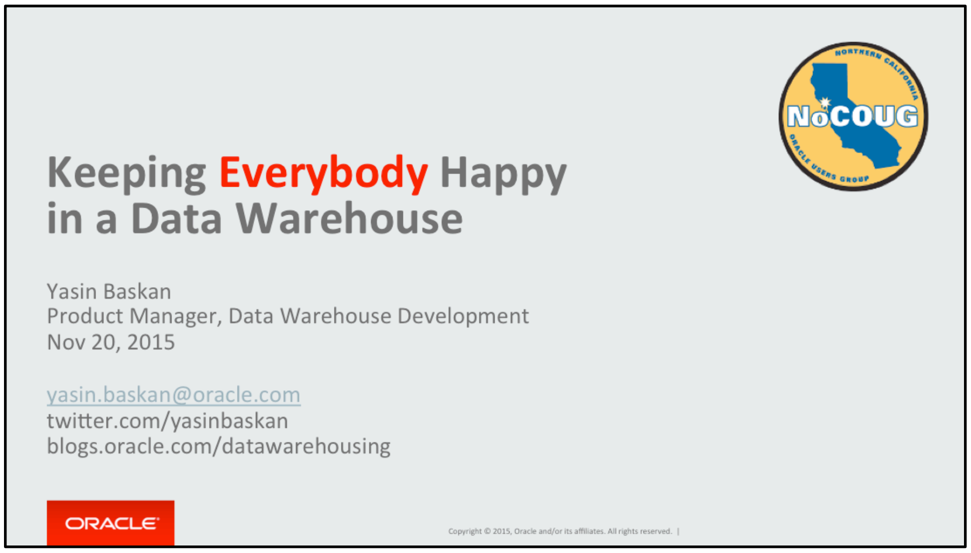
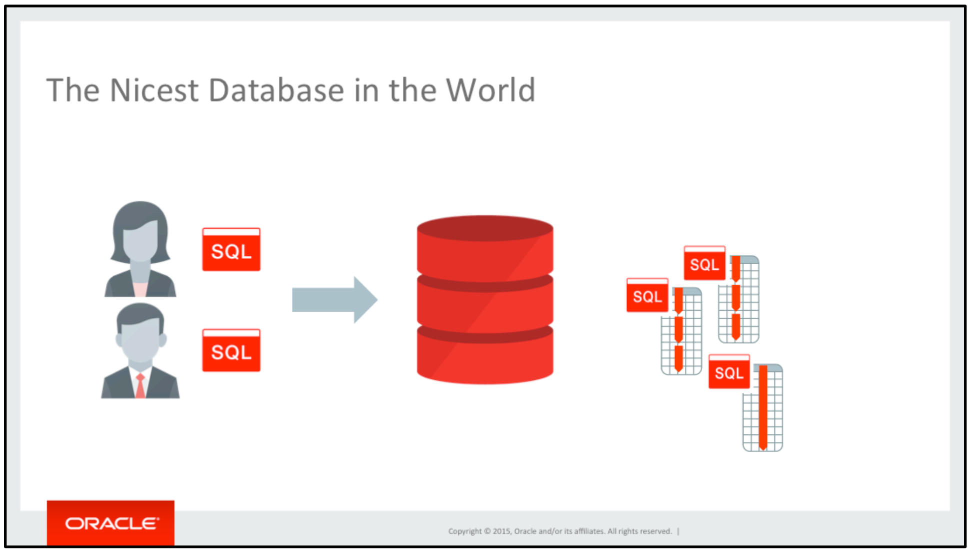


Keeping Everybody Happy in a Data Warehouse

Yasin Baskan
Product Manager, Data Warehouse Development
Nov 20, 2015

yasin.baskan@oracle.com
twitter.com/yasinbaskan
blogs.oracle.com/datawarehousing

This session is about keeping ALL users happy in a DW. The question is; is it possible to make everyone happy? We will find out in this presentation.

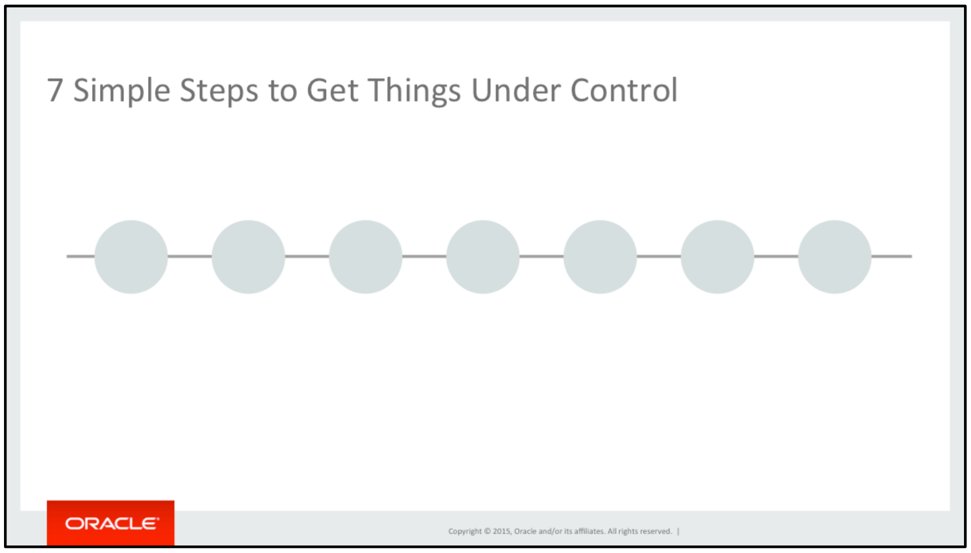


The Nicest Database in the World

SQL
SQL
SQL
SQL
SQL
ORACLE

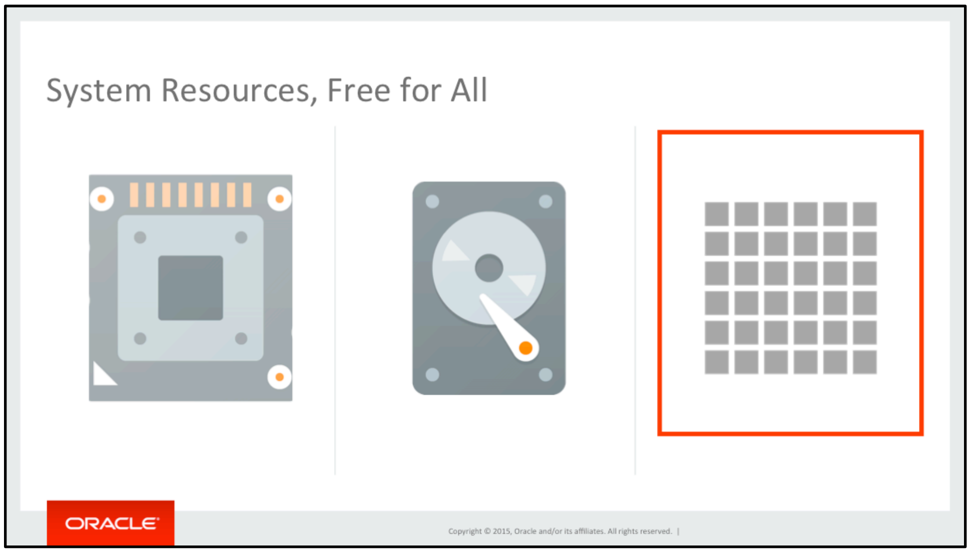
The Oracle Database is the nicest database in the world. It treats every user equally. It is like a fantasy land, as a user you can whatever you want. This may be OK in an OLTP system where every user accesses the database through an application and can only perform what the application allows. But in a DW there are lots of different kinds if users like power users, BI users, etc…

This means any user can step on other users' feet and make the others unhappy.



7 Simple Steps to Get Things Under Control

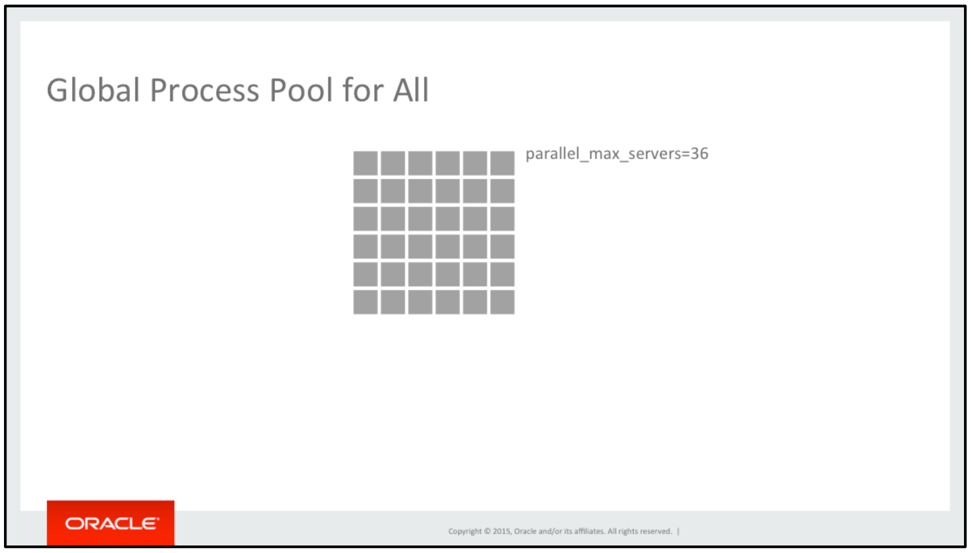
In this session we will talk about seven simple steps to get things under control.



System Resources, Free for All

First, let's look at how system resources are used by users. Any resources in the system is available to any user for free. Any user can use CPU, IO, network, memory, etc… The most important resource in a DW is the server process, you cannot use other resources like CPU and IO without getting processes allocated for you. That is why we will focus on not every type of resource but only processes.

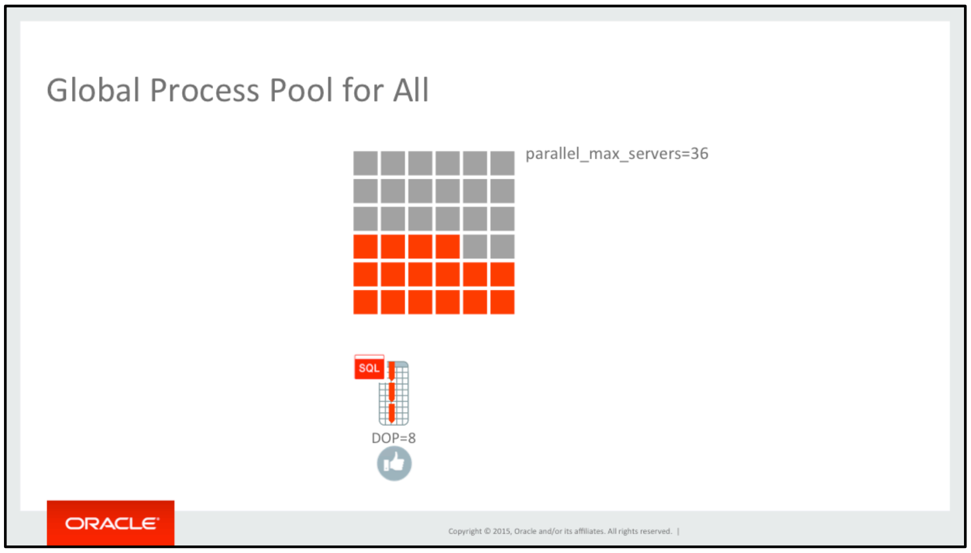
Since we are talking about DW and large data sets it is essential that multiple processes are used to execute SQL statements, that is why this session is about parallel execution processes (PX servers) and not about serial execution where you have only one process running the SQL statement.



Global Process Pool for All

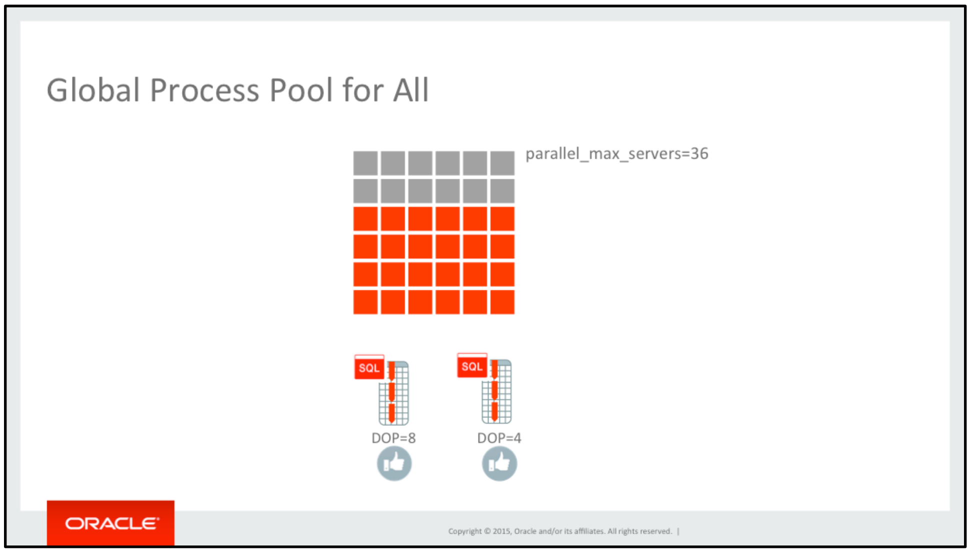
parallel_max_servers=36

Let's look at how PX servers are allocated to SQL statements in a default database configuration. The database is nice to its users but it also protects itself from over-use with parameters like processes, cpu_count, etc… The parameter it uses to limit the number of PX servers in the system is parallel_max_servers. The database keeps a pool of processes named "PX server pool", the number of processes in this pool is determined by parallel_max_servers.



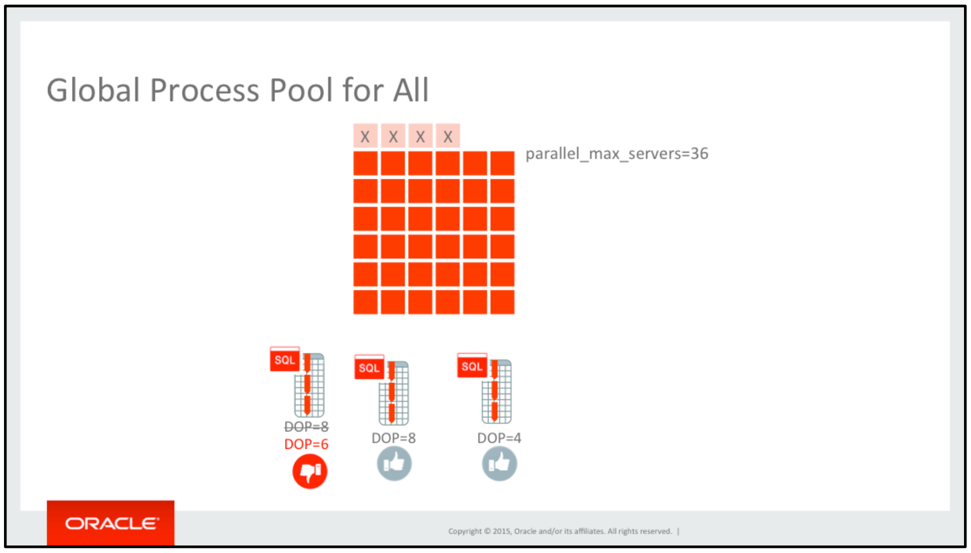
Global Process Pool for All

This example shows how processes from this pool are allocated to SQL statements. We have a statement asking for a Degree of Parallelism (DOP) of 8. We see that 16 processes are allocated from the pool to this statement. 2 is the magic number here. Most of the statements will use 2*DOP number of processes because of the producer/consumer model used in parallel execution. We use 2 sets of PX servers to carry out the SQL statement, each set uses PX servers equal to the DOP. There are exceptions to this which will be covered later in this presentation.



