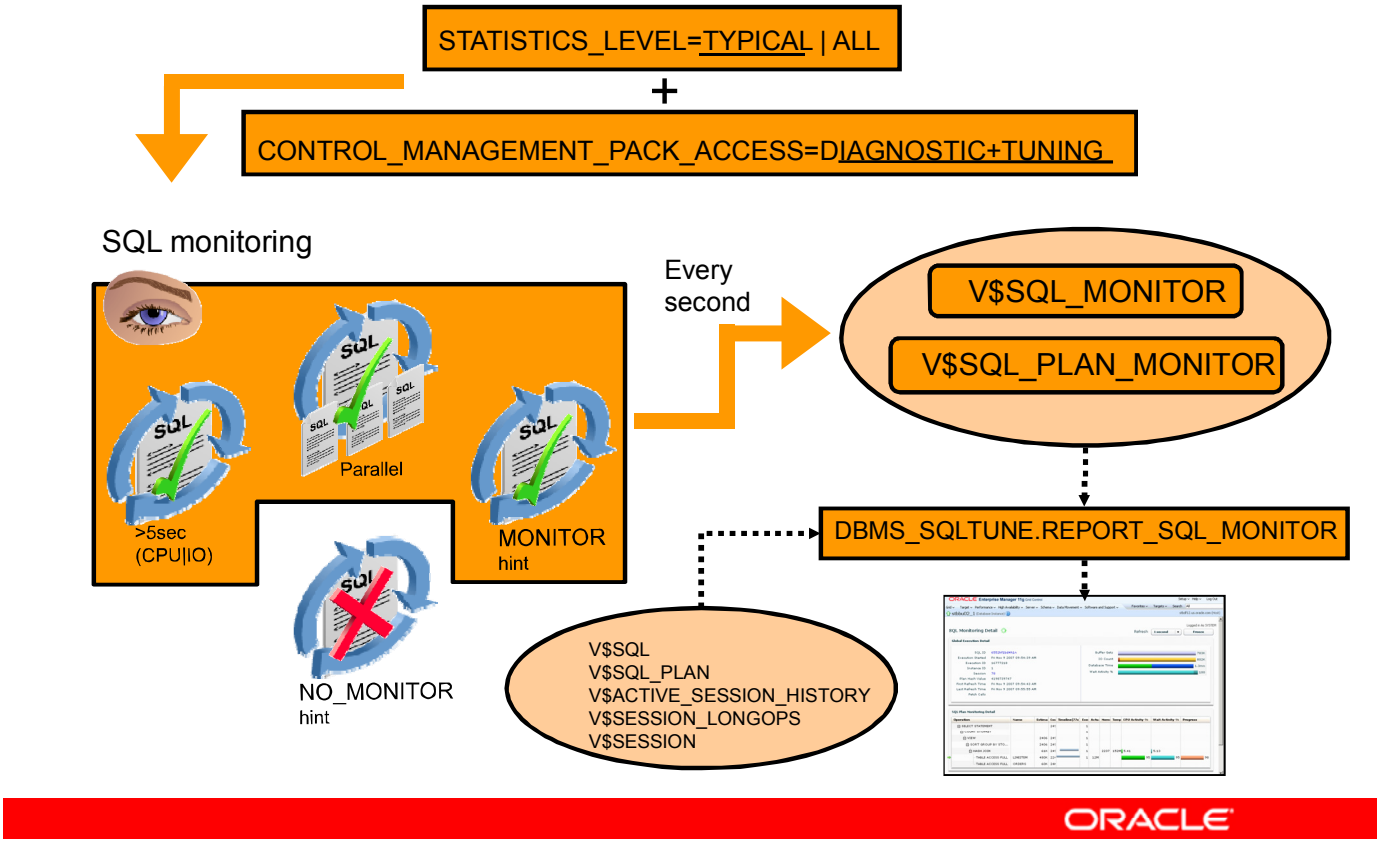
Refer to ”Master Note Parallel Execution Wait Events (Doc ID 1097154.1)” in My Oracle Support Knowledge base.

Other Views for Parallel Execution Tuning and Troubleshooting

- The `V$PX_PROCESS` view contains information about the parallel processes, including status, session ID, process ID, and other information.
- The `V$PQ_SLAVE` view lists statistics for each of the active parallel execution servers on an instance.
- The `V$PQ_SYSSTAT` view lists system statistics for parallel queries. After you have run a query or DML operation, you can use the information derived from `V$PQ_SYSSTAT` to view the number of parallel execution server processes used, and other information for the system.

Note: To obtain detailed information about these views, refer to *Oracle Database Reference 12c Release 1 (12.1).*

SQL Monitoring: Overview



The SQL monitoring feature is enabled by default when the `STATISTICS_LEVEL` initialization parameter is either set to `ALL` or `TYPICAL` (the default value).

Additionally, the `CONTROL_MANAGEMENT_PACK_ACCESS` parameter must be set to `DIAGNOSTIC+TUNING` (the default value) because SQL monitoring is a feature of the Oracle Database Tuning Pack.

By default, SQL monitoring is automatically started when a SQL statement runs parallel, or when it has consumed at least five seconds of the CPU or I/O time in a single execution.

As mentioned, SQL monitoring is active by default. However, two statement-level hints are available to force or prevent a SQL statement from being monitored. To force SQL monitoring, use the `MONITOR` hint. To prevent the hinted SQL statement from being monitored, use the `NO_MONITOR` hint.

You can monitor the statistics for SQL statement execution by using the `V$SQL_MONITOR` and `V$SQL_PLAN_MONITOR` views.

After monitoring is initiated, an entry is added to the dynamic performance `V$SQL_MONITOR` view. This entry tracks key performance metrics collected for the execution, including the elapsed time, CPU time, number of reads and writes, I/O wait time, and various other wait times. These statistics are refreshed in near real time as the statement executes, generally once every second.

After the execution ends, monitoring information is not deleted immediately, but is kept in the V$SQL_MONITOR view for at least one minute. The entry is eventually deleted so that its space can be reclaimed as new statements are monitored.

The V$SQL_MONITOR and V$SQL_PLAN_MONITOR views can be used in conjunction with the following views to obtain additional information about the execution that is monitored:

VSQL, VSQL_PLAN, V$ACTIVE_SESSION_HISTORY, V$SESSION_LONGOPS, and V$SESSION

Instead, you can use the SQL monitoring report to view SQL monitoring data.

The SQL monitoring report is also available in a GUI version through Enterprise Manager.

