

Using Bind Variables

Chapter 11

GÜVEN GÜNEL (guven.gunel@ingbank.com.tr) has a non-transferable license to use this Student Guide.

Using Bind Variables

11

Using Bind Variables

ORACLE

Objectives

Objectives

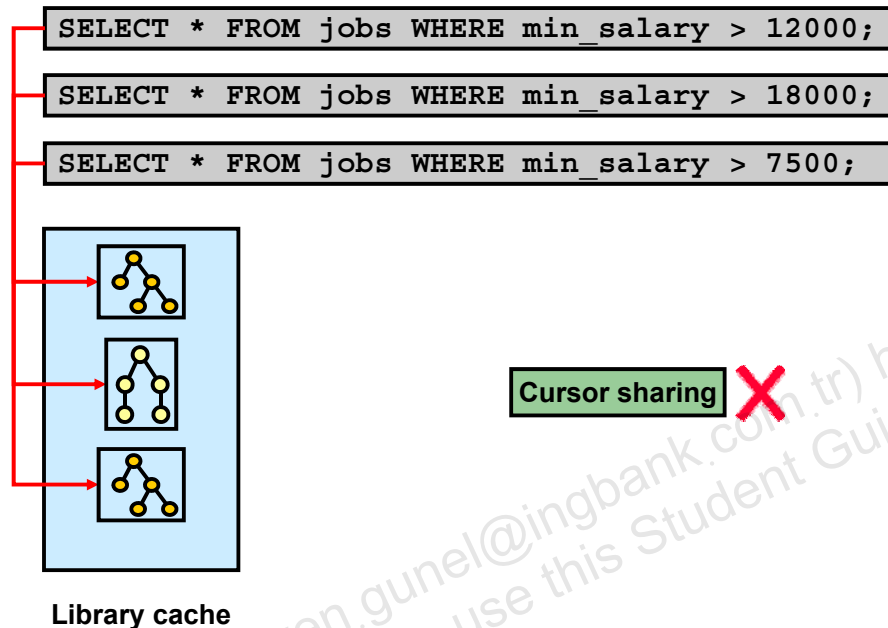
After completing this lesson, you should be able to:

- List the benefits of using bind variables
- Use bind peeking
- Use adaptive cursor sharing

ORACLE

Cursor Sharing and Different Literal Values

Cursor Sharing and Different Literal Values



Cursor Sharing and Different Literal Values

If your SQL statements use literal values for the `WHERE` clause conditions, there will be many versions of almost identical SQL stored in the library cache. For each different SQL statement, the optimizer must perform all the steps for processing a new SQL statement. This may also cause the library cache to fill up quickly because of all the different statements stored in it.

When coded this way, you are not taking advantage of cursor sharing. If the cursor is shared using a bind variable rather than a literal, there will be one shared cursor, with one execution plan.