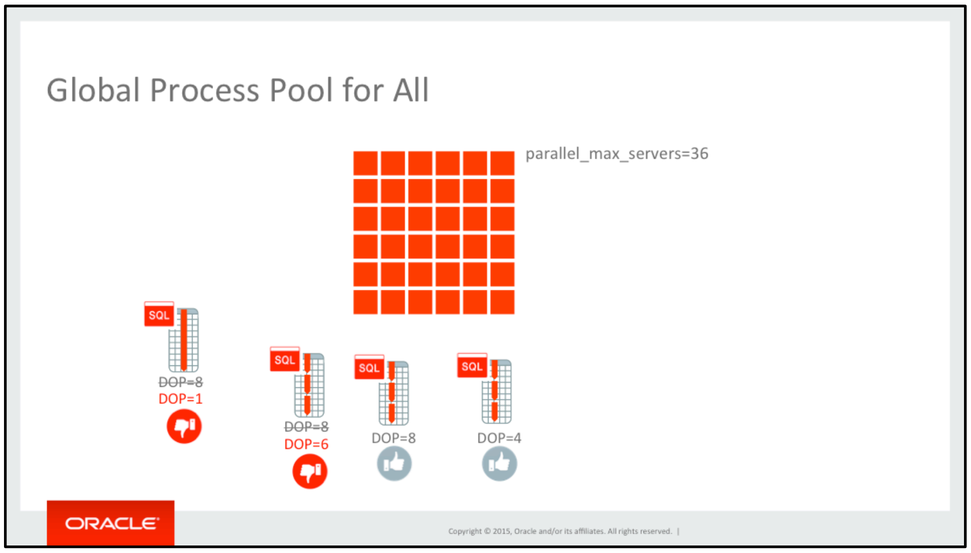
Global Process Pool for All

What happens when another statement is submitted at this point? In this example the second statement is asking for a DOP of 4 which means 8 processes. The database looks at the number of currently allocated processes and the number of available processes in the pool. Since we have enough available processes this statement gets allocated 8.



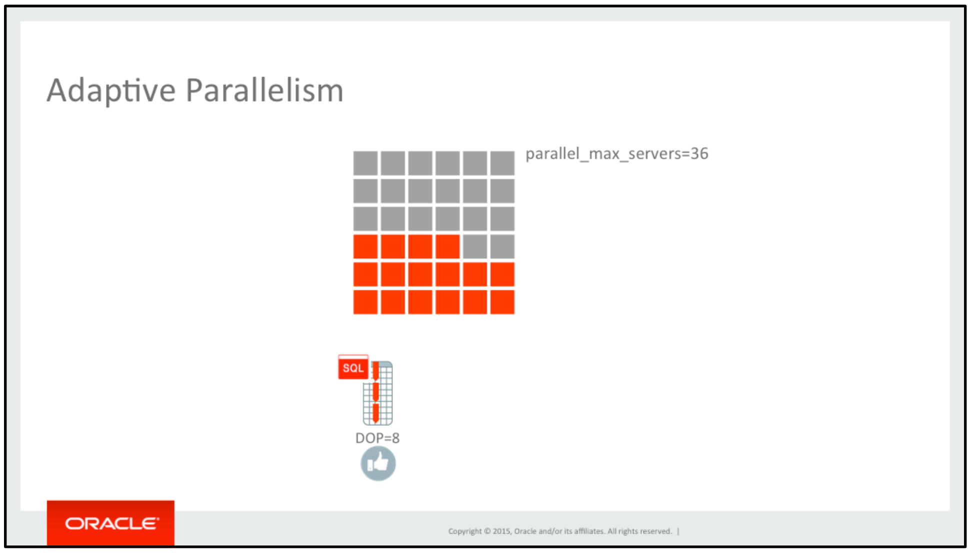
Global Process Pool for All

Now we have a third statement coming into the picture while the previous ones are still running. This statement asks for a DOP of 8 which means 16 processes. But we only have 12 processes left in the pool. In this case this statement will be allocated those 12 processes instead of 16. This is called a DOP downgrade, this statement gets downgraded from DOP=8 to DOP=6. A SQL statement cannot change its DOP after it has started, so in this case even if other queries finish and some processes become available this statement will run with DOP=6 to the end. This means this user will get less performance than he/she needed. Even worse the performance will be unpredictable, based on the time the statement runs it may or may not be downgraded based on the number of available processes.



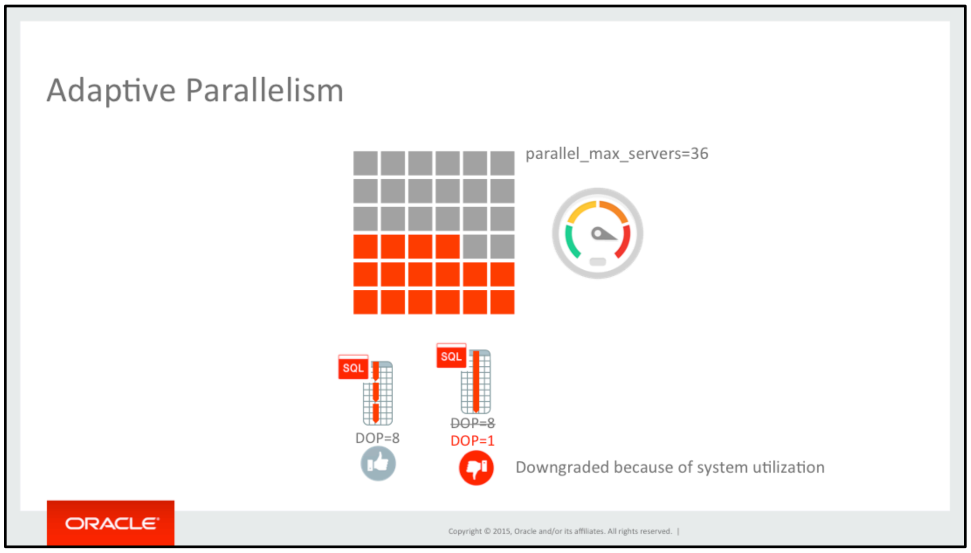
Global Process Pool for All

Here we see another statement submitted with a DOP of 8. Since all processes in the pool are in-use this statement does not get any processes allocated and runs with DOP=1 which means it runs serially. This user will see even worse performance because the downgrade is more dramatical than the downgrade from DOP=8 to DOP=6.



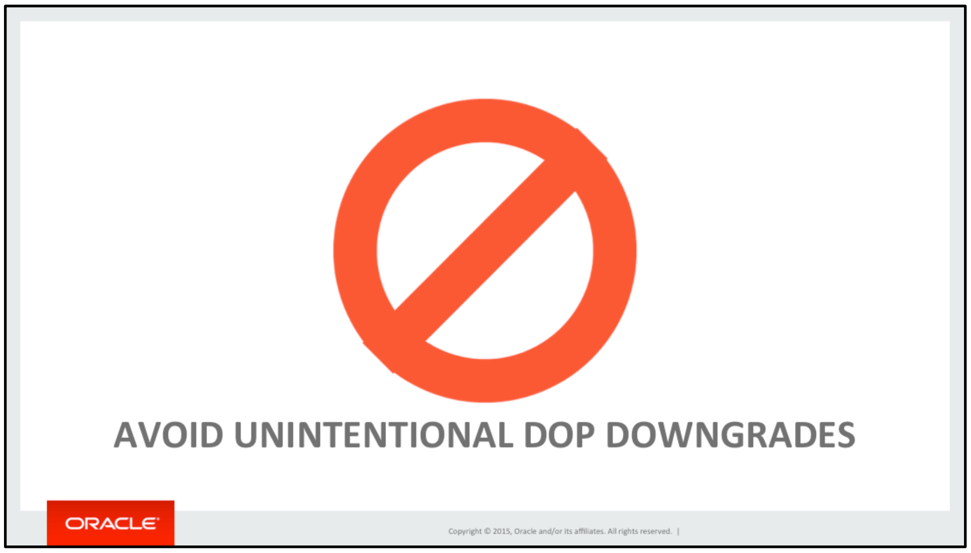
Adaptive Parallelism

Process shortage is one of the reasons of DOP downgrades. Another reason is the adaptive parallelism feature. Here we see a statement running with a DOP of 8 and using 16 processes from the PX server pool.



Adaptive Parallelism

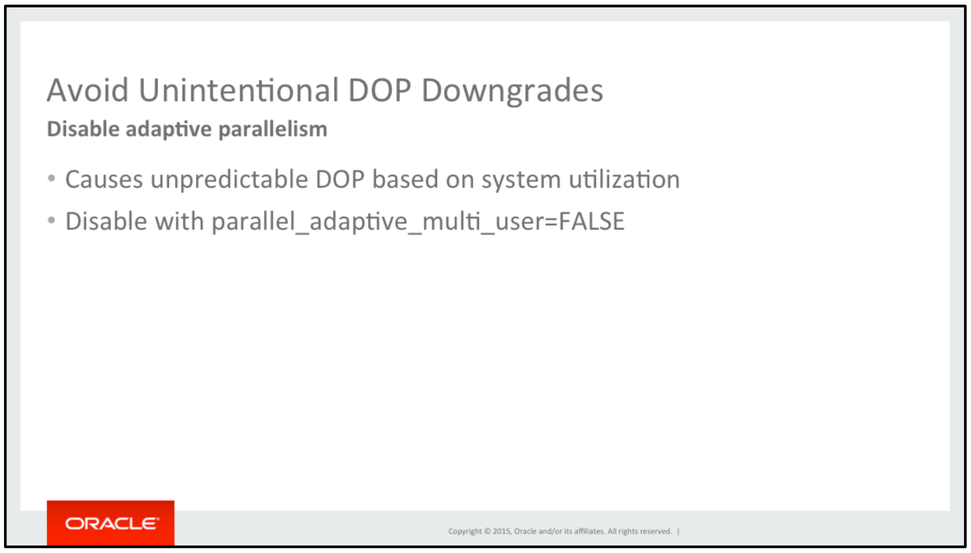
Now another statement is submitted with a DOP of 8. Even if there are enough processes in the pool we see that the statement gets downgraded to serial. Adaptive parallelism looks at the load of the system and may decide to downgrade statements if it think the system is over utilized. This again cause unpredictable performance because the DOP you will get depends on if this feature kicks in or not based on system load.



AVOID UNINTENTIONAL DOP DOWNGRADES

The DOP downgrades we have seen so far are undesirable or unintentional DOP downgrades. There is no way to predict if a statement will get downgraded or not, it depends on the number of available processes and the system load when the statement is submitted.

So, the first step out of seven is to avoid these kinds of downgrades.

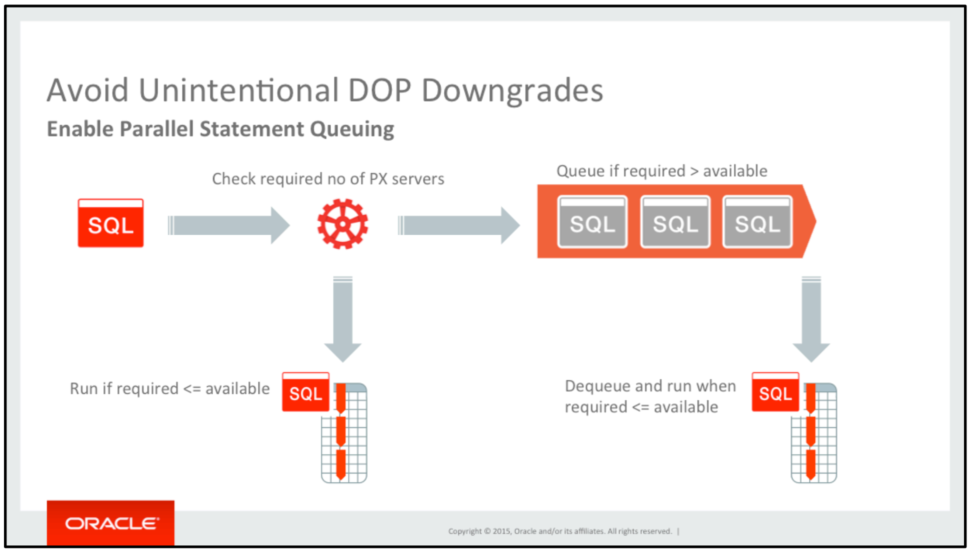


Avoid Unintentional DOP Downgrades

Disable adaptive parallelism

- Causes unpredictable DOP based on system utilization
- Disable with parallel_adaptive_multi_user=FALSE

The first thing to do to avoid unintentional downgrades Is to disable the adaptive parallelism feature.

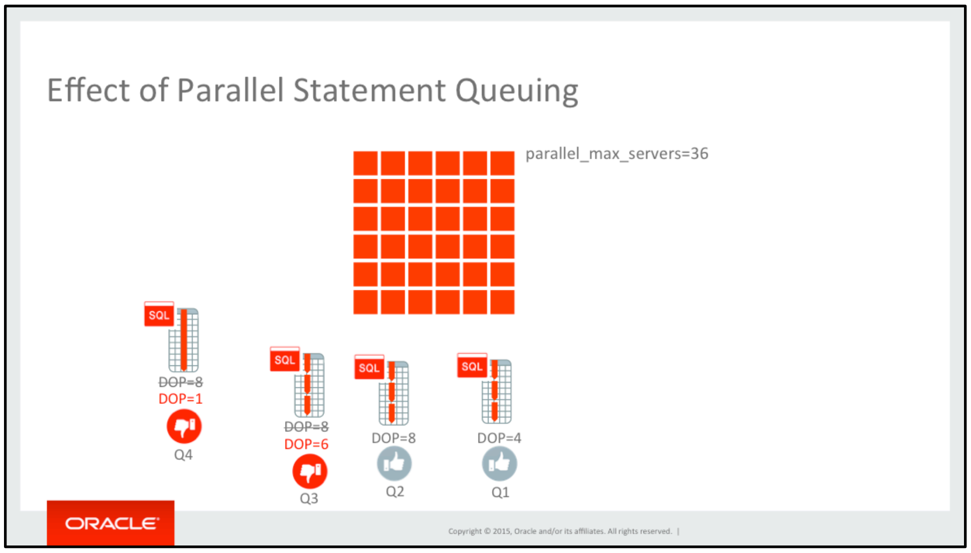


Avoid Unintentional DOP Downgrades

Enable Parallel Statement Queuing

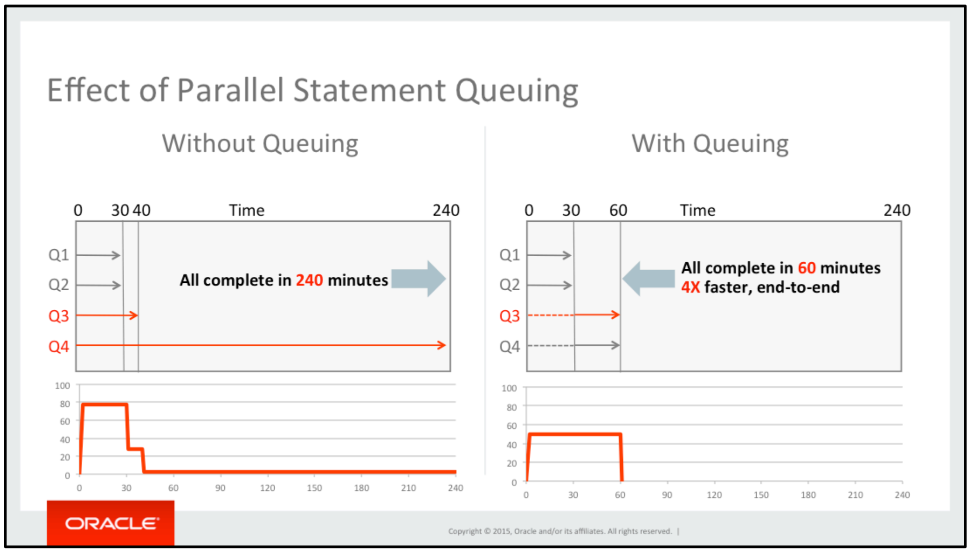
SQL
Check required no of PX servers
Queue if required > available
SQL
SQL
SQL
Run if required <= available
SQL
Dequeue and run when required <= available
SQL

The second thing to do is to enable parallel statement queuing.



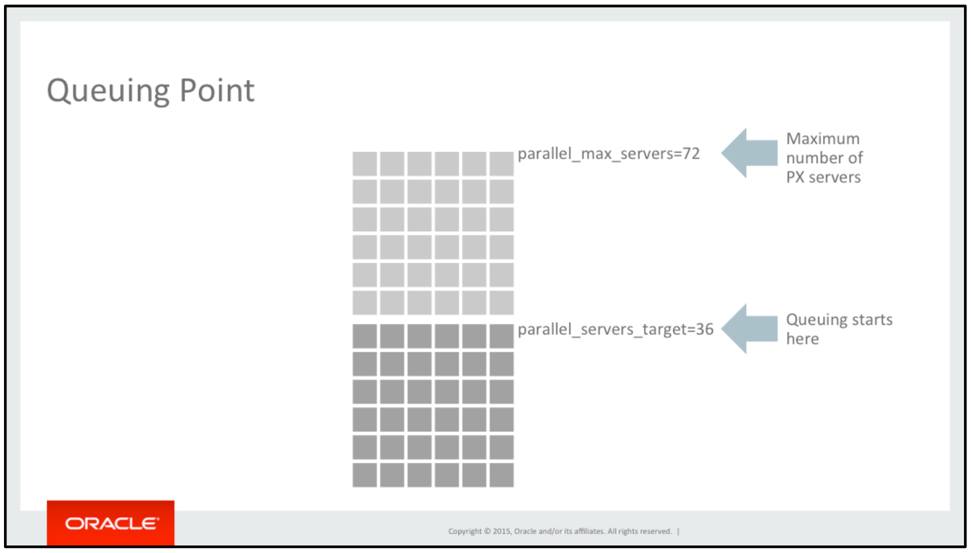
Effect of Parallel Statement Queuing

Let's go back to our first example to see the effect of parallel statement queuing. In that example we had 4 queries running concurrently. Q1 and Q2 were running without downgrades, Q3 and Q4 were downgraded to DOP=6 and DOP=1 respectively.