Diagnosing Parallel Execution

There are several examples to discover different methods to diagnose and troubleshoot parallel execution:

- Nonparallel executions, though parallel expected
  - Query (example 1)
  - PDML (example 2)
  - PDDL (example 3)
- Parallel execution with unexpected DOP (examples 4, 5)
- Parallel statement queuing: unexpected queued statements (example 6)

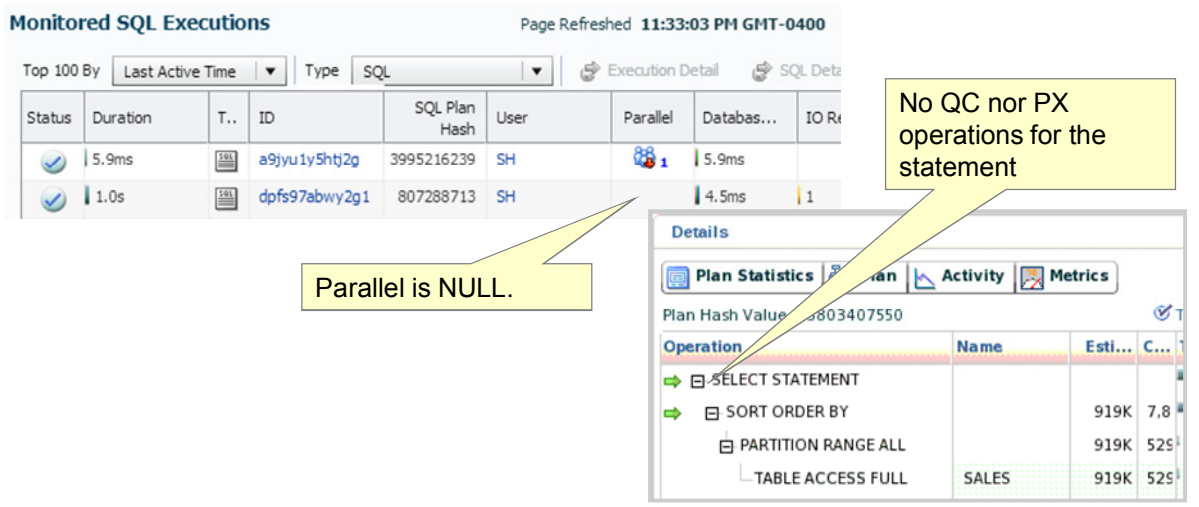
ORACLE

In the next few slides, several examples help you to discover different methods and tools for diagnosing and troubleshooting parallel executions.

Nonparallel Query: Example 1

```
SQL> select  /*+ monitor */  * from sh.sales order  by cust_id, prod_id;
```

- `PARALLEL_DEGREE_POLICY = AUTO`
- EM SQL Monitoring shows that no parallel processing occurs:



The slide example shows a query with the `MONITOR` hint. This hint allows a statement to be monitored in SQL Monitor Executions in Enterprise Manager even if the statement does not fulfill the requirements to be present in the list of Monitored SQL Executions.

You can access the SQL Monitoring feature in Enterprise Manager Cloud Control 12*c* by clicking the Performance tab. Scroll down and click **SQL Monitoring**.

Enterprise Manager Monitored SQL Executions shows the request that is running with a DOP of null, which means it executes serially.

Diagnosing with PLAN_TABLE

The EXPLAIN PLAN command line explains why parallel processing cannot be used:

```
SQL> EXPLAIN PLAN FOR
  2  select /*+ noparallel */ *  from sh.sales  order  by cust_id, prod_id;
SQL> select * from table(dbms_xplan.display);
PLAN_TABLE_OUTPUT
------------------------------------------------------------
| Id  | Operation           | Name  | Rows  | Bytes |TempSpc|
------------------------------------------------------------
|   0 | SELECT STATEMENT    |       |   918K|    25M|       |
|   1 |  SORT ORDER BY      |       |   918K|    25M|    42M|
|   2 |   PARTITION RANGE ALL|      |   918K|    25M|       |
|   3 |    TABLE ACCESS FULL | SALES |   918K|    25M|       |
------------------------------------------------------------
Note
-----
   - Degree of Parallelism is 1 because of hint
```

ORACLE

EXPLAIN PLAN Command

The EXPLAIN PLAN in the slide example shows that serial execution occurred instead of parallel execution due to a hint.

Troubleshooting with IO Statistics Collection

```
SQL> conn / as sysdba
Connected.
SQL>  SET SERVEROUTPUT ON
SQL> DECLARE lat INTEGER;
  2  iops INTEGER;
  3   mbps INTEGER;
  4  begin
  5  DBMS_RESOURCE_MANAGER.CALIBRATE_IO (4,10,iops, mbps, lat);
  6  DBMS_OUTPUT.PUT_LINE ('max_iops = ' || iops);
  7  DBMS_OUTPUT.PUT_LINE ('latency = ' || lat);
  8  dbms_output.put_line('max_mbps = '|| mbps);
  9  end;
 10  /
```

```
SQL> EXPLAIN PLAN FOR
  2  select /*+ monitor */ *  from sh.sales  order  by cust_id, prod_id;
Explained.
SQL> select * from table(dbms_xplan.display) ;
…
Note
-----
 - automatic DOP: Computed Degree of Parallelism is 1 because of
parallel threshold
```

EXPLAIN PLAN Command

The EXPLAIN PLAN in the slide example shows that Oracle managed to compute an automatic DOP, because the IO calibration has been executed. The IO statistics are present and helped Oracle compute Auto DOP.

The “Computed Degree of Parallelism is 1 because of parallel threshold” message provides the reason why the DOP computed is 1. The serial elapsed time is found to be smaller than the threshold determined by the PARALLEL_MIN_TIME_THRESHOLD parameter.

Diagnosing and Troubleshooting

PARALLEL_DEGREE_POLICY = AUTO
PARALLEL_MIN_TIME_THRESHOLD = AUTO

- Because the serial plan estimated elapsed time is smaller than the PARALLEL_MIN_TIME_THRESHOLD elapsed time (of 10 seconds), the serial execution is chosen as faster than the parallel execution.
- Decrease PARALLEL_MIN_TIME_THRESHOLD:

```
SQL> alter system set PARALLEL_MIN_TIME_THRESHOLD = 1;
```

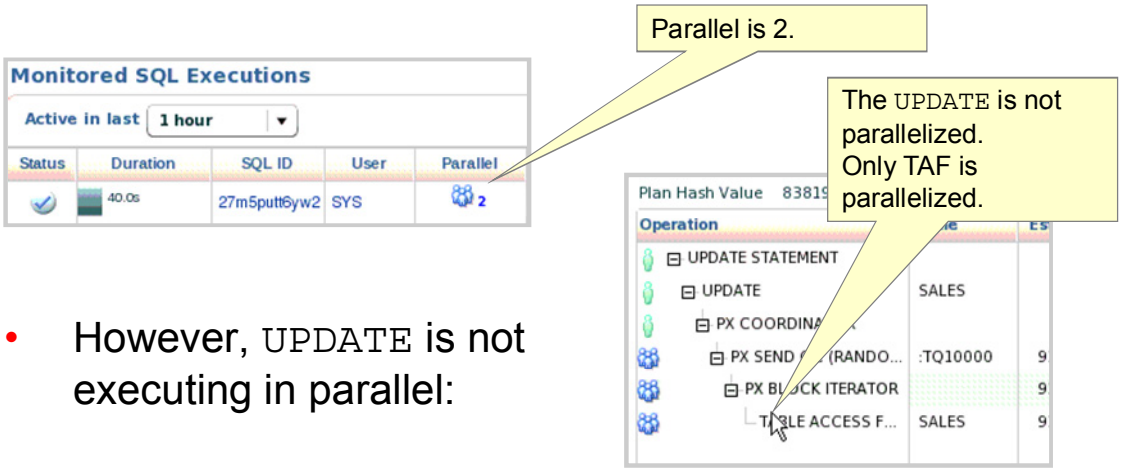
```
SQL> EXPLAIN PLAN FOR
  2  select /*+ monitor */ *  from sh.sales  order  by cust_id, prod_id;
Explained.
SQL> select * from table(dbms_xplan.display) ;
…
Note
-----
 - automatic DOP: Computed Degree of Parallelism is 2
```


The message is now “Computed Degree of Parallelism is 2.” The serial elapsed time is found to be greater than the threshold determined by the PARALLEL_MIN_TIME_THRESHOLD parameter because it was decreased from 10 seconds (the default value) to 1 second.