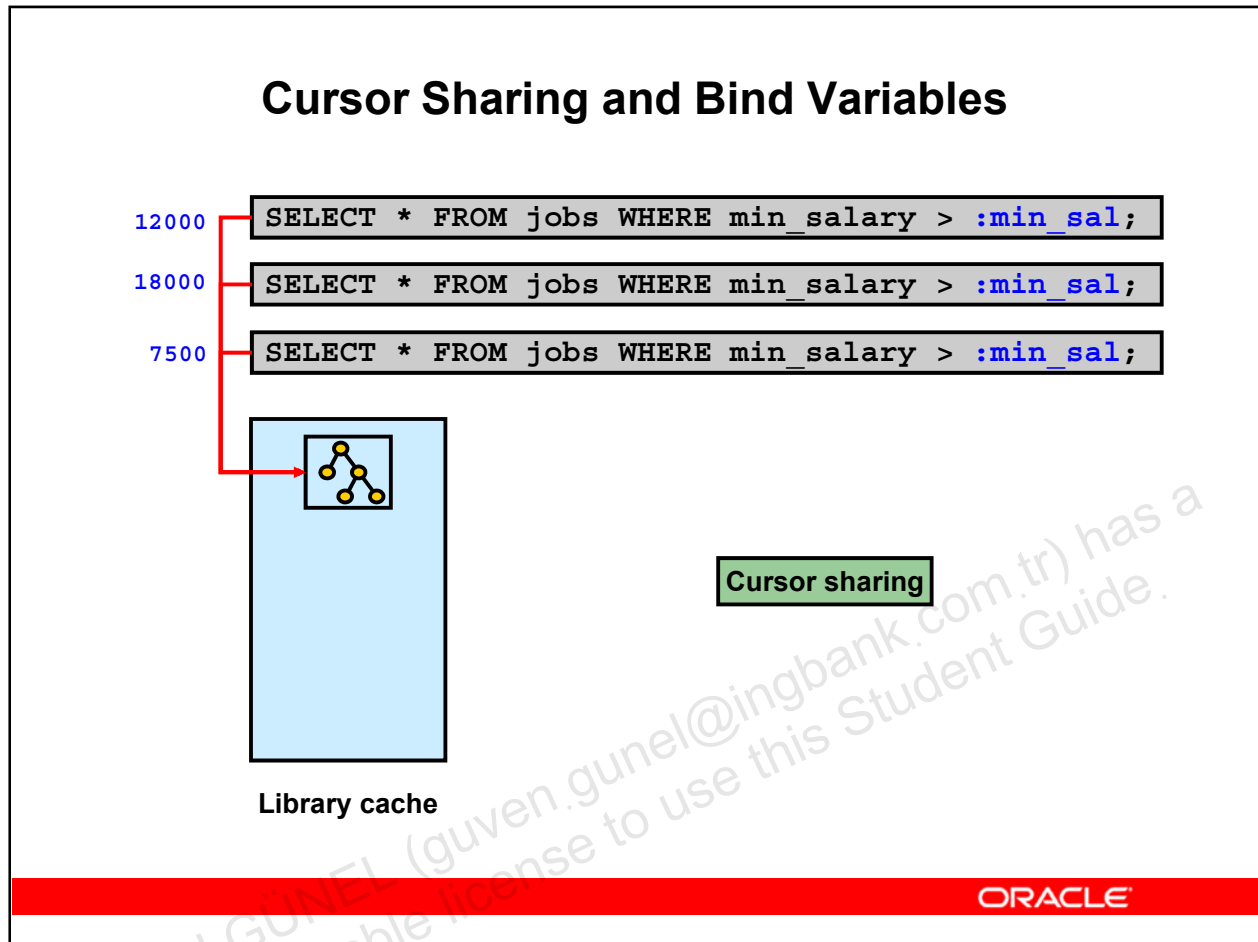
However, depending on the literal value provided, different execution plans might be generated by the optimizer. For example, there might be several `JOBS`, where `MIN_SALARY` is greater than 12000. Alternatively, there might be very few `JOBS` that have a `MIN_SALARY` greater than 18000. This difference in data distribution could justify the addition of an index so that different plans can be used depending on the value provided in the query. This is illustrated in the slide. As you can see, the first and third queries use the same execution plan, but the second query uses a different one.

From a performance perspective, it is good to have separate cursors. However, this is not very economic because you could have shared cursors for the first and last queries in this example.

Note: In the case of the example in the slide, `V$SQL.PLAN_HASH_VALUE` is identical for the first and third query.

GÜVEN GÜNEL (guven.gunel@ingbank.com.tr) has a
non-transferable license to use this Student Guide.

Cursor Sharing and Bind Variables



Cursor Sharing and Bind Variables

If, instead of issuing different statements for each literal, you use a bind variable, then that extra parse activity is eliminated (in theory). This is because the optimizer recognizes that the statement is already parsed and decides to reuse the same execution plan even though you specified different bind values the next time you execute the same statement.

In the example in the slide, the bind variable is called `min_sal`. It is to be compared with the `MIN_SALARY` column of the `JOBS` table. Instead of issuing three different statements, issue a single statement that uses a bind variable. At execution time, the same execution plan is used, the given value is substituted for the variable.

However, from a performance perspective, this is not the best situation because you get best performance two times out of three. On the other hand, this is very economic because you just need one shared cursor in the library cache to execute all the three statements.