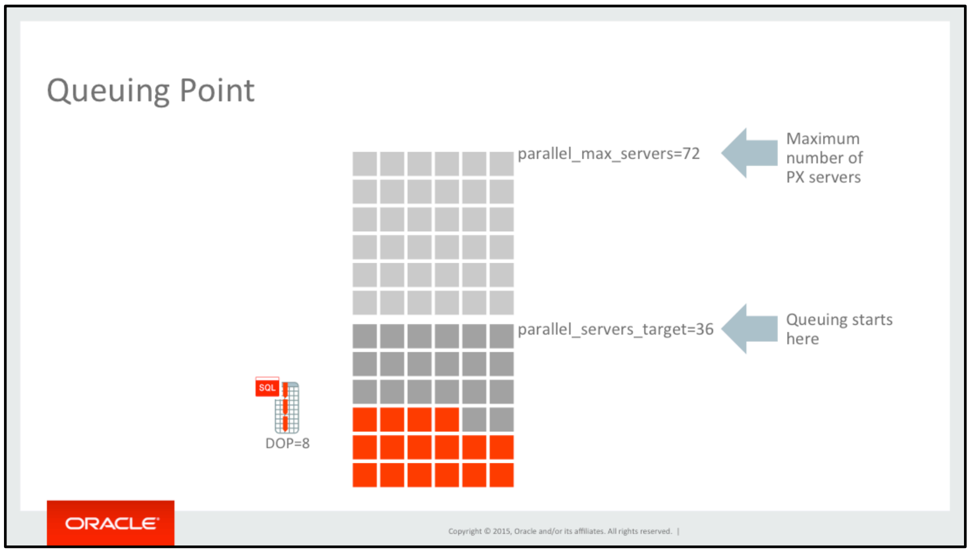
Without queuing we see that all are running at the same time, Q1 and Q2 finish in 30 minutes. Q3 finishes in 40 minutes and Q4 finishes in 240 minutes because it ran in serial. The system utilization is about 80% when all queries are running. After Q1, Q2 and Q3 finish the utilization is very low to the end because only one process is running.

If we enable parallel statement queuing for the same workload Q3 and Q4 get queued instead of getting downgraded. After Q1 and Q2 finish and their PX servers are released back to the pool Q3 and Q4 can start. Since they are not downgraded they finish much faster in this case and the whole workload finishes in 60 minutes compared to 240 minutes without queuing. If we look at the system utilization in this case we see that we use less resources and for a shorter time. This means for this workload we can increase the DOPs so that the queries finish even faster or we can submit more concurrent queries because there is ample CPU available.



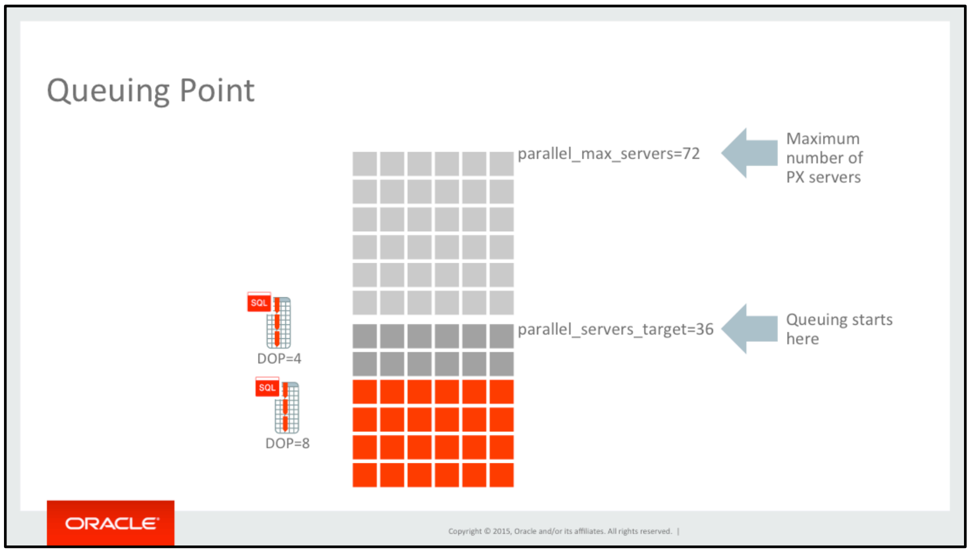
Queuing Point

When do we start queuing statement? The queuing point is determined by the parameter parallel_servers_target. When the number of in-use processes reach parallel_servers_target subsequent parallel statement will be queued. There is a reason why there are two parameters to control the number of processes and we will come to that later.



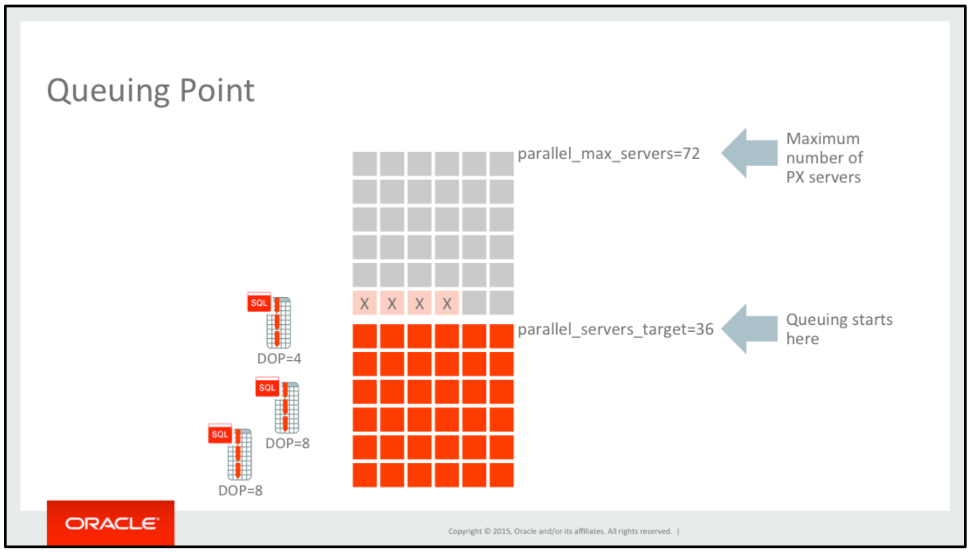
Queuing Point

These few slides show how queuing works.



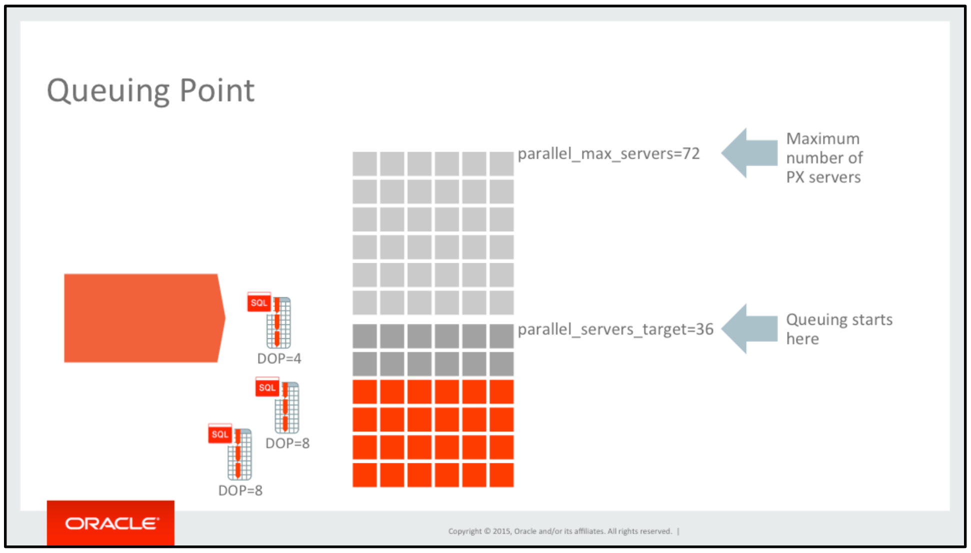
Queuing Point

We now have two queries running and they are using 24 processes.



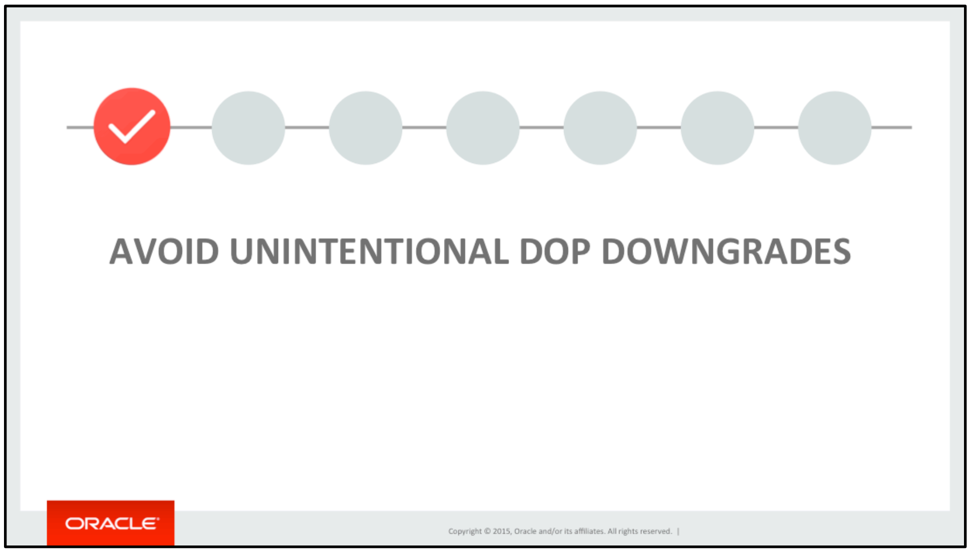
Queuing Point

When a new statement comes and asks for a DOP of 6 which means 16 processes the database sees that it will exceed parallel_servers_target.

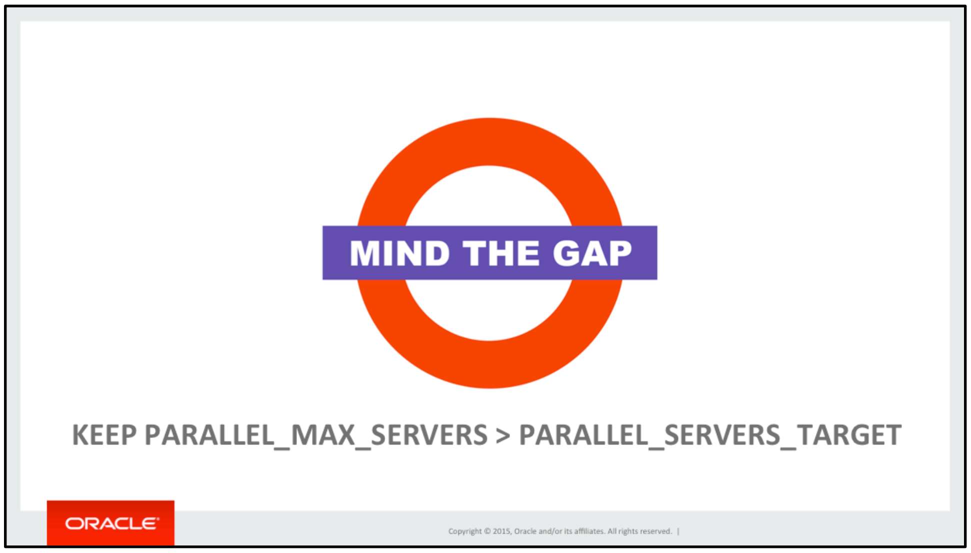


Queuing Point

This statement is not allowed to executed and is queued until 16 processes become available.



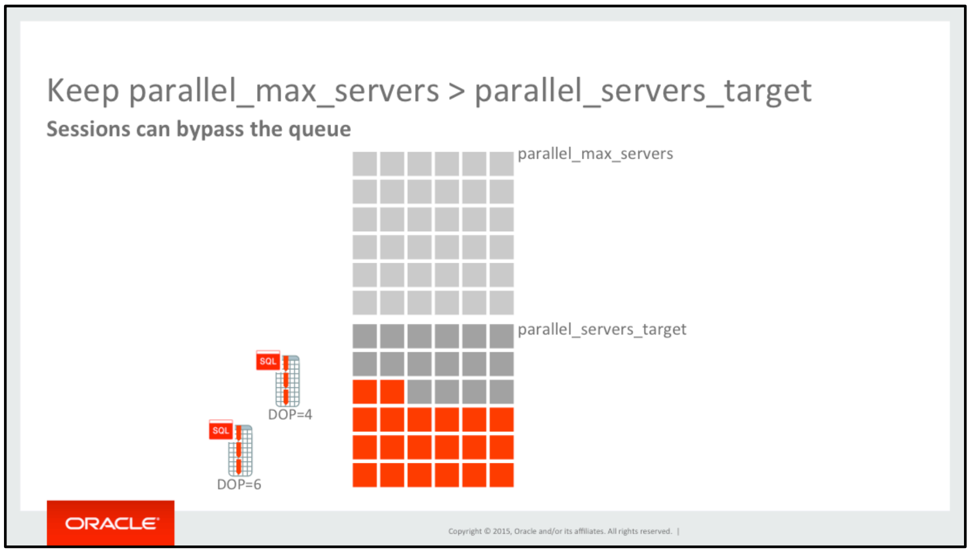
AVOID UNINTENTIONAL DOP DOWNGRADES


MIND THE GAP

KEEP PARALLEL_MAX_SERVERS > PARALLEL_SERVERS_TARGET

ORACLE

Now let's look at why you need to always put a gap between these two parameters.

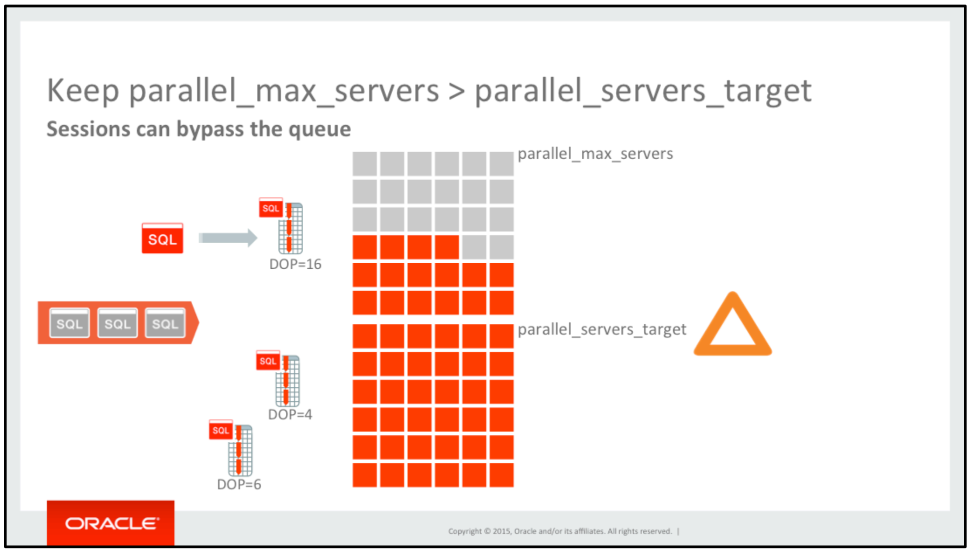


Keep parallel_max_servers > parallel_servers_target

Sessions can bypass the queue

Even if you enable parallel statement queuing system-wide there may still be queries bypassing the queue. These statements are allowed to use any process in the pool up to parallel_max_servers.

Here we see two statements running and there are only 16 processes left below parallel_servers_target.