Nonparallel UPDATE: Example 2

```
UPDATE /*+ PARALLEL (sales,2) */ sales SET qty_s=qty_s+100;
```

- PARALLEL_DEGREE_POLICY = MANUAL
- EM SQL Monitoring **shows that parallel processing is used with DOP 2:**



- However, UPDATE is not executing in parallel:

The slide example shows an UPDATE with PARALLEL hint with a DOP of 2.

You can access the SQL Monitoring feature in Enterprise Manager Cloud Control 12*c* by clicking the Performance tab. Scroll down and click **SQL Monitoring**.

Enterprise Manager Monitored SQL Executions shows the statement that is running with a DOP of 2.

Clicking the Status icon of the statement shows the execution plan with the QC and one PX server set performing the Table Access Full of the SALES table. The statement shows the execution plan with the QC retrieving rows from the PX server set that performed the Table Access Full on the SALES table.

The UPDATE is performed after the QC retrieves all the rows to UPDATE.

Diagnosing with V$PQ_SESSTAT: Example 2

- Before UPDATE:

```
SQL> SELECT * FROM V$PQ_SESSTAT;

STATISTIC                      LAST_QUERY SESSION_TOTAL
------------------------------ ---------- -------------
Queries Parallelized                    0             0
DML Parallelized                        0             0
DDL Parallelized                        0             0
DFO Trees                               0             0
Server Threads                          0             0
Allocation Height                       0             0
Allocation Width                        0             0
```

- After UPDATE: No DML Parallelized; only query

```
STATISTIC                      LAST_QUERY SESSION_TOTAL
------------------------------ ---------- -------------
Queries Parallelized                    1             1
DML Parallelized                        0             0
DDL Parallelized                        0             0
```


These are the definitions of the columns and values:

STATISTIC: The following statistics (fixed rows) have been defined for this view. After you have run a query or DML operation, you can use the information to view the number of PX processes used. The names of the statistics can be:

- Queries Parallelized: Number of queries run in parallel
- DML Parallelized: Number of DML operations run in parallel
- DFO Trees: Number of executed DFO trees
- Server Threads: Total number of query servers used
- Allocation Height: DOP
- Allocation Width: Requested number of instances
- Local Msgs Sent: Number of local (intrainstance) messages sent
- Distr Msgs Sent: Number of remote (interinstance) messages sent
- Local Msgs Recv'd: Number of local (intrainstance) messages received
- Distr Msgs Recv'd: Number of remote (interinstance) messages received

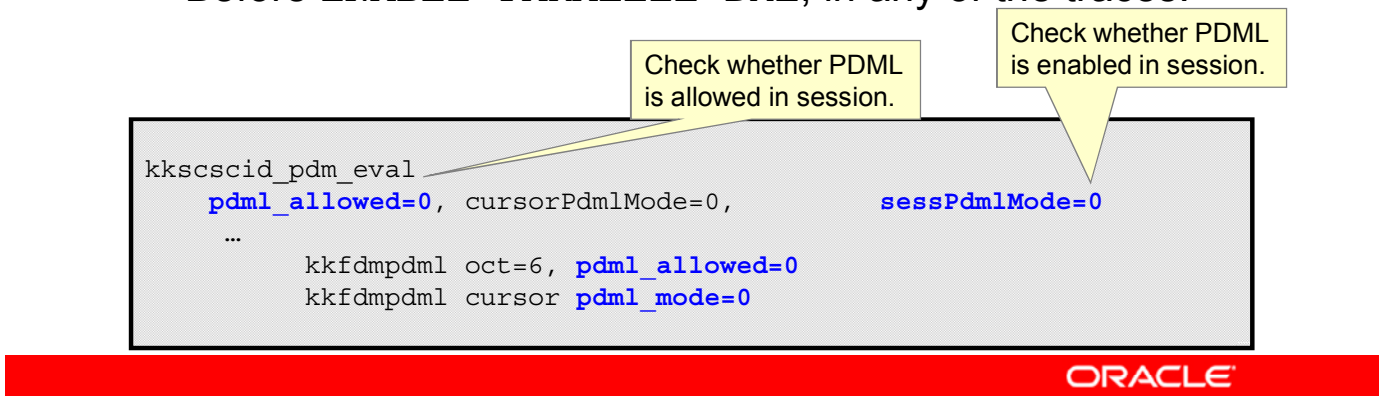
LAST_QUERY: The value of the statistic for the last operation

SESSION_TOTAL: The value of the statistic for the entire session to this point

Diagnosing with _PX_TRACE: Example 2

```
ALTER SESSION SET "_PX_TRACE"="compilation","execution","messaging";
UPDATE /*+ PARALLEL (sales,2) */ sales SET qty_s=qty_s+100;
```

- Check for trace files in the directory defined in DIAGNOSTIC_DEST.
- In the diag/rdbms/orcl/orcl/trace directory:
  - orcl_ora_5324.trc: Trace of QC
  - orcl_p000_1504.trc: Trace of PX server 1
  - orcl_p001_5116.trc: Trace of PX server 2
- Before ENABLE PARALLEL DML, in any of the traces:



You use the _PX_TRACE event to trace the session.

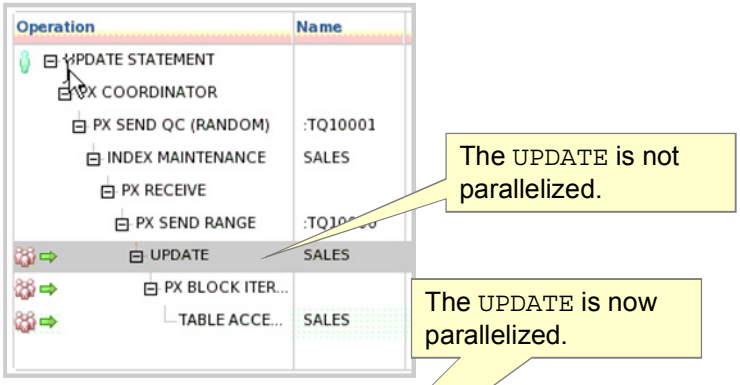
One trace file is created for the QC process and as many traces as PX server processes.

In the QC trace file, it is clearly stated that pdml_allowed is not enabled; the value is 0.