Bind Variables in SQL*Plus

Bind Variables in SQL*Plus

```
SQL> variable job_id varchar2(10)
SQL> exec :job_id := 'SA_REP';

PL/SQL procedure successfully completed.

SQL> select count(*) from employees where job_id = :job_id;

  COUNT (*)
  -----
         30

SQL> exec :job_id := 'AD_VP';

PL/SQL procedure successfully completed.

SQL> select count(*) from employees where job_id = :job_id;

  COUNT (*)
  -----
         2
```

ORACLE

Bind Variables in SQL*Plus

Bind variables can be used in SQL*Plus sessions. In SQL*Plus, use the `VARIABLE` command to define a bind variable. Then, you can assign values to the variable by executing an assignment statement with the `EXEC [UTE]` command. Any references to that variable from then on use the value you assigned.

In the example in the slide, the first count is selected while `SA_REP` is assigned to the variable. The result is 30. Then, `AD_VP` is assigned to the variable, and the resulting count is 2.

Bind Variables in Enterprise Manager

Bind Variables in Enterprise Manager

SQL Worksheet : orcl.us.oracle.com

Enter a SQL statement to execute. If there are multiple statements, the location of the cursor or a highlighted statement determines which will be executed. Statements should be separated with blank lines.

SQL Commands

select count(*) from hr.employees where salary between :low_sal and :hi_sal

☒ Use bind variables for execution

Remove Move Up Move Down Add 5 Rows Remove All

Select Value Data Type

<input checked="" type="radio"/> 5000	NUMBER
<input type="radio"/> 10000	NUMBER
<input type="radio"/>	STRING
<input type="radio"/>	STRING
<input type="radio"/>	STRING

☐ Auto commit
☐ Allow only SELECT statements
Execute

Last Executed SQL

```
select count(*)
from hr.employees
where salary between :low_sal and :hi_sal
```

Last Execution Details

SQL Repair Advisor SQL Details Schedule SQL Tuning Advisor

Results Statistics Plan

Execution Time (seconds) 0.0010

EDUNT(*)

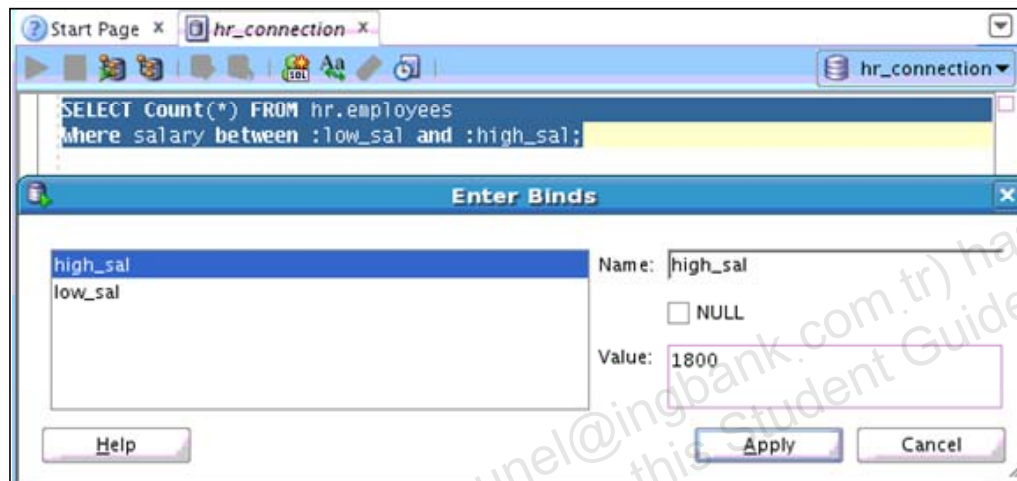
43

Bind Variables in Enterprise Manager

On the SQL Worksheet page of Enterprise Manager (see the SQL Worksheet link in the Related Links region of the Database Home page), you can specify that a SQL statement should use bind variables. You can do this by selecting the "Use bind variables for execution" check box. When you select that, several fields are generated, where you can enter bind variable values. Refer to these values in the SQL statement using variable names that begin with a colon. The order in which variables are referred to defines which variable gets which value. The first variable referred to gets the first value, the second variable gets the second value, and so on. If you change the order in which variables are referenced in the statement, you may need to change the value list to match that order.