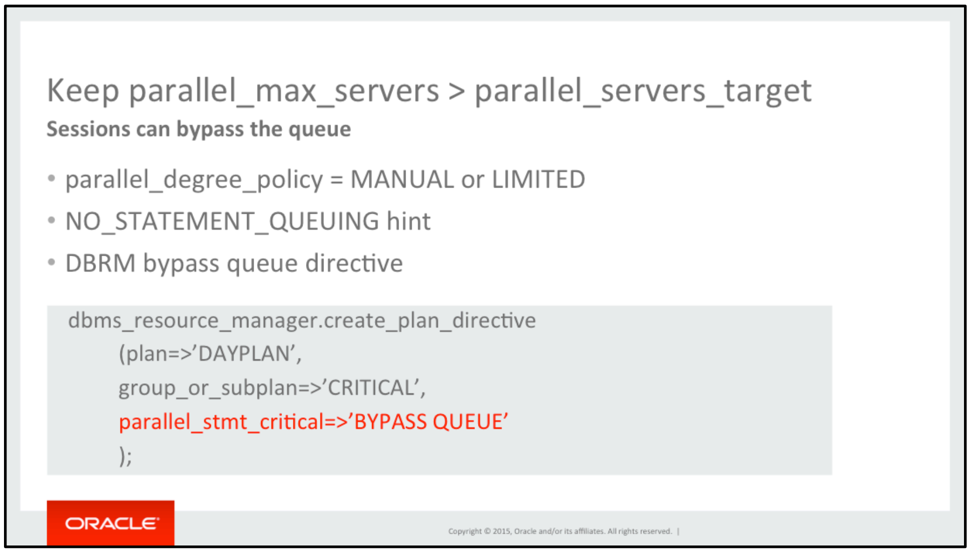


Keep parallel_max_servers > parallel_servers_target

Sessions can bypass the queue

A statement that is allowed to bypass the queue comes and asks for a DOP of 16 which means 32 processes. Since this statement is allowed to bypass the queue it goes above parallel_servers_target and allocates 32 processes. Now parallel_servers_target is exceed and any new statement will be queued. If more statements bypass the queue this may cause the statements in the queue wait forever.

This is why you need to prevent bypassing the queue at all. There are other ways to make sure critical users get the processes they need without waiting, we will talk about those methods later.



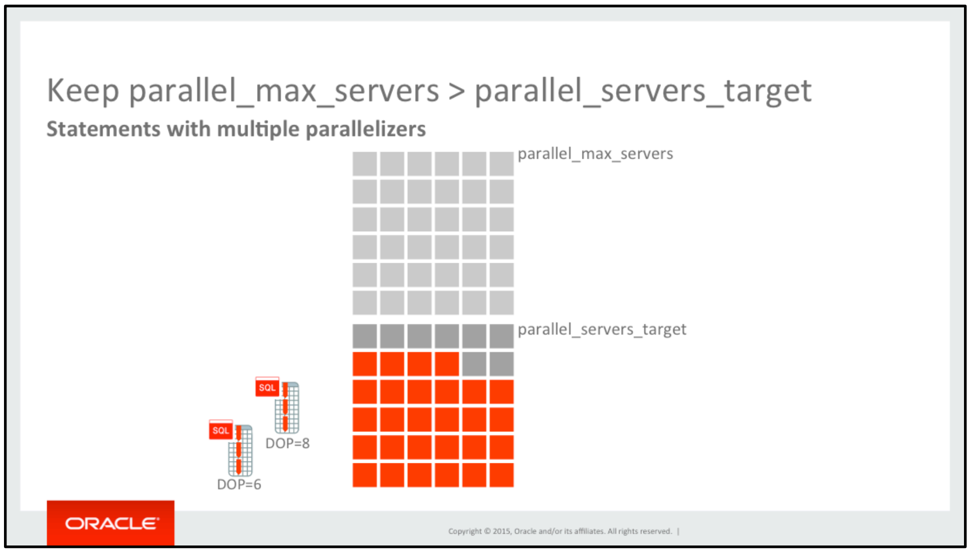
Keep parallel_max_servers > parallel_servers_target

Sessions can bypass the queue

- parallel_degree_policy = MANUAL or LIMITED
- NO_STATEMENT_QUEUING hint
- DBRM bypass queue directive

```
dbms_resource_manager.create_plan_directive
        (plan=>'DAYPLAN',
        group_or_subplan=>'CRITICAL',
        parallel_stmt_critical=>'BYPASS QUEUE'
        );
```

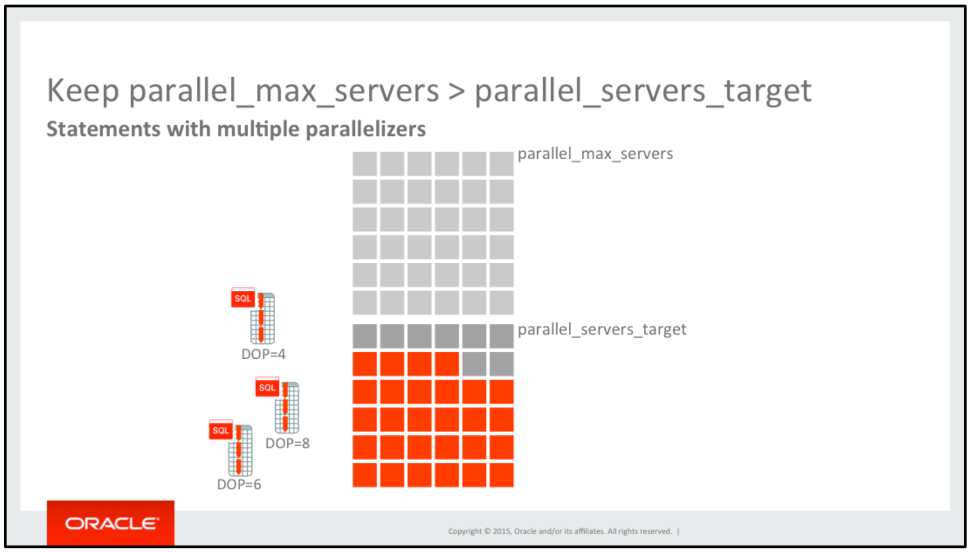
These are the ways a session can bypass the queue.



Keep parallel_max_servers > parallel_servers_target

Statements with multiple parallelizers

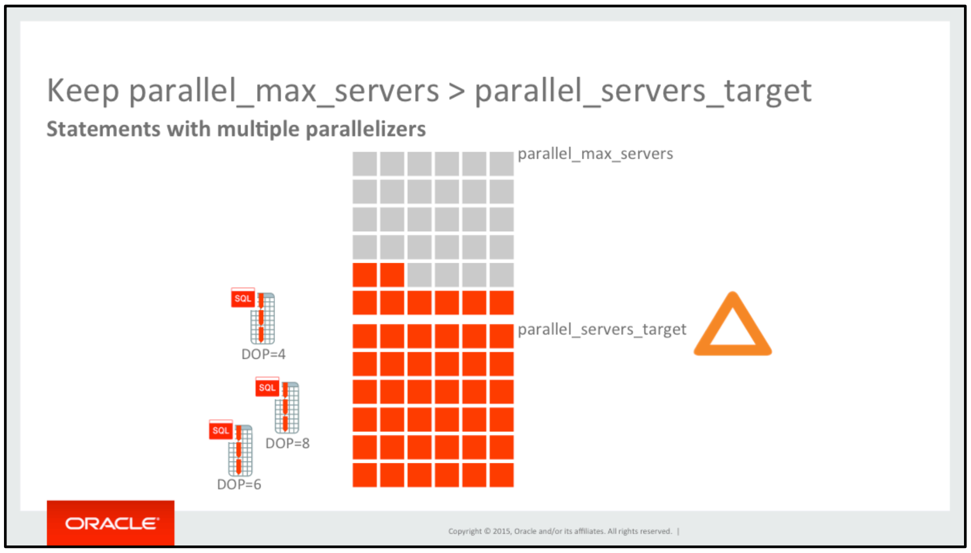
Another case when a statement can exceed parallel_servers_target is multiple parallelizers. Multiple parallelizers are statements that use more than DOP*2 number of processes.



Keep parallel_max_servers > parallel_servers_target

Statements with multiple parallelizers

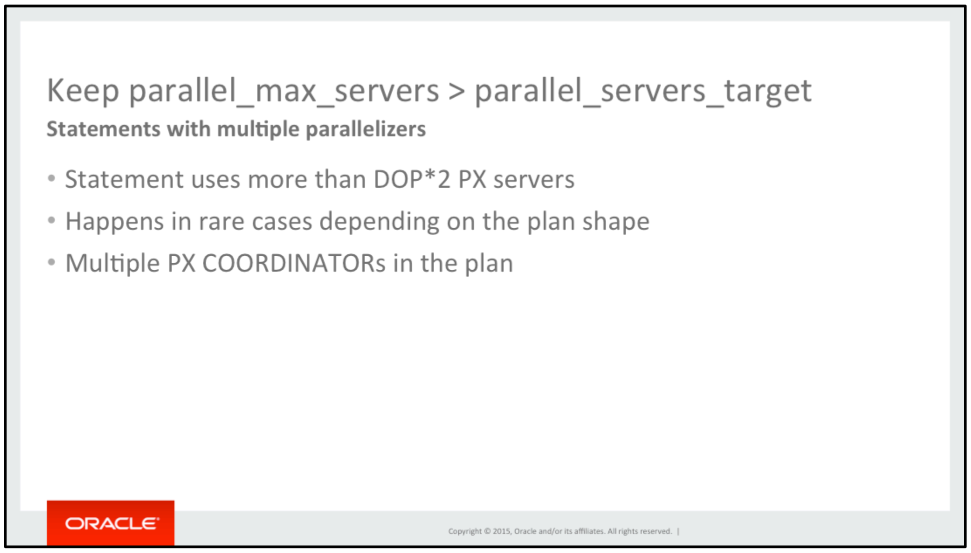
Here we see that a statement is asking for a DOP of 4. Since this means 8 processes by default the database will allow this statement to run because there are 8 processes available below parallel_servers_target.



Keep parallel_max_servers > parallel_servers_target

Statements with multiple parallelizers

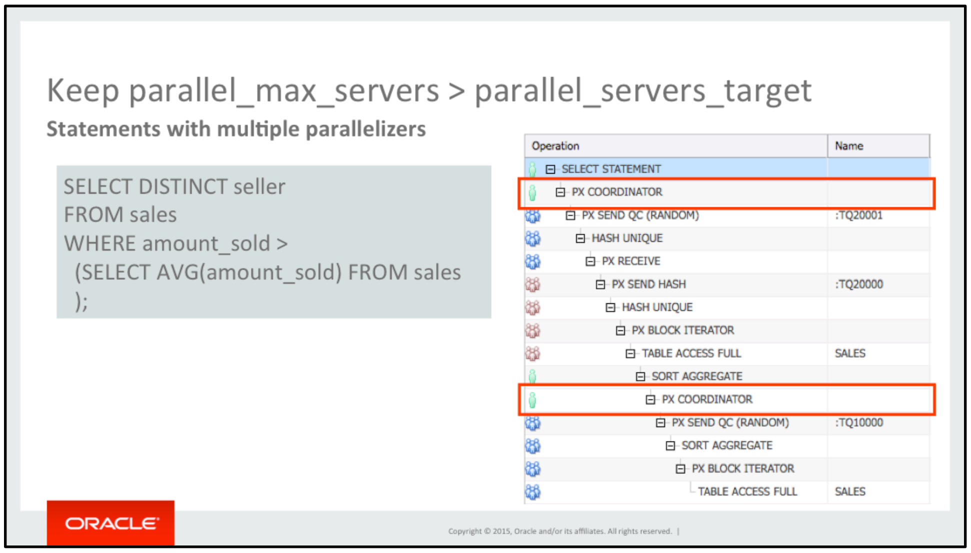
But we see that this statement used 16 processes instead of 8. This is because the plan for this statement had multiple parallelizers and used more than DOP*2 processes.



Keep parallel_max_servers > parallel_servers_target

Statements with multiple parallelizers

- Statement uses more than DOP*2 PX servers
- Happens in rare cases depending on the plan shape
- Multiple PX COORDINATORs in the plan

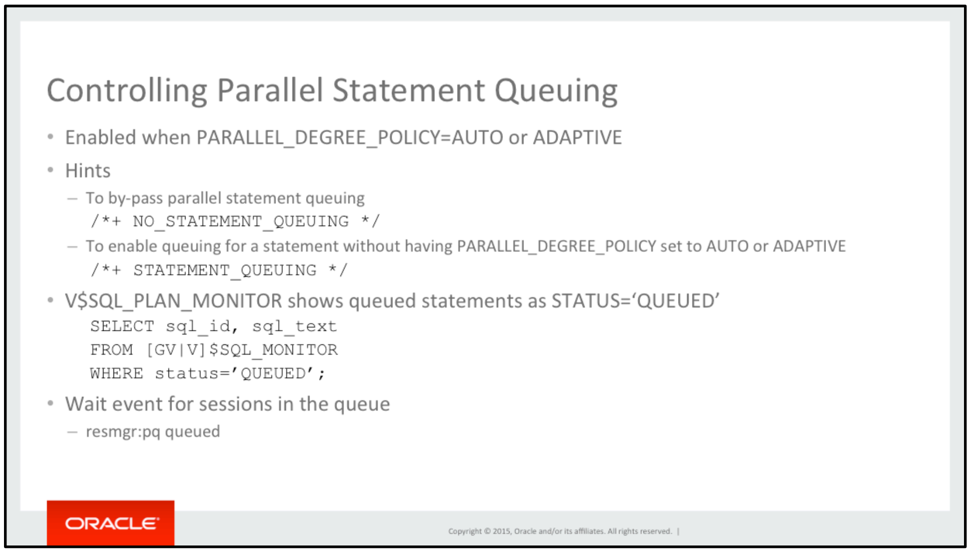


Keep parallel_max_servers > parallel_servers_target

Statements with multiple parallelizers

```
SELECT DISTINCT seller
FROM sales
WHERE amount_sold >
  (SELECT AVG(amount_sold) FROM sales
  );
```

```
Operation                                   Name
SELECT STATEMENT
  PX COORDINATOR
    PX SEND QC (RANDOM)                     :TQ20001
      HASH UNIQUE
        PX RECEIVE
          PX SEND HASH                      :TQ20000
            HASH UNIQUE
              PX BLOCK ITERATOR
                TABLE ACCESS FULL           SALES
                  SORT AGGREGATE
                    PX COORDINATOR
                      PX SEND QC (RANDOM)   :TQ10000
                        SORT AGGREGATE
                          PX BLOCK ITERATOR
                            TABLE ACCESS FULL SALES
```



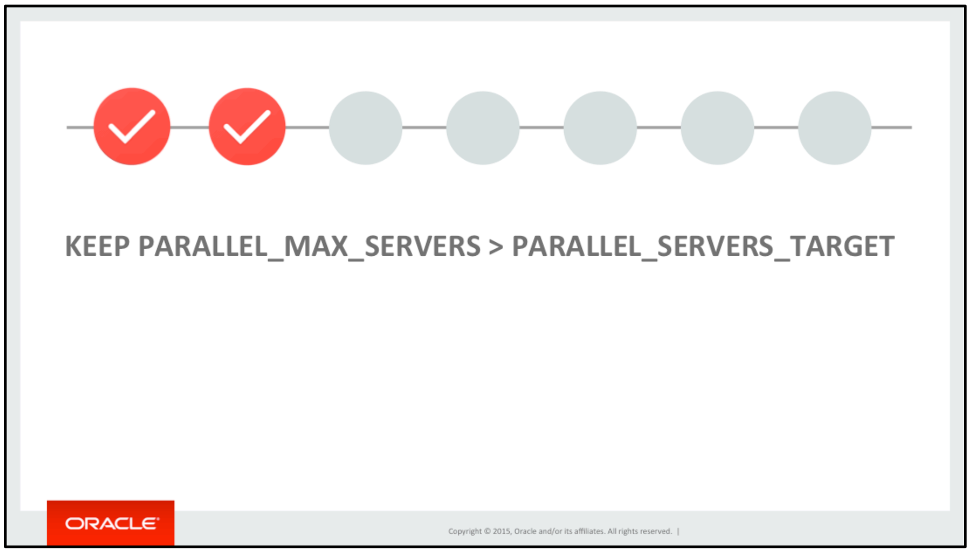
Controlling Parallel Statement Queuing

- Enabled when PARALLEL_DEGREE_POLICY=AUTO or ADAPTIVE
- Hints
  - To by-pass parallel statement queuing

    ```
    /*+ NO_STATEMENT_QUEUING */
    ```
  - To enable queuing for a statement without having PARALLEL_DEGREE_POLICY set to AUTO or ADAPTIVE

    ```
    /*+ STATEMENT_QUEUING */
    ```
- V$SQL_PLAN_MONITOR shows queued statements as STATUS='QUEUED'

  ```
  SELECT sql_id, sql_text
  FROM [GV|V]$SQL_MONITOR
  WHERE status='QUEUED';
  ```
- Wait event for sessions in the queue
  - resmgr:pq queued