Diagnosing with SQL Monitoring and V$PQ_SESSTAT

```
SQL> ALTER SESSION ENABLE PARALLEL DML;
```

EM SQL Monitoring shows that parallel processing is used with DOP 2:




```
SQL> SELECT * FROM V$PQ_SESSTAT;
STATISTIC                      LAST_QUERY SESSION_TOTAL
------------------------------ ---------- -------------
Queries Parallelized                    0             1
DML Parallelized                        1             1
DDL Parallelized                        0             0
```

ORACLE

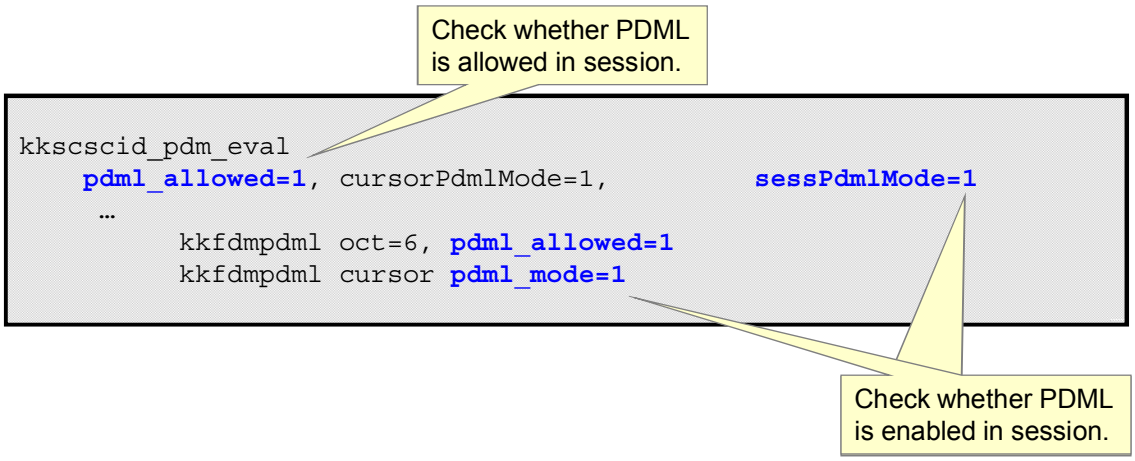
You enable the use of `PARALLEL DML` in the session.

In EM, the execution plan clearly shows that one PX server set was used and that the `UPDATE` was performed by the PX servers of the set.

The `V$PQ_SESSTAT` view clearly states that the last DML statement was parallelized.

Diagnosing with _PX_TRACE: Example 2

After ENABLE PARALLEL DML, in any of the PX servers traces:



After you enable the session to use PARALLEL DML, you use the _PX_TRACE event to trace the session.

In the QC trace file, it is clearly stated that pdml_allowed is now enabled; the value is 1.

Nonparallel ALTER INDEX: Example 3

1. The EXPLAIN PLAN command line shows that no parallel execution occurs:

```
SQL> ALTER SESSION ENABLE PARALLEL DDL;
SQL> ALTER INDEX sh.products_pk  REBUILD ;
SQL> EXPLAIN PLAN FOR  ALTER INDEX sh.products_pk  REBUILD ;
SQL> select * from table(dbms_xplan.display) ;
|   0 | ALTER INDEX STATEMENT |             |    72 |   288
|   1 |  INDEX BUILD UNIQUE   | PRODUCTS_PK |       |
|   2 |   SORT CREATE INDEX   |             |    72 |   288
|   3 |    TABLE ACCESS FULL  | PRODUCTS    |    72 |   288
```


In the slide example, though the session was enabled to use PARALLEL DDL in step 1, the EXPLAIN PLAN shows that the ALTER INDEX REBUILD was performed serially.

Nonparallel ALTER INDEX: Example 3

1. The EXPLAIN PLAN command line shows that no parallel execution occurs:

```
SQL> ALTER SESSION ENABLE PARALLEL DDL;
SQL> ALTER INDEX sh.products_pk  REBUILD ;
SQL> EXPLAIN PLAN FOR  ALTER INDEX sh.products_pk  REBUILD ;
SQL> select * from table(dbms_xplan.display) ;
|   0 | ALTER INDEX STATEMENT |             |    72 |   288
|   1 |  INDEX BUILD UNIQUE   | PRODUCTS_PK |       |
|   2 |   SORT CREATE INDEX   |             |    72 |   288
|   3 |    TABLE ACCESS FULL  | PRODUCTS    |    72 |   288
```

2.

```
SQL> ALTER SESSION FORCE PARALLEL DDL;
SQL> ALTER INDEX sh.products_pk  REBUILD ;
```


In step 2 of the slide example, the session is forced to use PARALLEL DDL.

Nonparallel ALTER INDEX: Example 3

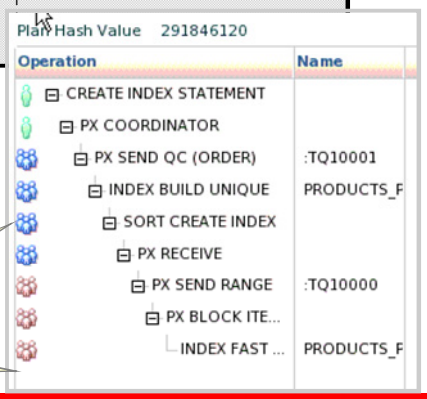
1. EXPLAIN PLAN command line shows no parallel execution occurs:

```
SQL> ALTER SESSION ENABLE PARALLEL DDL;
SQL> ALTER INDEX sh.products_pk  REBUILD ;
SQL> EXPLAIN PLAN FOR  ALTER INDEX sh.products_pk  REBUILD ;
SQL> select * from table(dbms_xplan.display) ;
|   0 | ALTER INDEX STATEMENT |             |    72 |   288
|   1 |  INDEX BUILD UNIQUE   | PRODUCTS_PK |       |
|   2 |   SORT CREATE INDEX   |             |    72 |   288
|   3 |    TABLE ACCESS FULL  | PRODUCTS    |    72
```

2. Enterprise Manager SQL Monitoring shows that parallel processing is used with DOP 4:

```
SQL> ALTER SESSION FORCE PARALLEL DDL;
SQL> ALTER INDEX sh.products_pk  REBUILD ;
```

The ALTER INDEX is now parallelized.



ORACLE

In the slide example, EM clearly shows that two PX sets were used to execute the request and that the operations SORT CREATE INDEX and INDEX BUILD UNIQUE were performed by the PX servers of set 2 (the consumers), whereas the PX servers of set 1 read the rows from the source index (the producers).