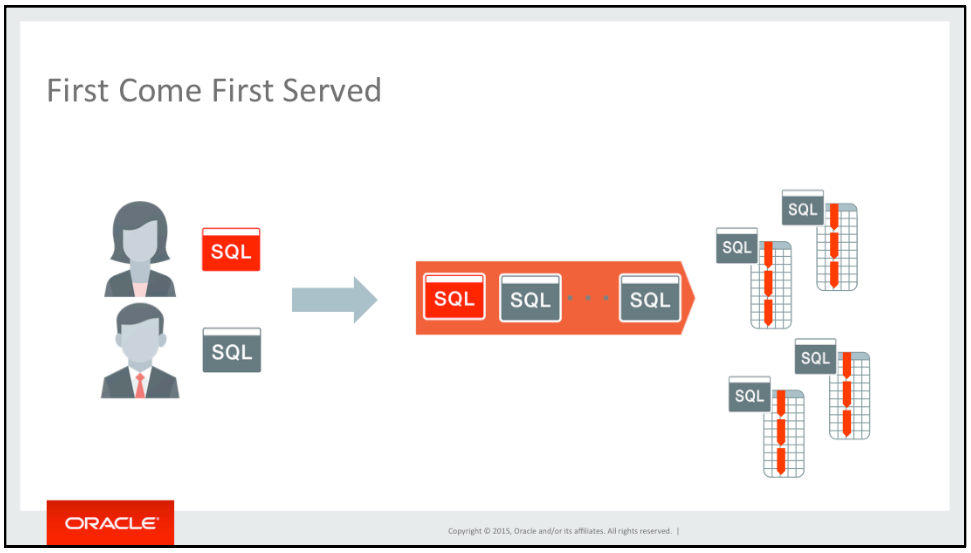


KEEP PARALLEL_MAX_SERVERS > PARALLEL_SERVERS_TARGET

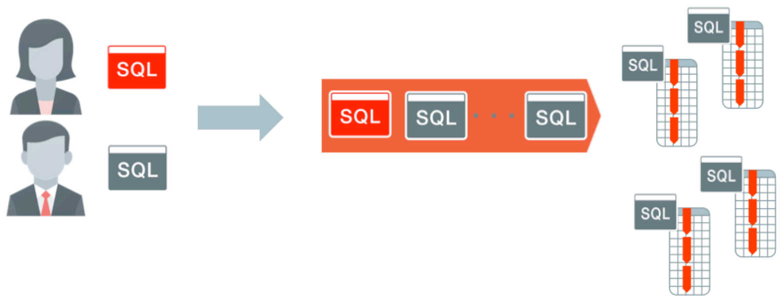
ORACLE

PROTECT THE INNOCENT

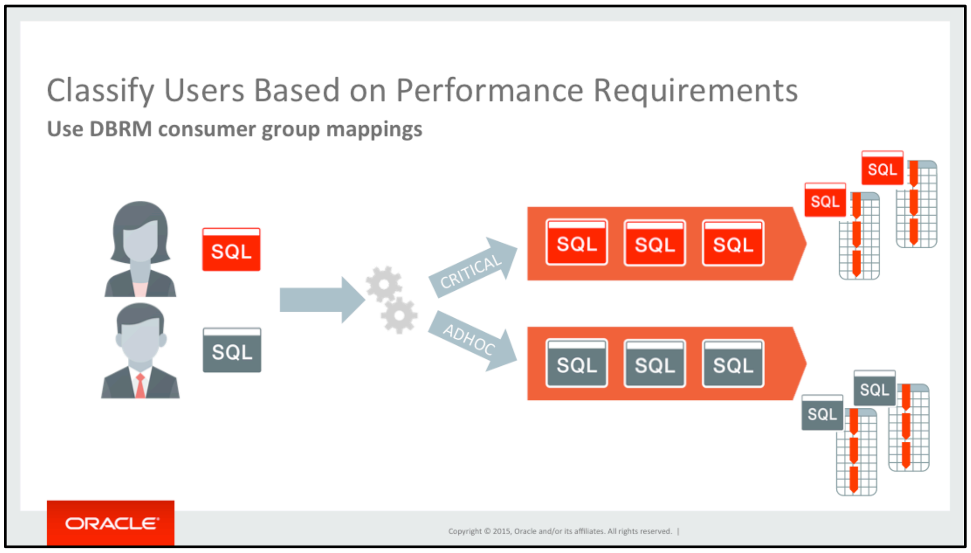
ORACLE



First Come First Served



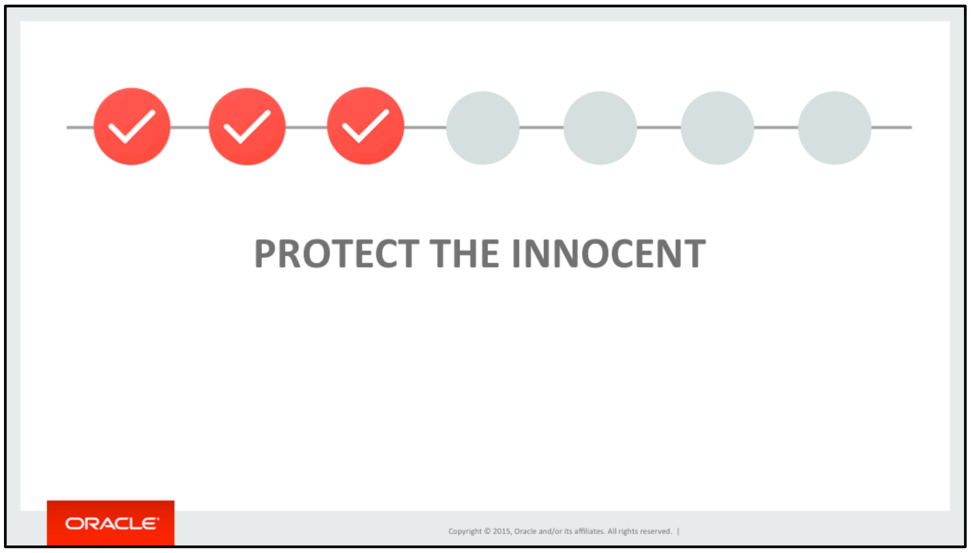
The parallel statement queue acts as a first-in first-out queue. Since the database treats each user equally any user can submit any number of statements and they will go into the queue when needed. In this example we see that user John submitted lots of statements before user Jane. Some of these statements are running and lots more are waiting in the queue. When Jane submits a statement it will go to the end of the queue and will wait until all of John's statements finish.



Classify Users Based on Performance Requirements

Use DBRM consumer group mappings

When you use Database Resource Manager and consumer groups each consumer group will have its own queue. By default DBRM will treat each queue equally and will try to pick from each queue equally. This way Jane will not have to wait behind John and will have an equal chance of running.

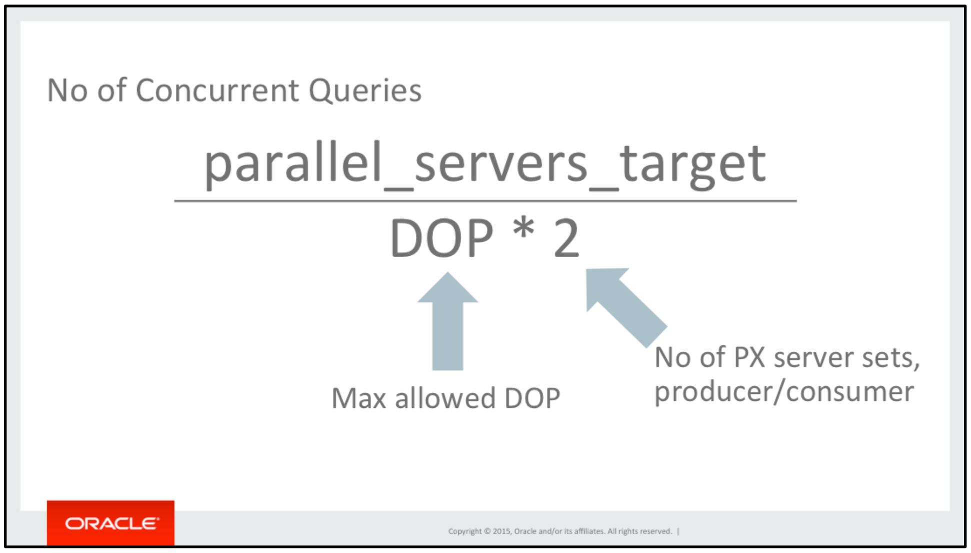


PROTECT THE INNOCENT

DOP

LIMIT THE DOP FOR CONCURRENCY

We talked about unintentional DOP downgrades causing unpredictable performance before. You may want to limit the DOPs intentionally to achieve concurrency in the system.



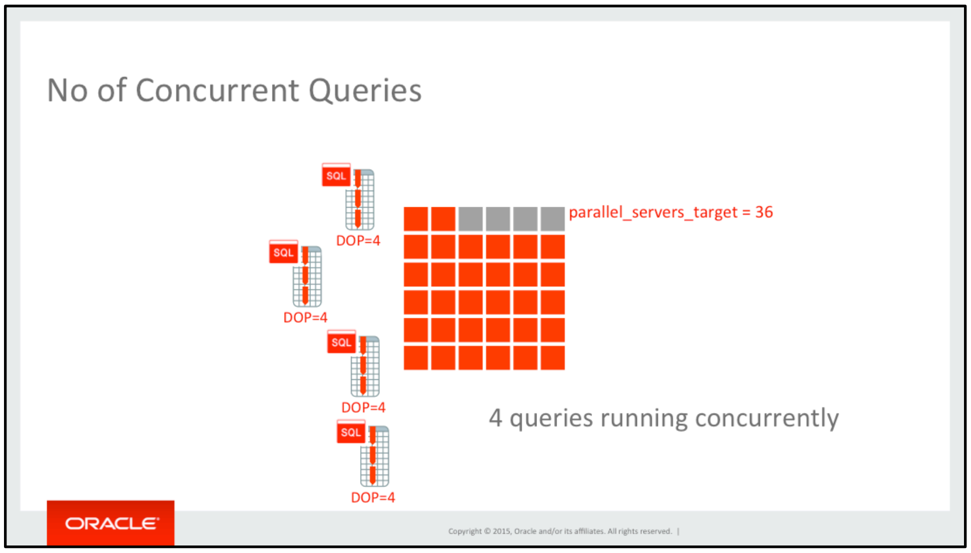
No of Concurrent Queries

$$\frac{\text{parallel\_servers\_target}}{\text{DOP} * 2}$$

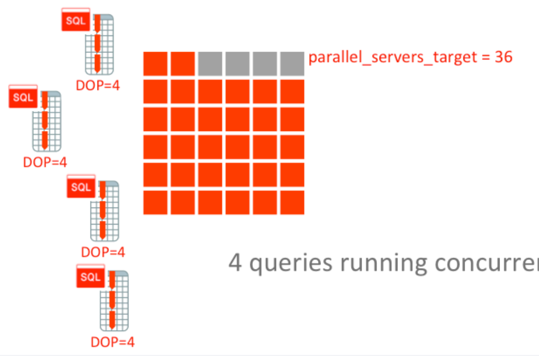
Max allowed DOP
No of PX server sets, producer/consumer
ORACLE
Copyright © 2015, Oracle and/or its affiliates. All rights reserved. |

Conservatively the number of concurrent queries you can run can be calculated by this formula. This assume everyone is using the max DOP and there are no exceptions like multiple parallelizers.

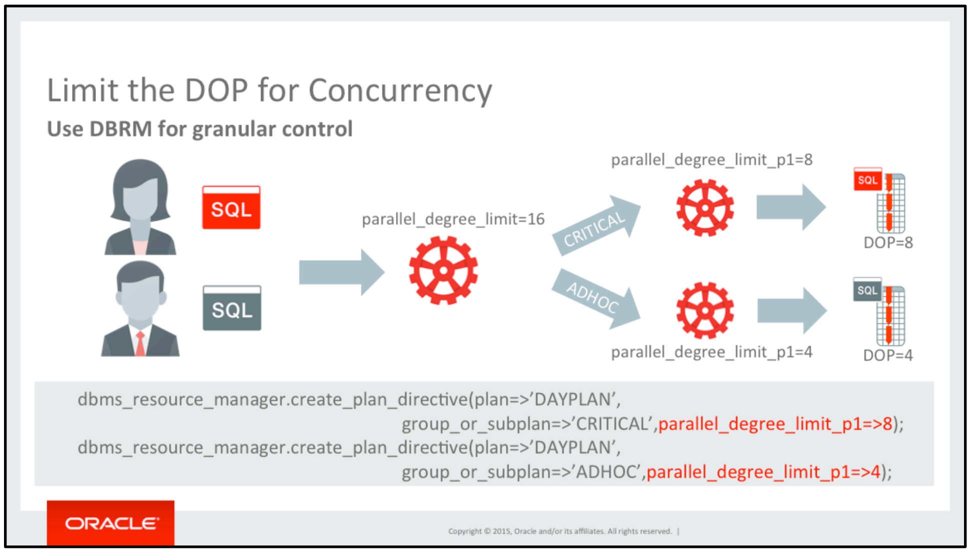
This shows that you can test with different parallel_servers_target and max DOP limits to achieve the concurrency you need.



No of Concurrent Queries



4 queries running concurrently

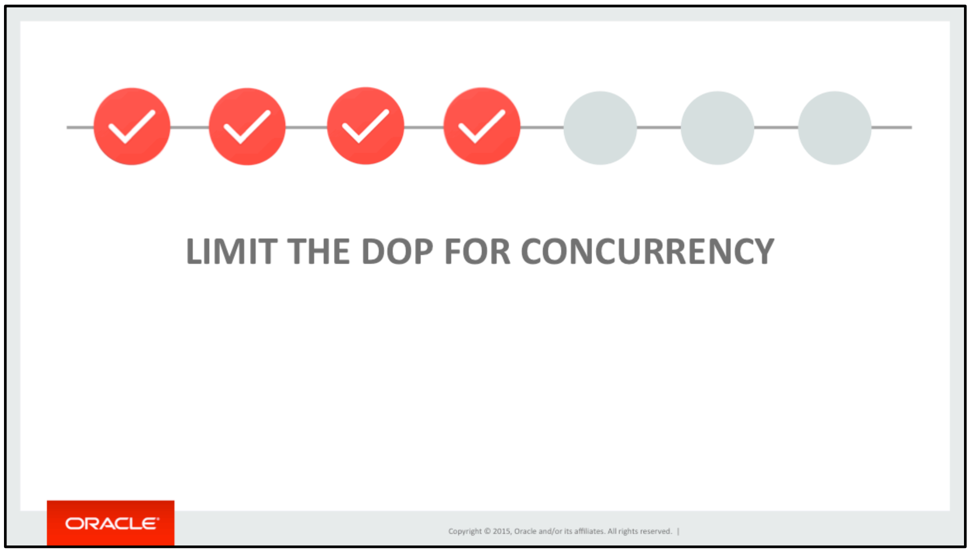


Limit the DOP for Concurrency

Use DBRM for granular control

```
dbms_resource_manager.create_plan_directive(plan=>'DAYPLAN',
                                group_or_subplan=>'CRITICAL',parallel_degree_limit_p1=>8);
dbms_resource_manager.create_plan_directive(plan=>'DAYPLAN',
                                group_or_subplan=>'ADHOC',parallel_degree_limit_p1=>4);
```

There are two ways to limit the DOP for users. The parameter parallel_degree_limit acts as a global limit and limits every session that is using Auto DOP. The DBRM directive parallel_degree_limit_p1 provides granular control over DOPs so that you can put different limits for different users.