Diagnosing with V$PQ_SESSTAT: Example 3

- Before ALTER INDEX REBUILD:

```
SQL> SELECT * FROM V$PQ_SESSTAT;

STATISTIC                      LAST_QUERY SESSION_TOTAL
------------------------------ ---------- -------------
Queries Parallelized                    0             0
DML Parallelized                        0             0
DDL Parallelized                        0             0
```

- After ALTER INDEX REBUILD:

```
STATISTIC                      LAST_QUERY SESSION_TOTAL
------------------------------ ---------- -------------
Queries Parallelized                    0             0
DML Parallelized                        0             0
DDL Parallelized                        1             1
```


These are the definitions of the columns and values:

STATISTIC: The following statistics (fixed rows) have been defined for this view. After you have run a query or DML operation, you can use the information to view the number of PX processes used. The names of the statistics can be:

- Queries Parallelized: Number of queries run in parallel
- DML Parallelized: Number of DML operations run in parallel
- DFO Trees: Number of executed DFO trees
- Server Threads: Total number of cluster databases used
- Allocation Height: Requested number of servers per instance
- Allocation Width: Requested number of instances
- Local Msgs Sent: Number of local (intrainstance) messages sent
- Distr Msgs Sent: Number of remote (interinstance) messages sent
- Local Msgs Recv'd: Number of local (intrainstance) messages received
- Distr Msgs Recv'd: Number of remote (interinstance) messages received
- LAST_QUERY: The value of the statistic for the last operation
- SESSION_TOTAL: The value of the statistic for the entire session to this point

Diagnosing with _PX_TRACE: Example 3

- Before `FORCE PARALLEL DDL`, no PX server trace, no parallelism
- After `FORCE PARALLEL DDL`, in PX server traces:

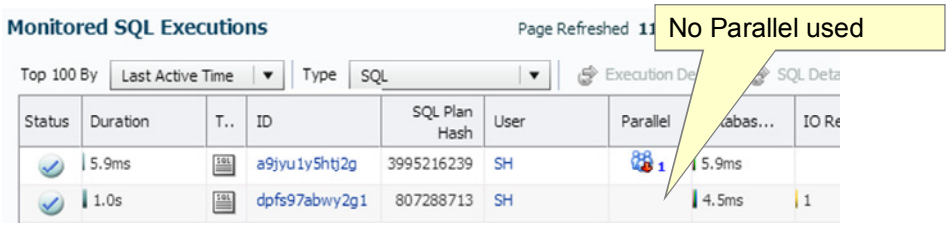
```
parallel_ddl_forced_degree          = default
parallel_ddl_forced_instances       = default
parallel_ddl_mode                   = forced
```


After you forced the session to use `PARALLEL DDL`, the `_PX_TRACE QC` trace file clearly states that `parallel_ddl_mode` was forced.

Auto DOP with Unexpected DOP: Example 4

```
SQL> SELECT /*+ MONITOR */ * FROM sh.sales
  2  ORDER BY prod_id ;
```

- EM SQL Monitoring shows that no parallel processing occurs:



The slide example shows a query with the `MONITOR` hint. This hint allows a statement to be monitored in SQL Monitor Executions in Enterprise Manager even if the statement does not fulfill the requirements to be present in the list of Monitored SQL Executions.

You can access the SQL Monitoring feature in Enterprise Manager Cloud Control 12*c* by clicking the Performance tab. Scroll down and click **SQL Monitoring**.

Enterprise Manager Monitored SQL Executions shows the request that is running with a DOP of null, which means it executes serially.

Diagnosing with PLAN_TABLE

The EXPLAIN PLAN command line explains why parallel processing is not used:

```
SQL> EXPLAIN PLAN FOR
  2  SELECT /*+ MONITOR */ * FROM sales ORDER BY 1,2 ;

SQL> select * from table(dbms_xplan.display) ;
------------------------------------------------------------
PLAN_TABLE_OUTPUT
------------------------------------------------------------
| Id  | Operation           | Name  | Rows  | Bytes |TempSpc
------------------------------------------------------------
|   0 | SELECT STATEMENT    |       |    72 |   576 |  1955
|   1 |  SORT ORDER BY      |       |    72 |   576 |  1955
|   2 |   PARTITION RANGE ALL|       |   918K|  7178K|  1920
|   3 |    TABLE ACCESS FULL | SALES |   918K|  7178K|  1920
------------------------------------------------------------
PLAN_TABLE_OUTPUT
------------------------------------------------------------
```

No Note at the bottom page: no DOP computation

EXPLAIN PLAN Command

The EXPLAIN PLAN in the slide example shows that Oracle did not try to compute any DOP. There are no messages and no notes after the PLAN_TABLE_OUTPUT.

Diagnosing and Troubleshooting

PARALLEL_DEGREE_POLICY = LIMITED

```
SQL> select DEGREE from dba_tables where table_name='SALES';
```

- Because the PARALLEL_DEGREE_POLICY parameter is set to LIMITED and the SH.SALES DEGREE value is not defined, serial execution is chosen.
- Change PARALLEL_DEGREE_POLICY to AUTO:

```
SQL> alter system set PARALLEL_DEGREE_POLICY = AUTO;
```

```
SQL> EXPLAIN PLAN FOR
  2  select /*+ monitor */ *  from sh.sales  order  by cust_id, prod_id;
Explained.
SQL> select * from table(dbms_xplan.display) ;
……
Note
-----
 - automatic DOP: Computed Degree of Parallelism is 2
```

ORACLE

Auto DOP with Unexpected DOP: Example 4

Examine the PARALLEL_DEGREE_POLICY parameter. You will discover that it is set to LIMITED.

In this case, Oracle uses the DEGREE set at the object level, in this case of SALES table. Because the SALES table does not have any DEGREE set, Oracle runs the statement in serial.

In the slide example, the PARALLEL_DEGREE_POLICY parameter is then set to AUTO.

A new EXPLAIN PLAN shows that Oracle now computes the DOP.

Manual DOP with Unexpected DOP: Example 5

- Three sessions are launched:

```
SQL> SELECT /*+ MONITOR */ * from sh.sales ORDER BY 1,2 ;
```

- EM SQL Monitoring shows that parallel processing for a statement is downgraded to 1 instead of 8:

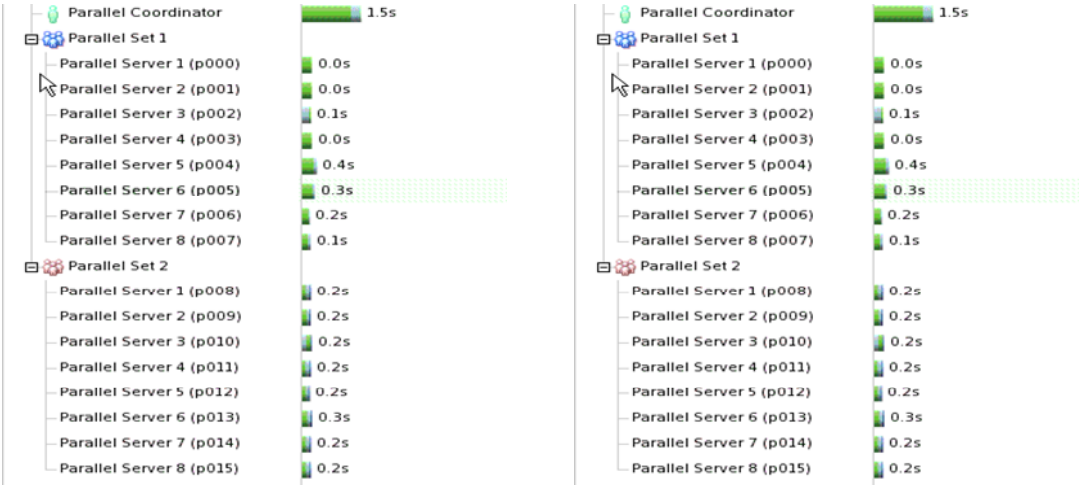
The slide example shows a query with the `MONITOR` hint. This hint allows a statement to be monitored in SQL Monitor Executions in Enterprise Manager even if the statement does not fulfill the requirements to be present in the list of Monitored SQL Executions.