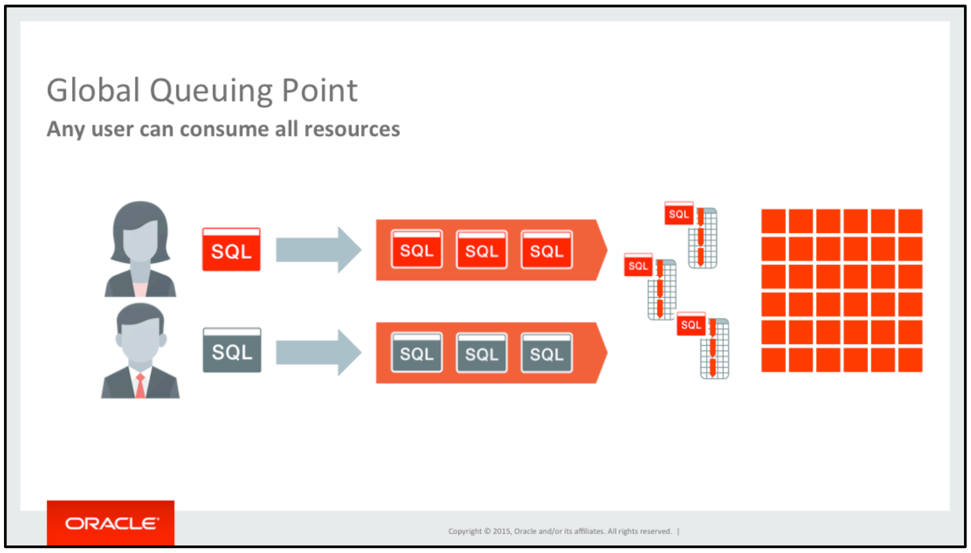


LIMIT THE DOP FOR CONCURRENCY

ORACLE

MAKE SOME USERS MORE EQUAL THAN THE OTHERS

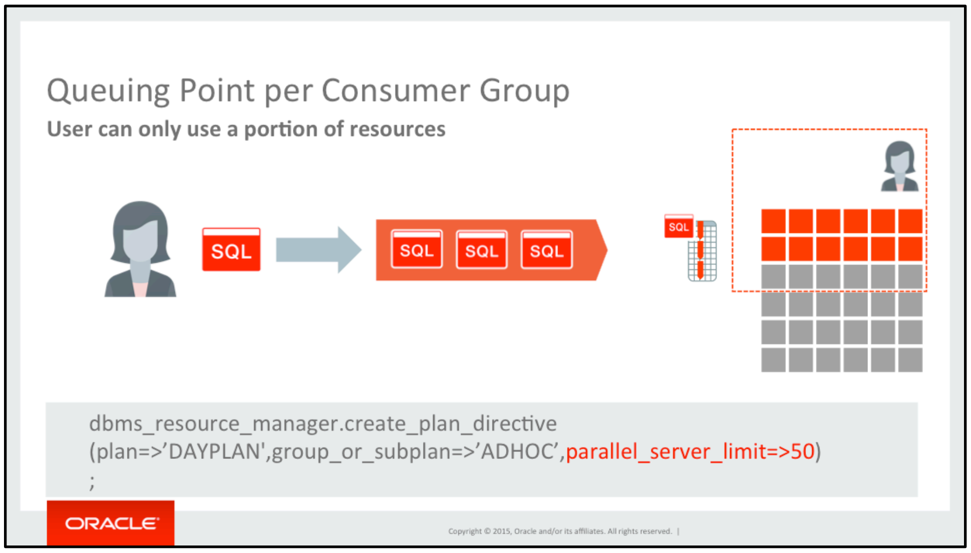
Not every user is equal in the DW world. You need to make sure users are prioritized based on performance and business requirements.



Global Queuing Point

Any user can consume all resources

The queuing point set by parallel_servers_target is global. If one user grabs all processes until that point all others users need to wait in the queue. Even if you use consumer groups and different queues you need to wait for other user's queries to finish. This example shows that Jane submitted several statements before John and grabbed all processes.



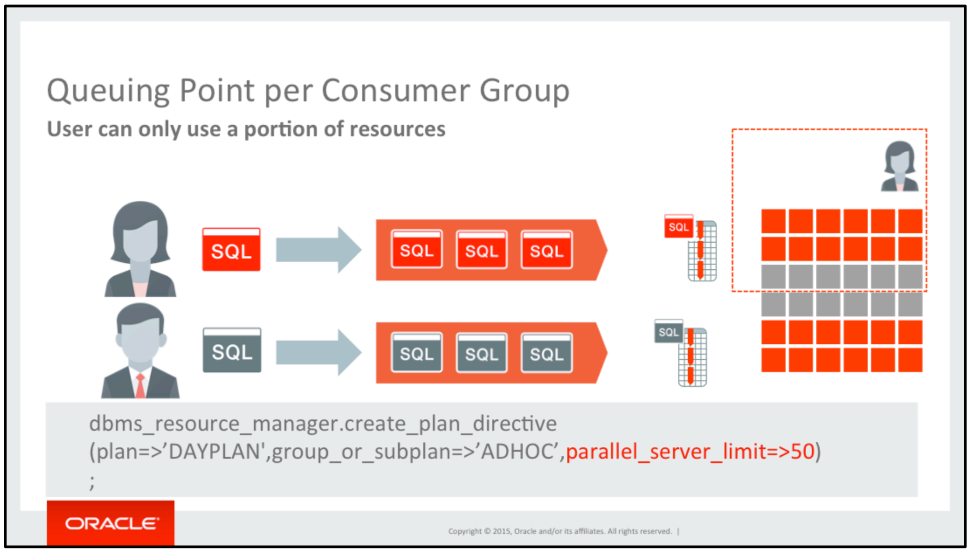
Queuing Point per Consumer Group

User can only use a portion of resources

SQL
SQL
SQL
SQL
SQL

```
dbms_resource_manager.create_plan_directive
(plan=>'DAYPLAN',group_or_subplan=>'ADHOC',parallel_server_limit=>50)
;
```

The DBRM directive parallel_server_limit prevents a consumer group from using all processes up to parallel_servers_target. This example shows that Jane can only user 50% of parallel_servers_target. When she uses that many processes her subsequent statements will be queued.

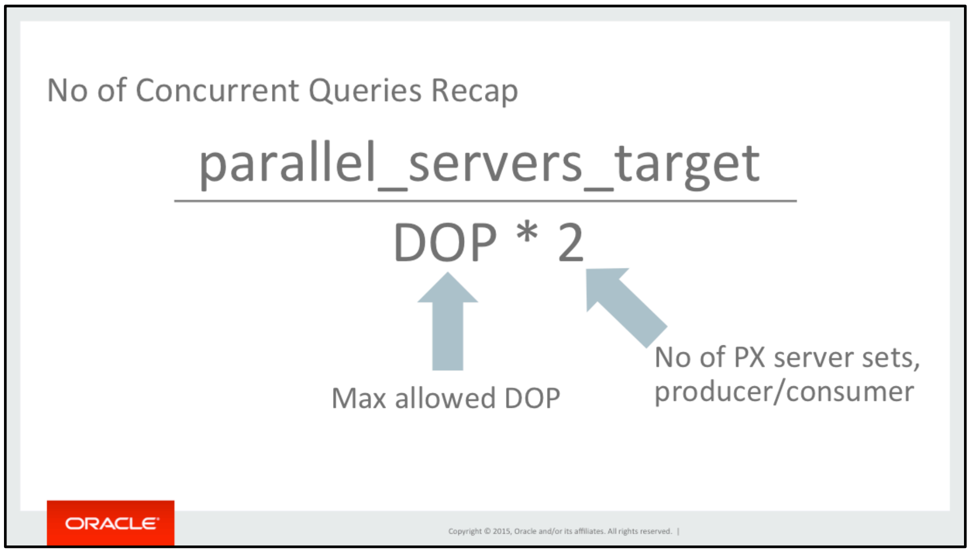


Queuing Point per Consumer Group

User can only use a portion of resources

```
dbms_resource_manager.create_plan_directive
(plan=>’DAYPLAN',group_or_subplan=>’ADHOC’,parallel_server_limit=>50)
;
```

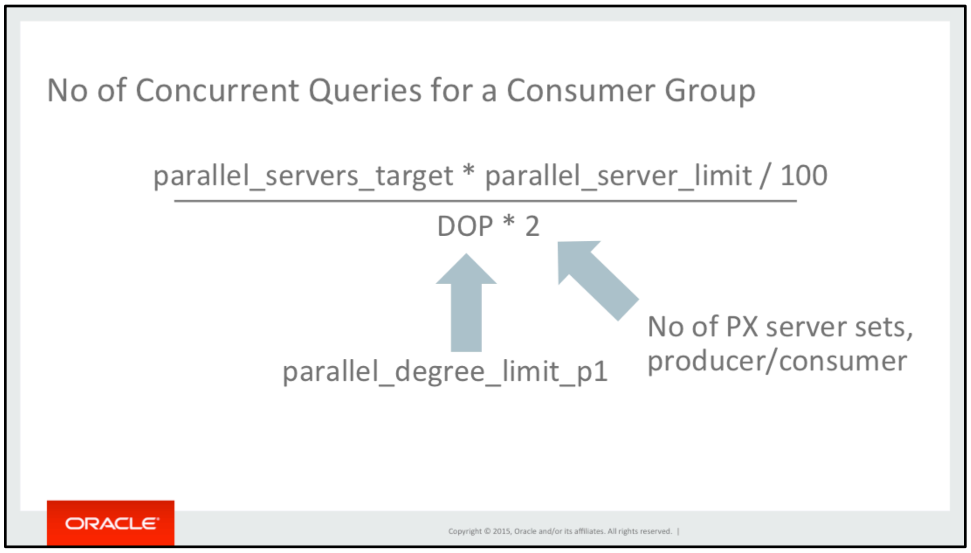
This way another user can come and use the remaining processes without waiting for her.



No of Concurrent Queries Recap

$$\frac{\text{parallel\_servers\_target}}{\text{DOP} * 2}$$

Max allowed DOP
No of PX server sets, producer/consumer

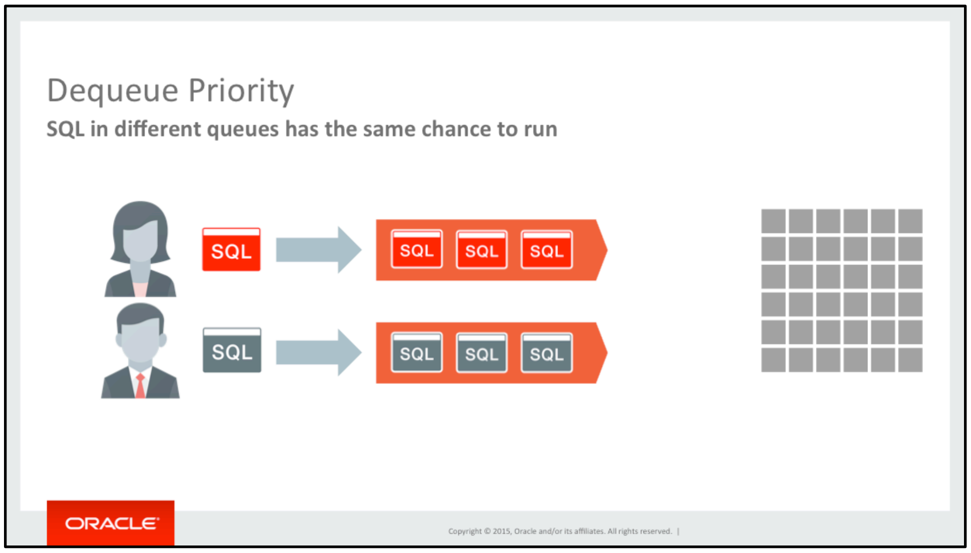


No of Concurrent Queries for a Consumer Group

$$\frac{\text{parallel\_servers\_target * parallel\_server\_limit / 100}}{\text{DOP * 2}}$$

parallel_degree_limit_p1
No of PX server sets, producer/consumer

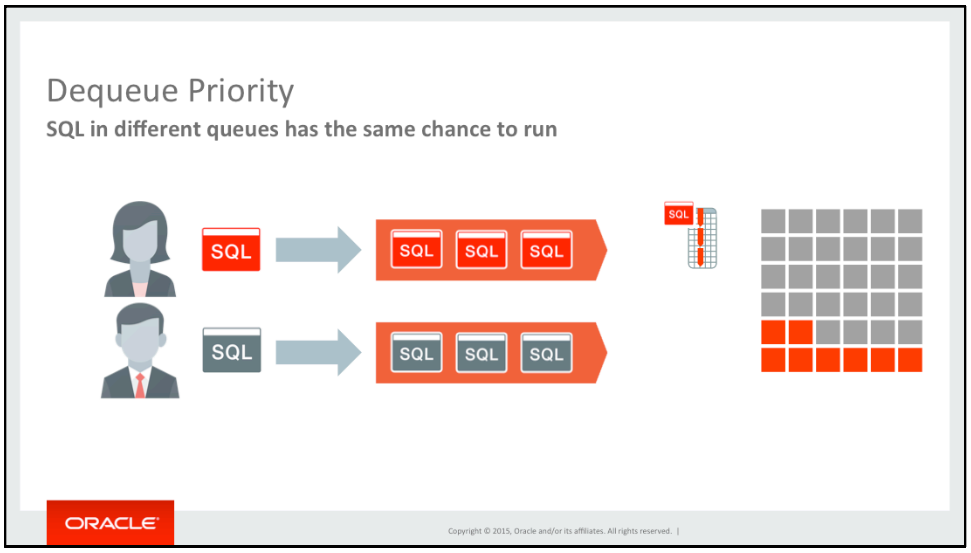
This slide shows how the formula change when you use parallel_servers_limit and parallel_degree_limit_p1 for a consumer group. Basically the queuing point for the consumer group becomes parallel_servers_target*parallel_server_limit/100.



Dequeue Priority

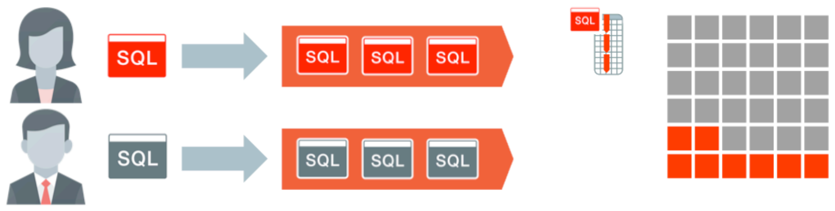
SQL in different queues has the same chance to run

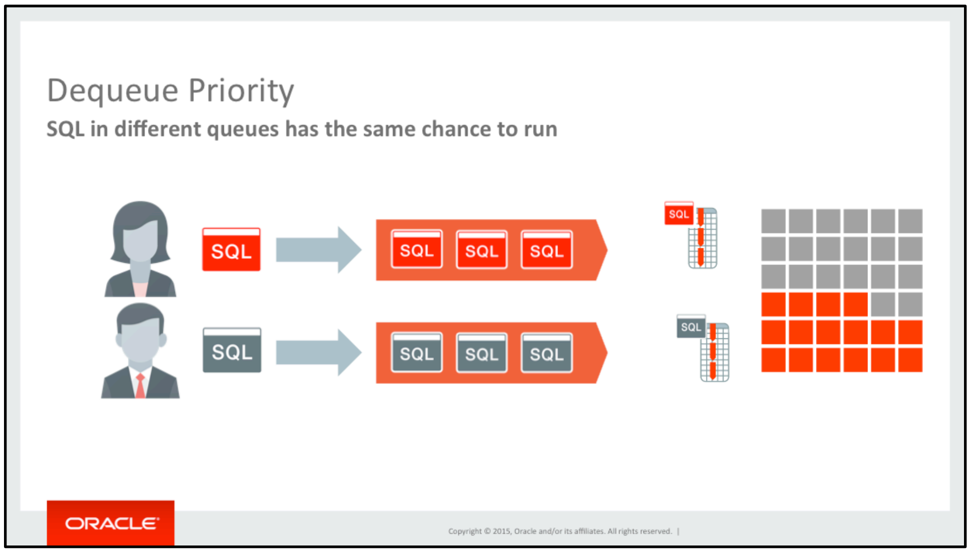
The second thing you can do to make sure critical users are prioritized is to use the dequee priorities.



Dequeue Priority

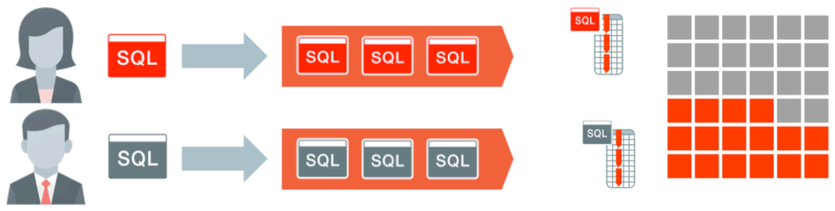
SQL in different queues has the same chance to run

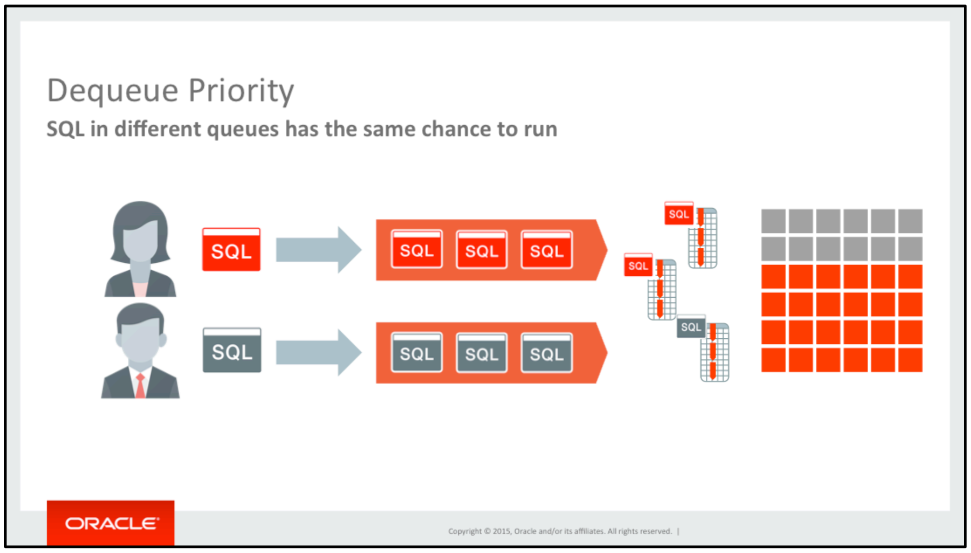




Dequeue Priority

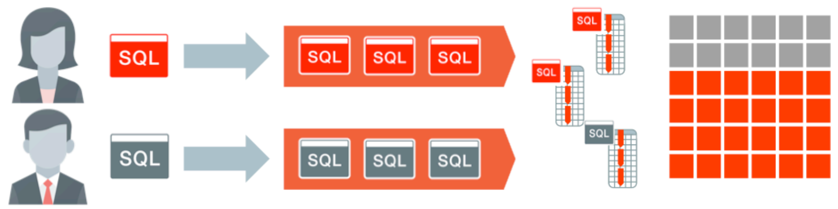
SQL in different queues has the same chance to run

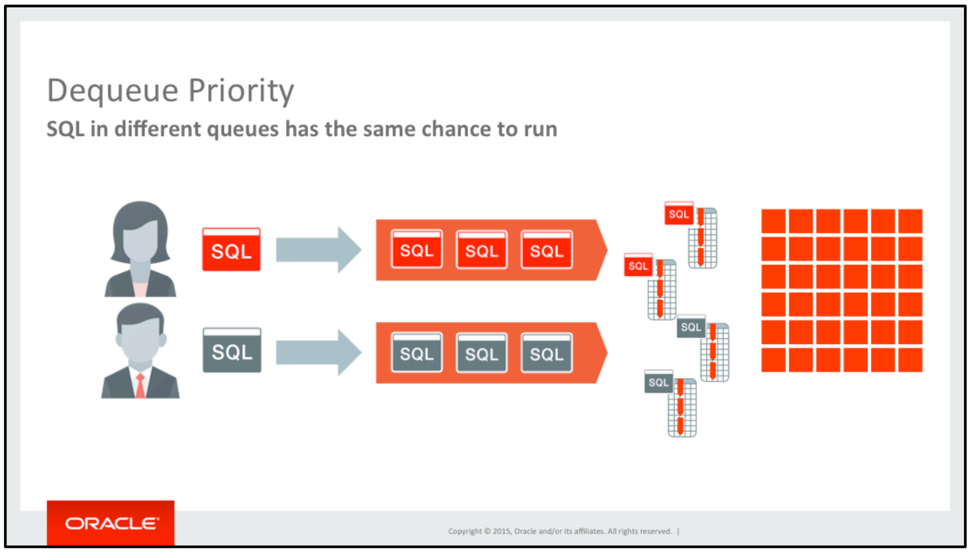




Dequeue Priority

SQL in different queues has the same chance to run





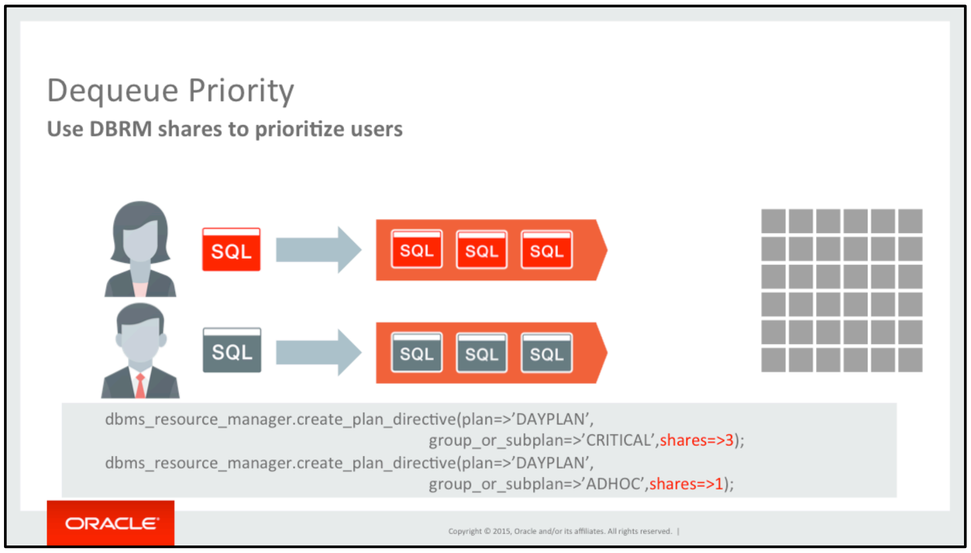
Dequeue Priority

SQL in different queues has the same chance to run

SQL
SQL
SQL
SQL
SQL
SQL
SQL
SQL
SQL
SQL
SQL
SQL

ORACLE

By default DBRM treats each queue equally. It tries to pick statements from each queue without any priority.



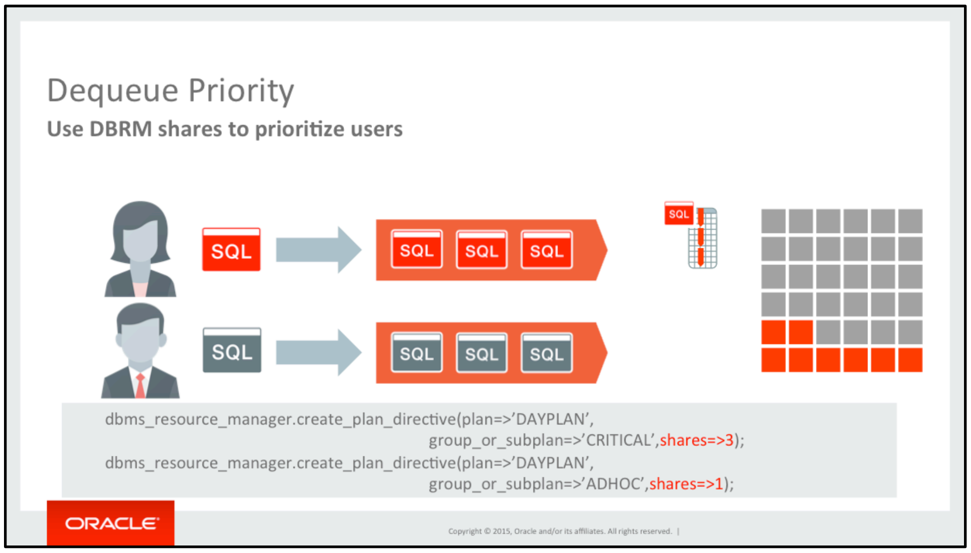
Dequeue Priority

Use DBRM shares to prioritize users

SQL
SQL
SQL
SQL
SQL
SQL
SQL
SQL

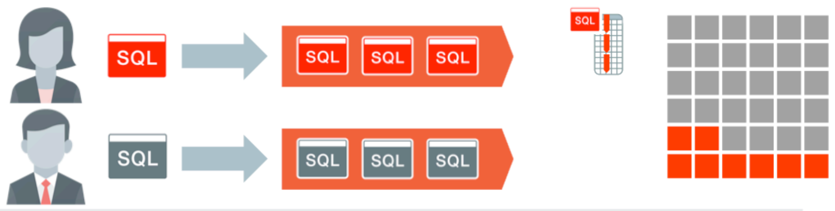
```
dbms_resource_manager.create_plan_directive(plan=>'DAYPLAN',
                                            group_or_subplan=>'CRITICAL',shares=>3);
dbms_resource_manager.create_plan_directive(plan=>'DAYPLAN',
                                            group_or_subplan=>'ADHOC',shares=>1);
```

The shares directive in DBRM determines the CPU allocation for consumer groups. You can use the same directive as dequeue priority for consumer group queues. This example shows that DBRM will try to favor Jane over John when it is picking up queries from the queues because of the shares settings.

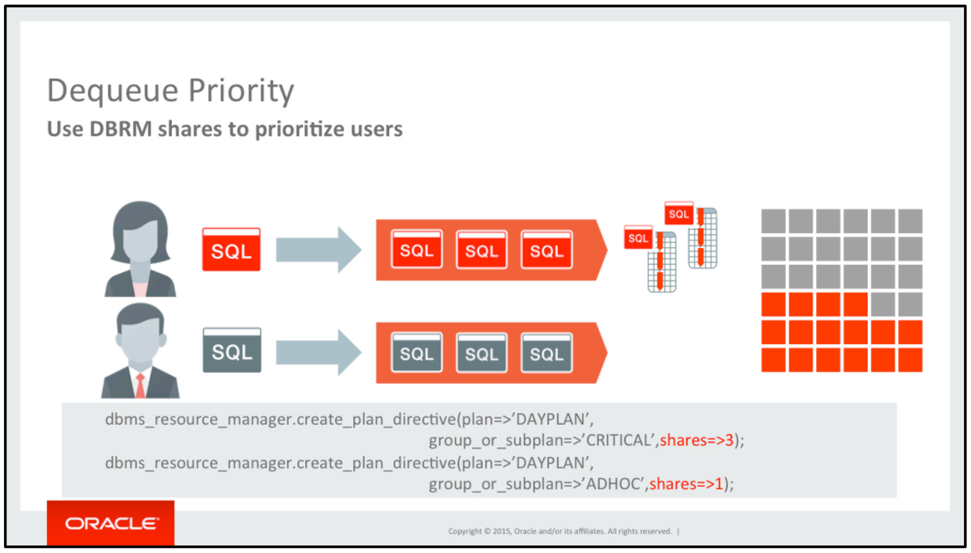


Dequeue Priority

Use DBRM shares to prioritize users

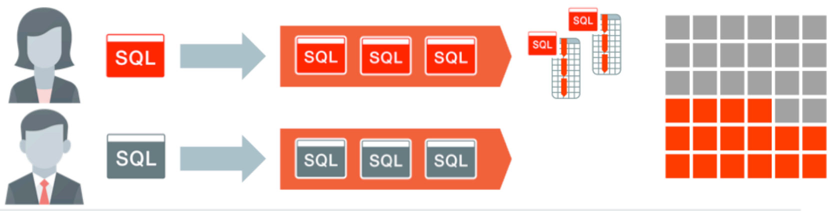


```
dbms_resource_manager.create_plan_directive(plan=>'DAYPLAN',
                                            group_or_subplan=>'CRITICAL',shares=>3);
dbms_resource_manager.create_plan_directive(plan=>'DAYPLAN',
                                            group_or_subplan=>'ADHOC',shares=>1);
```

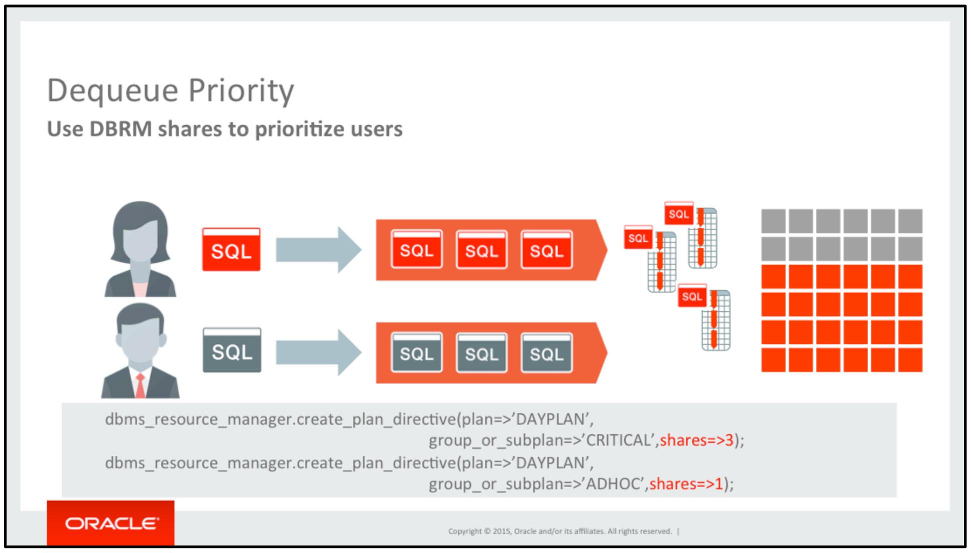


Dequeue Priority

Use DBRM shares to prioritize users

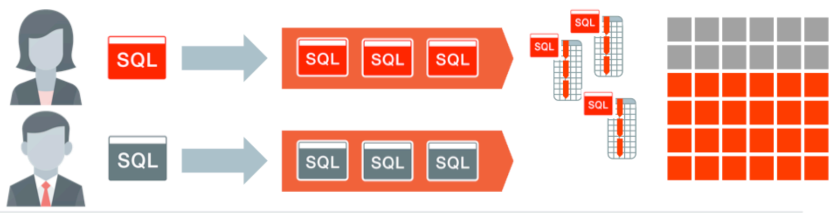


```
dbms_resource_manager.create_plan_directive(plan=>'DAYPLAN',
                                            group_or_subplan=>'CRITICAL',shares=>3);
dbms_resource_manager.create_plan_directive(plan=>'DAYPLAN',
                                            group_or_subplan=>'ADHOC',shares=>1);
```

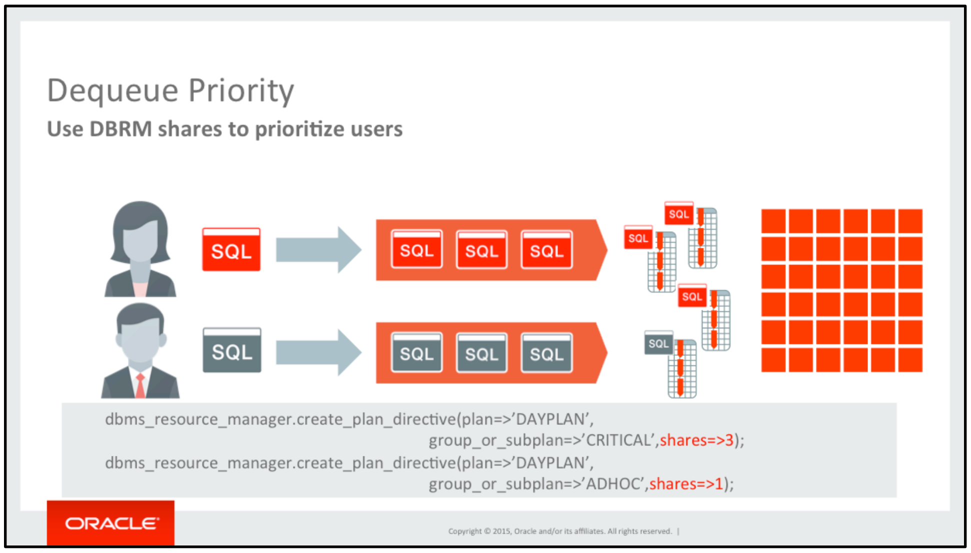


Dequeue Priority

Use DBRM shares to prioritize users

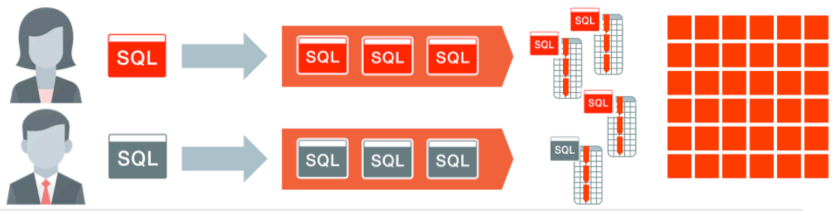


```
dbms_resource_manager.create_plan_directive(plan=>'DAYPLAN',
                                            group_or_subplan=>'CRITICAL',shares=>3);
dbms_resource_manager.create_plan_directive(plan=>'DAYPLAN',
                                            group_or_subplan=>'ADHOC',shares=>1);
```