You can access the SQL Monitoring feature in Enterprise Manager Cloud Control 12*c* by clicking the Performance tab. Scroll down and click **SQL Monitoring**.

Enterprise Manager Monitored SQL Executions shows the request that is running with a DOP of 1 instead of 8, which means it executes serially.

Manual DOP with Unexpected DOP: Example 5

- `PARALLEL_MAX_SERVERS = 32`
- EM SQL Monitoring shows the following if you expand:



  - Two sessions with DOP 8: 2 * DOP 8 * 2 (PX sets) = 32.
  - The third session does not run in parallel because no PX server is available.

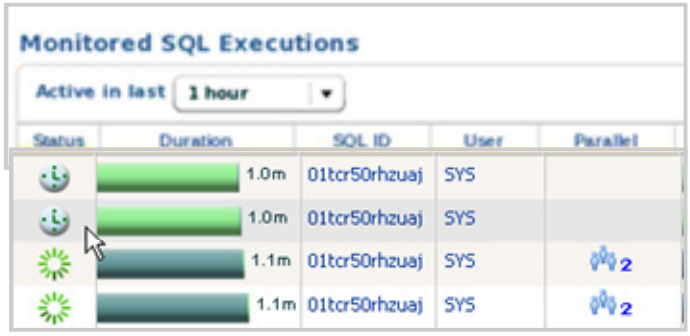
Examine the `PARALLEL_MAX_SERVERS` parameter. You will discover that it is set to `32`.

In EM, the SQL Monitoring shows two statements running with a DOP of 8 each. Each request uses two PX sets and, therefore, 16 PX servers each. Both running statements use 32 PX servers.

The maximum number of PX servers available for the instance is 32. As all of the resources are currently used, the next statement requiring PX servers cannot obtain any and, therefore, runs in serial.

Auto DOP with Unexpected Queued Statements: Example 6

- Four sessions are launched with Max DOP 2.
- Only two sessions execute, two others queue. Why?



- Diagnose with SQL Monitoring to count the PX used:
  - DOP 2 * 2 (PX sets) = 4 for each statement: Total 8.
  - The third statement requests DOP 2, requiring 4 PX servers.
  - `PARALLEL_SERVERS_TARGET=10`
  - The third statement waits for 4 available PX servers.

ORACLE

Examine the `PARALLEL_SERVERS_TARGET` parameter. You will discover that it is set to `10`.

In EM, the SQL Monitoring shows two statements running with a DOP of 2 each. Each request uses 2 PX sets and, therefore, 4 PX servers each. Both running statements use 8 PX servers.

The maximum number of PX servers usable for the instance is 10. Because there are only 2 PX servers free (of the usable resources), the next statement requiring a DOP of 2 with 2 PX sets cannot obtain the required PX servers and, therefore, is queued until 2 PX servers are free.

Diagnosing with _PX_TRACE: Example 6

```
ALTER SESSION SET "_PX_TRACE"=low,messaging;
```

- In QC trace file of session 1:

```
GROUP GET
        Acquired 4 slaves on 1 instances avg height=2 in 2 sets q serial:65025
        P000 inst 1 spid 6108
        P001 inst 1 spid 2536
        P002 inst 1 spid 5768
        P003 inst 1 spid 820
        Insts    1
        Svrs     4
```

DOP 2

2 PX sets with DOP 2: 4 PX servers

The statement gets 4 PX servers.

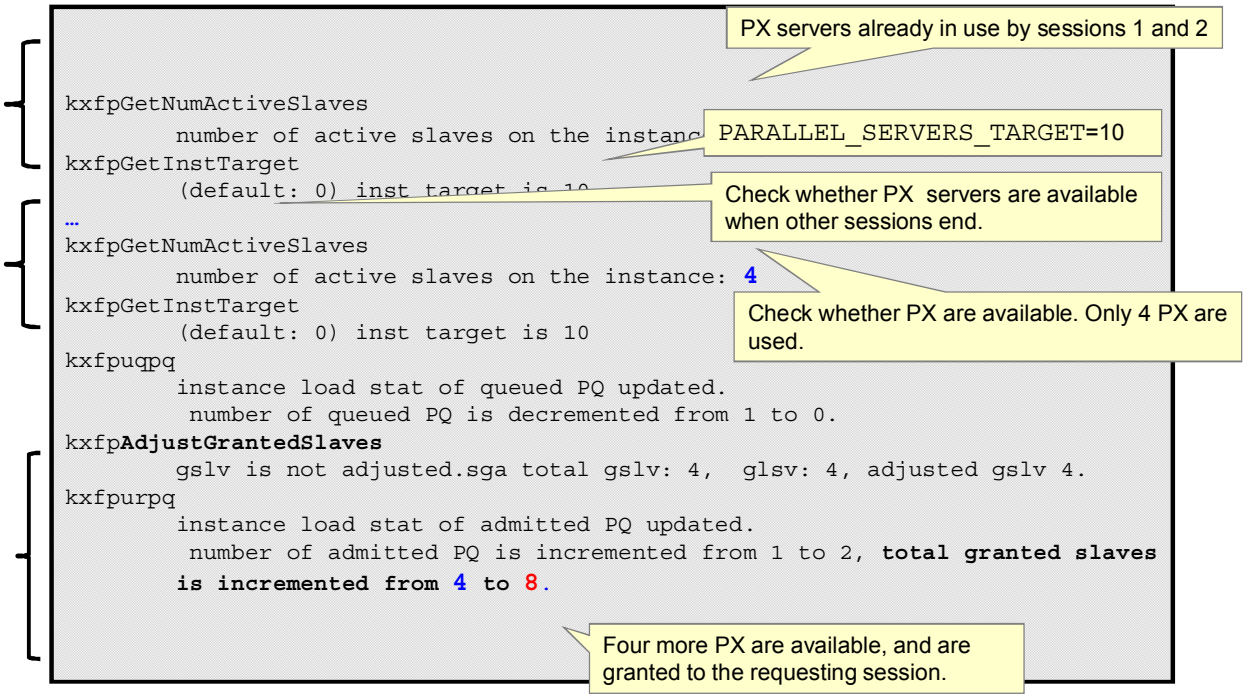
- In QC trace file of session 2:

```
GROUP GET
        Acquired 4 slaves on 1 instances avg height=2 in 2 sets q serial:65537
        P004 inst 1 spid 1744
        P005 inst 1 spid 4008
        P006 inst 1 spid 4516
        P007 inst 1 spid 532
        Insts    1
        Svrs     4
```


The _PX_TRACE QC trace file of the first running statement shows that it could obtain the 4 PX servers it requested. The _PX_TRACE QC trace file of the second running statement shows that it could obtain the 4 PX servers it requested.