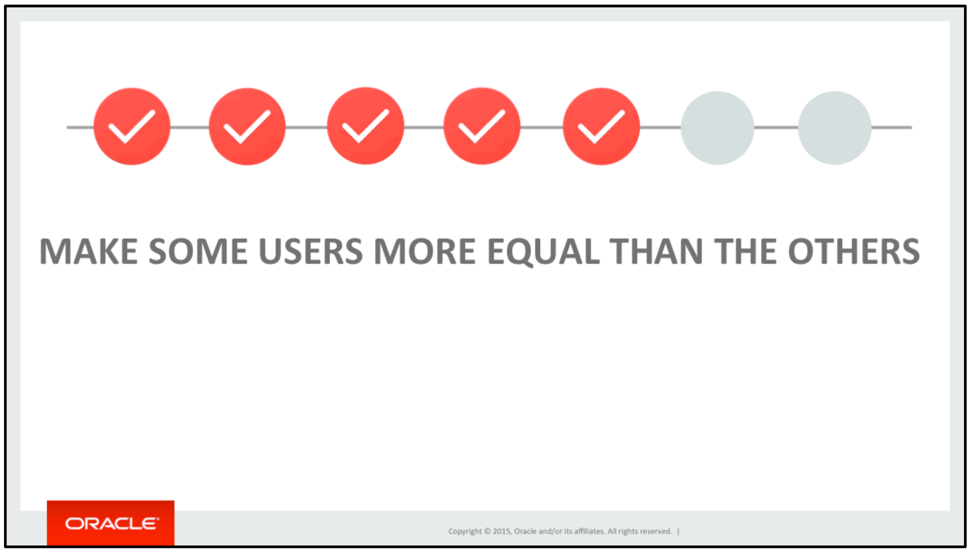


Dequeue Priority

Use DBRM shares to prioritize users



```
dbms_resource_manager.create_plan_directive(plan=>'DAYPLAN',
                                            group_or_subplan=>'CRITICAL',shares=>3);
dbms_resource_manager.create_plan_directive(plan=>'DAYPLAN',
                                            group_or_subplan=>'ADHOC',shares=>1);
```



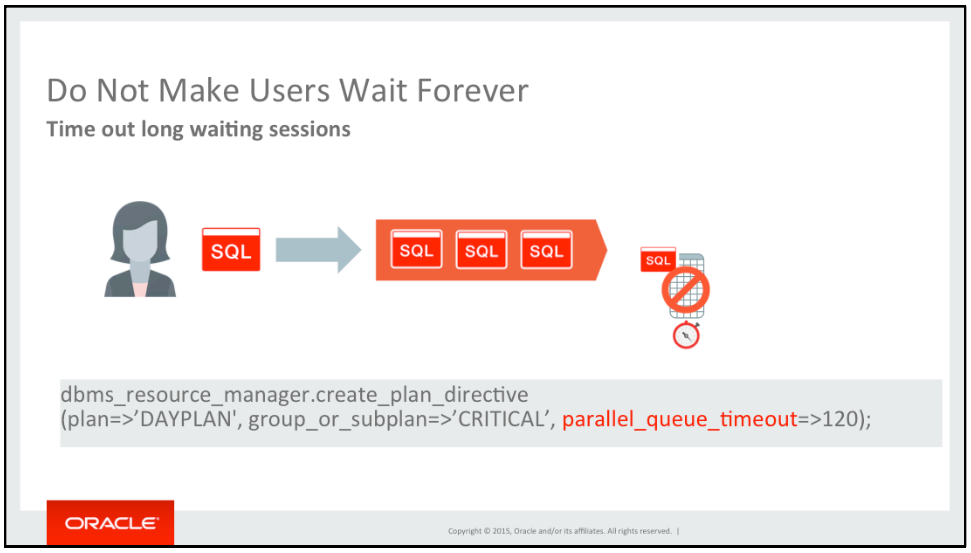
MAKE SOME USERS MORE EQUAL THAN THE OTHERS

ORACLE

EXIT

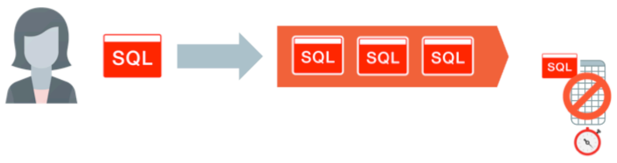
ALLOW A WAY OUT

ORACLE

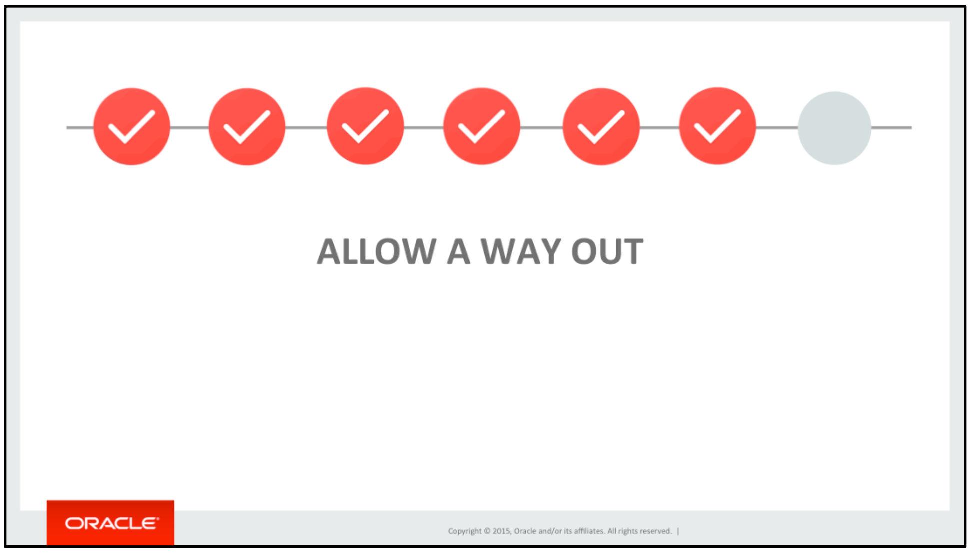


Do Not Make Users Wait Forever

Time out long waiting sessions



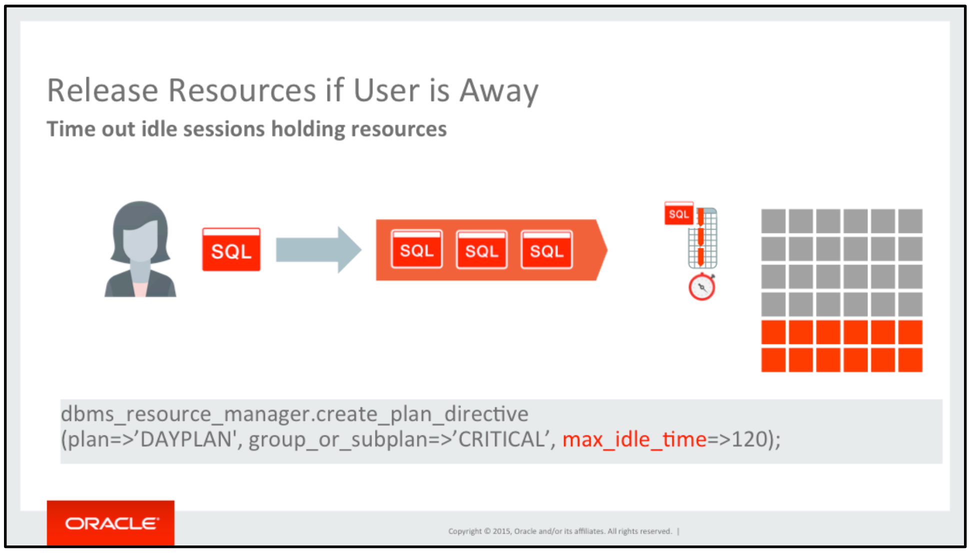
```
dbms_resource_manager.create_plan_directive
(plan=>’DAYPLAN', group_or_subplan=>’CRITICAL’, parallel_queue_timeout=>120);
```



ALLOW A WAY OUT

ORACLE

DO NOT ALLOW USERS TO HOLD RESOURCES FOREVER



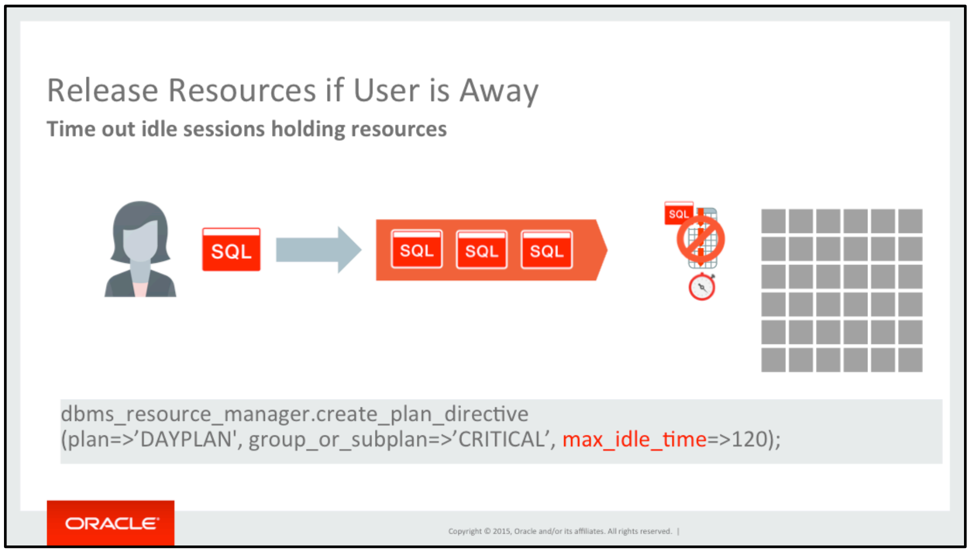
Release Resources if User is Away

Time out idle sessions holding resources

SQL
SQL
SQL
SQL
SQL

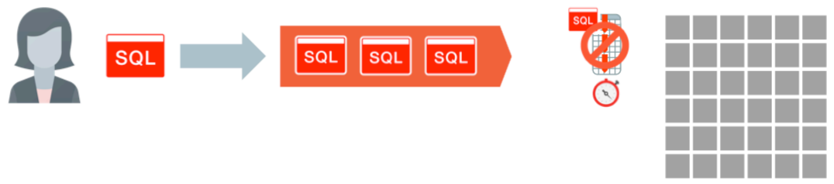
```
dbms_resource_manager.create_plan_directive
(plan=>'DAYPLAN', group_or_subplan=>'CRITICAL', max_idle_time=>120);
```

When a session using PX servers becomes idle before finishing the statement it keep holding those processes. This is a typical case when the user fetches only a few rows and waits without fetching the remaining rows. If the user keeps holding these processes for a long time other users may wait in the queue because of process shortage.

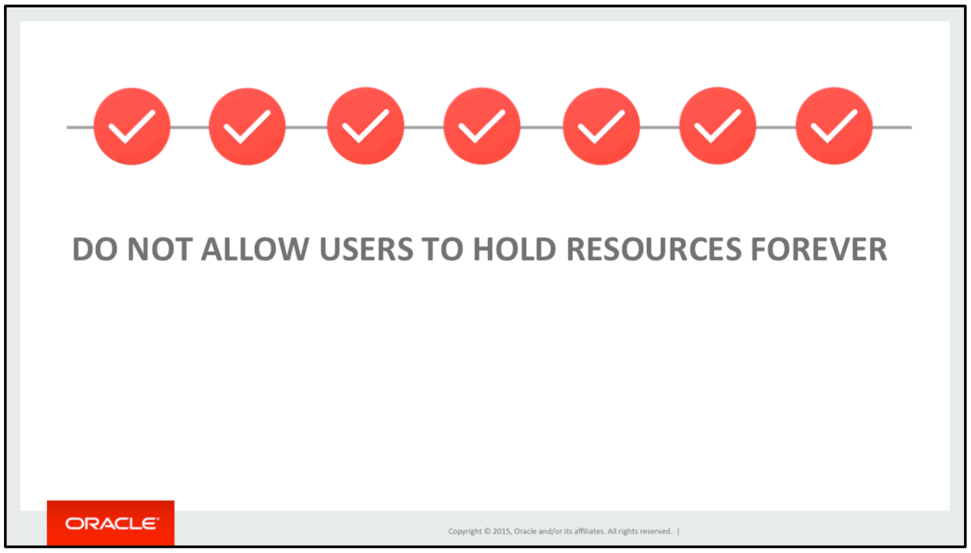


Release Resources if User is Away

Time out idle sessions holding resources

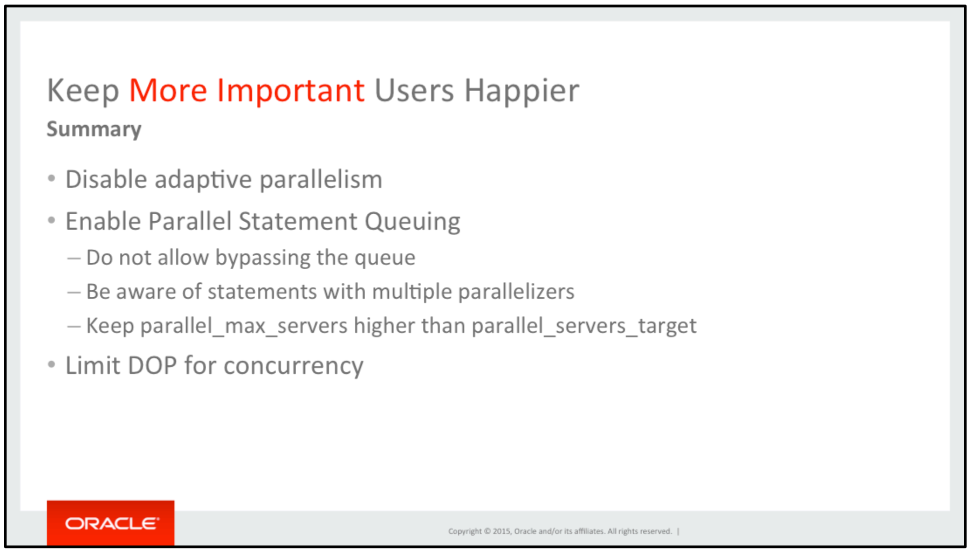


```
dbms_resource_manager.create_plan_directive
(plan=>'DAYPLAN', group_or_subplan=>'CRITICAL', max_idle_time=>120);
```



✓ ✓ ✓ ✓ ✓ ✓ ✓

DO NOT ALLOW USERS TO HOLD RESOURCES FOREVER

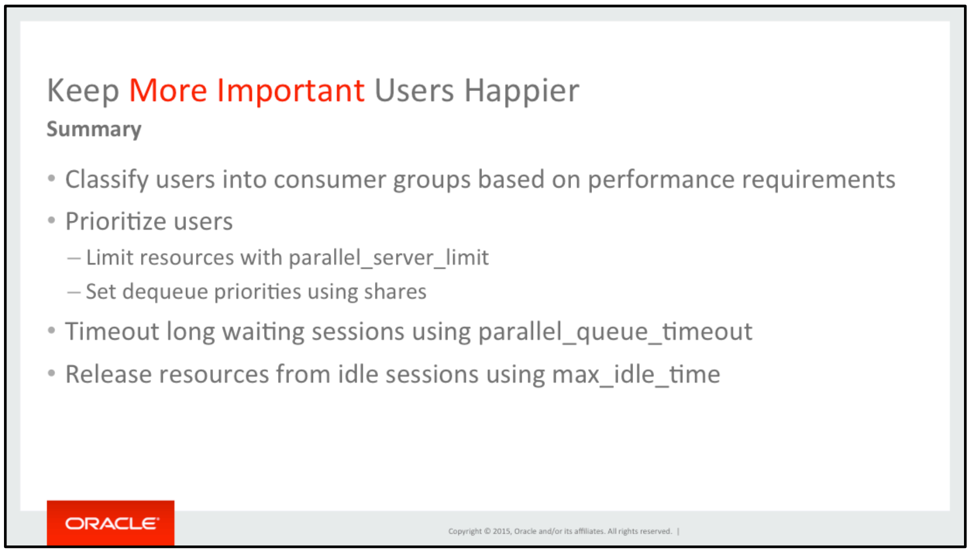


Keep More Important Users Happier

Summary

- Disable adaptive parallelism
- Enable Parallel Statement Queuing
  - Do not allow bypassing the queue
  - Be aware of statements with multiple parallelizers
  - Keep parallel_max_servers higher than parallel_servers_target
- Limit DOP for concurrency

ORACLE



Keep More Important Users Happier

Summary

- Classify users into consumer groups based on performance requirements
- Prioritize users
  - Limit resources with parallel_server_limit
  - Set dequeue priorities using shares
- Timeout long waiting sessions using parallel_queue_timeout
- Release resources from idle sessions using max_idle_time

ORACLE®