Diagnosing with _PX_TRACE: Example 6

In QC trace file of session 3:



The _PX_TRACE QC trace file of the third statement shows that it could not obtain the 4 PX servers that it requested until some of them were freed as one of the two other running statements completed.

Auto DOP with Unexpected Queued Statements

- Increase the `PARALLEL_SERVERS_TARGET`.
  - DOP 2 * 2 (PX sets) = 4 for each statement =Total 8
  - The third statement requests DOP 2 (4 PX servers).
  - `PARALLEL_SERVERS_TARGET=12`
  - The third statement executes with 4 available PX servers.
- In this particular case, decreasing the `PARALLEL_DEGREE_LIMIT=1` means no parallelism.

The solutions can be any of the following:

- Increase the `PARALLEL_SERVERS_TARGET`
- Decrease the `PARALLEL_DEGREE_LIMIT`, but in this particular case, it would result in setting `PARALLEL_DEGREE_LIMIT` to 1. This means that the statements run in serial.

Trace with _PX_TRACE

- Introduced in Oracle Database 9*i*

```
ALTER SESSION SET "_PX_TRACE"="compilation","execution","messaging";
```

- Area:
  - Compilation
  - Execution
  - Messaging
  - Buffer
  - Scheduling
  - Granule
  - Low: No PX trace, only QC trace
  - None: Stop tracing
- For more information, refer to "Tracing Parallel Execution with _px_trace. Part I (Document 444164.1)."

Syntax

```
"_px_trace" = <verbosity>,<area>,,,, <verbosity>,<area>, <time>
```

Up to three components can be specified when you start the trace:

- Area in which tracing is required
- Verbosity
- Timing information

The possible values for area are the following:

- `scheduling` (equivalent to some of event 10384 and some of 10390)
- `execution` (equivalent to some of event 10390)
- `granule` (equivalent to some of event 10390 and some of 10391)
- `messaging` (equivalent to event 10392 and event 10393)
- `buffer` (equivalent to event 10399)
- `compilation` (no equivalent event)
- `all` (all of the above)
- `none` (none of the above)

Different Parallel SQL Classes

- A SQL command runs in serial.
- Every row source of the SQL runs in parallel.
  - The PX server and QC are spawned on only one node in a cluster.
  - The PX server and QC are spawned on more than one node.
- SQL is run in mixed mode: Some steps run in parallel and some in serial.

If the parallel statements are slow, what can be done to tune them?

- Use keyword- and parameter-based `_PX_TRACE`, rather than event-based tracing.

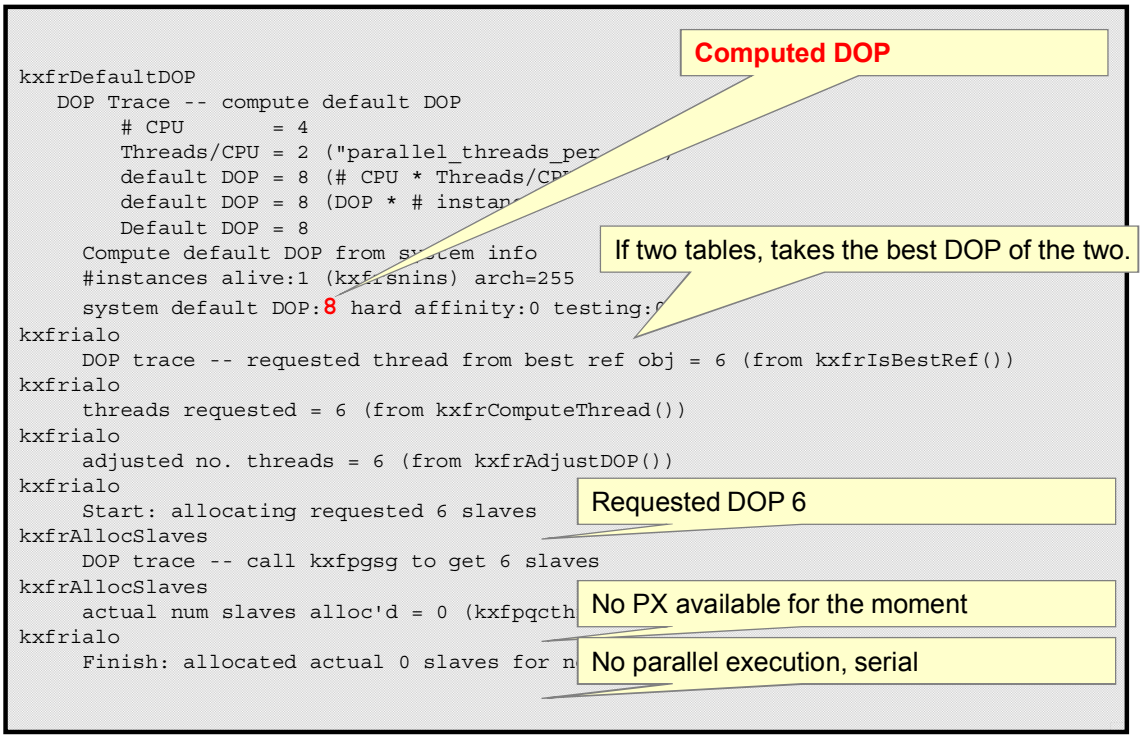
ORACLE

SQL Classes and Tuning

To tune the different types of parallel statements, use the `_PX_TRACE` event to obtain detailed information about PX server usage.

Tracing of Degree of Parallelism

In the QC trace with “compilation” keyword:



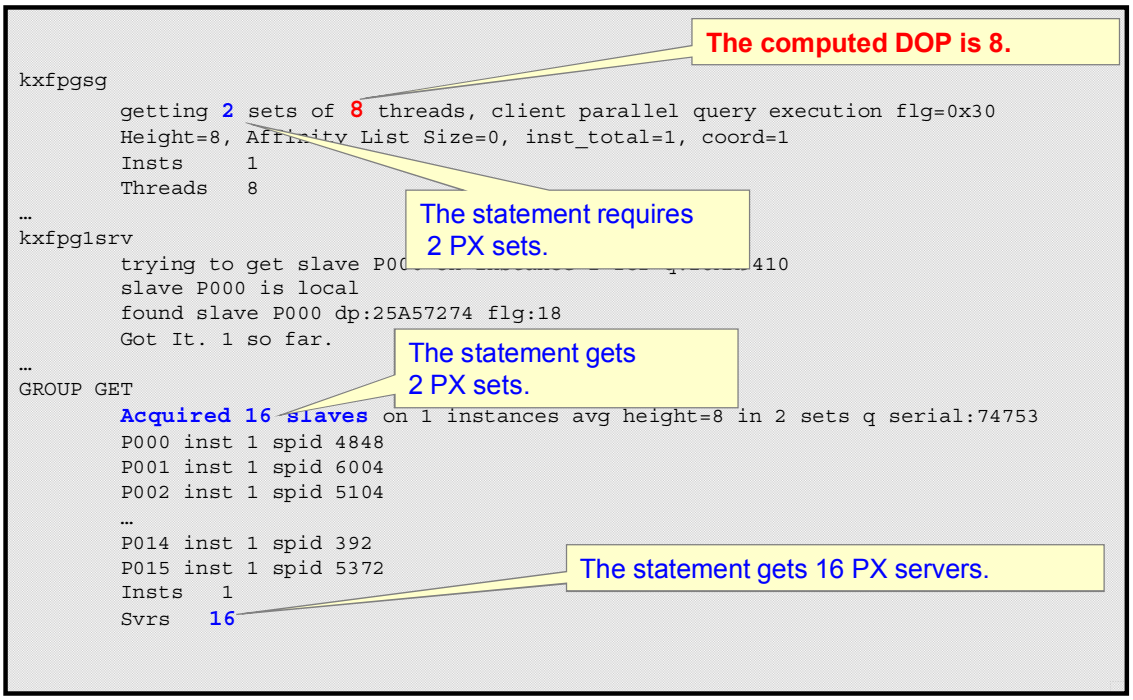
The slide example shows the various information captured during the statement tracing.

This information is collected in the QC trace file.

Other types of information are collected in the PX server processes trace files, such as block–range granules.

Tracing of Degree of Parallelism

In the QC trace with “compilation” keyword:



The slide example shows the various bits of information captured during the statement tracing.

This information is collected in the QC trace file.

Files to Send to Oracle Support

If you send traces files to Oracle Support:

- Identify all files related to parallel statement:

```
SQL> ALTER SESSION SET tracefile_identifier='10046';
SQL> ALTER SESSION SET timed_statistics = true;
SQL> ALTER SESSION SET statistics_level=all;
SQL> ALTER SESSION SET "_px_trace" = low , messaging;
SQL> ALTER SESSION SET max_dump_file_size = unlimited;
SQL> ALTER SESSION SET events '10046 trace name context
                              forever,level 12';
-- Execute the queries or operations to be traced here --
SQL> ALTER SESSION SET events '10046 trace name context off';
SQL> ALTER SESSION SET "_px_trace" = none;
```

  - All related trace files of PX server names: `orcl_p00n_xxxxx_10046.trc`
  - The QC trace file name: `orcl_ora_xxxxx_10046.trc`
- For more information, refer to Master Note: “How to Get a 10046 Trace for a Parallel Query (Doc ID 1102801.1)”

ORACLE

The slide example shows how to set an identifier on your session or statement so that you can distinguish all the files related to one statement from the files of another statement.

Additional Diagnostic Views: V$PX_PROCESS

The V$PX_PROCESS view contains information about the sessions running parallel execution.

SERVER_NAME	STATUS	PID	SPID	SID	SERIAL#	IS_GV	CON_ID
P000	IN USE	30	2332	41	19325	FALSE	0
P006	IN USE	48	10982	71	35966	FALSE	0
P003	IN USE	33	2338	73	24949	FALSE	0
P001	IN USE	31	2334	83	37012	FALSE	0
P007	IN USE	49	10984	89	38550	FALSE	0
P005	IN USE	46	10978	95	38723	FALSE	0
P004	IN USE	45	10976	100	40829	FALSE	0
P002	IN USE	32	2336	117	2219	FALSE	0
P00D	AVAILABLE	63	12900			FALSE	0
P009	AVAILABLE	54	12885			FALSE	0
P00A	AVAILABLE	56	12887			FALSE	0
P00C	AVAILABLE	62	12898			FALSE	0
P00B	AVAILABLE	59	12889			FALSE	0
P00E	AVAILABLE	65	12902			FALSE	0
P008	AVAILABLE	51	12883			FALSE	0
P00F	AVAILABLE	66	12904			FALSE	0

16 rows selected

The V$PX_PROCESS view contains information about the sessions running parallel execution. The operating system process ID column (SPID) can be used to collaborate Oracle Database dictionary information with diagnostic tools from the operating system. The operating system process can be identified in Linux as follows (Using a SPID identified on the slide):

```
[oracle@host01 Desktop]$ ps -ef | grep 10982
oracle   10982     1  0 22:54 ?  00:00:00 ora_p006_orcl
```

Additional Diagnostic Views: V$PQ_SYSSTAT

The V$PQ_SYSSTAT view lists system statistics for parallel queries.

STATISTIC	VALUE	CON_ID
Servers Busy	0	0
Servers Idle	28	0
Servers Highwater	32	0
Server Sessions	8414	0
Servers Started	345	0
Servers Shutdown	317	0
Servers Cleaned Up	0	0
Queries Queued	0	0
Queries Initiated	1812	0
Queries Initiated (IPQ)	0	0
DML Initiated	3	0
DML Initiated (IPQ)	0	0
DDL Initiated	3	0
DDL Initiated (IPQ)	0	0
DFO Trees	2007	0
Sessions Active	0	0
Local Msgs Sent	157926	0
Distr Msgs Sent	0	0
Local Msgs Recv'd	158214	0
Distr Msgs Recv'd	0	0

20 rows selected

The V$PQ_SYSSTAT view lists system statistics for parallel queries. After you have run a query or DML operation, you can use the information derived from V$PQ_SYSSTAT to view the number of slave processes used, and other information for the system.

Additional Diagnostic Views: V$PQ_SLAVE

The V$PQ_SLAVE view lists statistics for each of the active parallel execution servers on an instance.

SLAVE_NAME	STATUS	SESSIONS	MSGS_SENT_CUR	MSGS_RCVD_CUR	IDLE_TIME_TOTAL	BUSY_TIME_TOTAL	CPU_SECS_TOTAL	MSGS_SENT_TOTAL	MSGS_RCVD_TOTAL
P000	BUSY	1601	5	6	401	53	23	14770	13741
P001	BUSY	1599	5	6	401	53	17	14769	13859
P002	BUSY	1627	5	6	438	16	24	17695	11521
P003	BUSY	1346	5	6	439	16	20	12790	10544
P004	BUSY	554	22	19	7	0	0	7456	5838
P005	BUSY	490	21	18	7	0	0	6968	5491
P006	BUSY	241	15	12	7	0	0	3507	2875
P007	BUSY	210	17	14	7	0	0	3174	2676
P008	IDLE	151	0	0	0	0	0	2402	2078
P009	IDLE	146	0	0	0	0	0	2621	2339
P00A	IDLE	135	0	0	0	0	0	2415	2023
P00B	IDLE	124	0	0	0	0	0	2213	1806
P00C	IDLE	70	0	0	0	0	0	1472	1279
P00D	IDLE	33	0	0	0	0	0	902	903
P00E	IDLE	28	0	0	0	0	0	712	640
P00F	IDLE	30	0	0	0	0	0	673	566

16 rows selected

Note: Not all columns are displayed.

The V$PQ_SLAVE view lists statistics for each of the active parallel execution servers on an instance.

Note: In the slide above, the following columns all had zero values and are not displayed: IDLE_TIME_CUR, BUSY_TIME_CUR, and CPU_SECS_CUR.

Summary

In this lesson, you should have learned how to:

- Use diagnostic tools to troubleshoot parallel execution
- Use tracing to generate trace files to send to Oracle Support