Bind Variables in SQL Developer

Bind Variables in SQL Developer

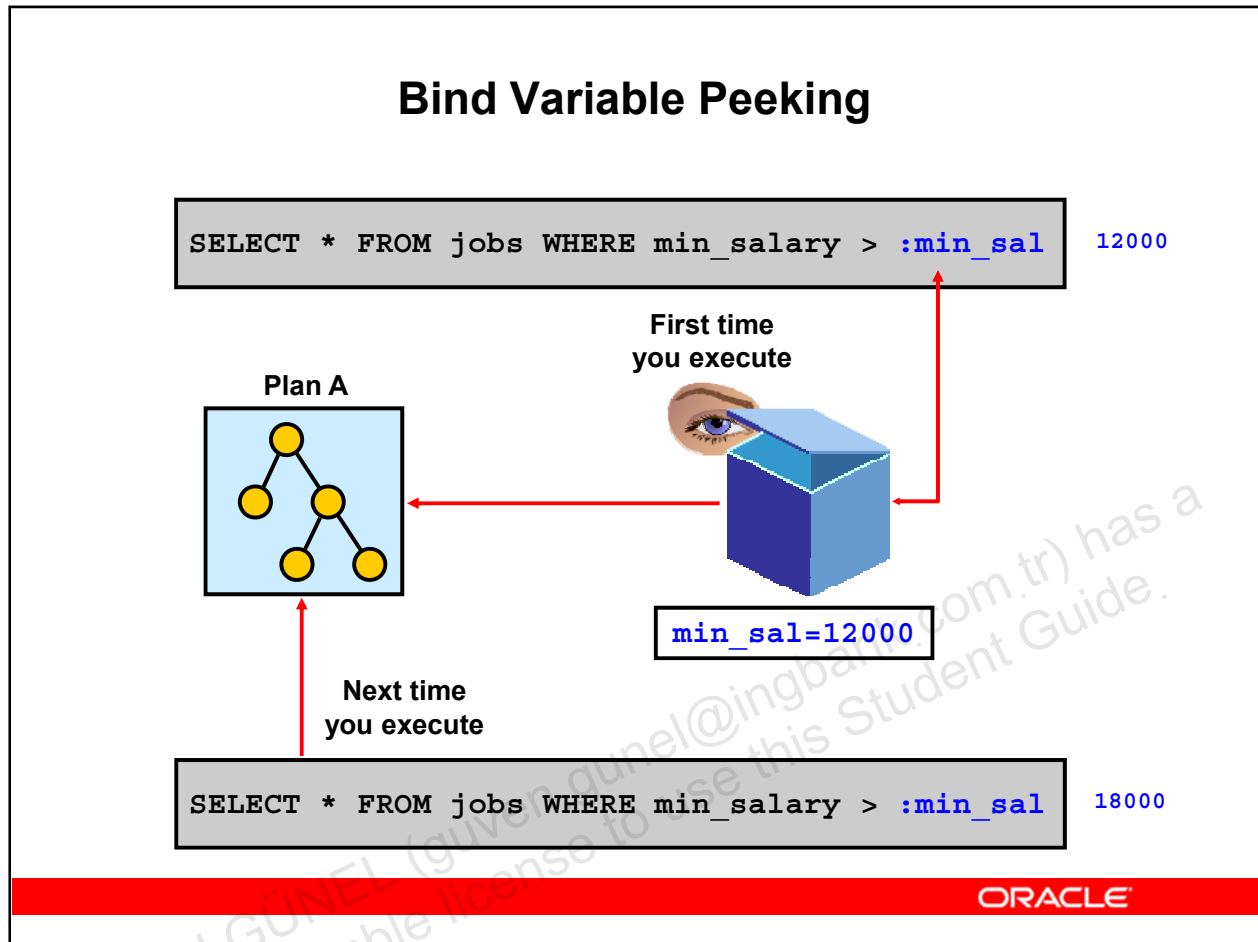


ORACLE

Bind Variables in SQL Developer

On the SQL Worksheet pane of SQL, you can specify that a SQL statement that uses bind variables. When you execute the statement, the Enter Binds dialog appears where you can enter bind variable values. Refer to these values in the SQL statement using variable names that begin with a colon. Select each bind variable in turn to enter a value for that variable.

Bind Variable Peeking

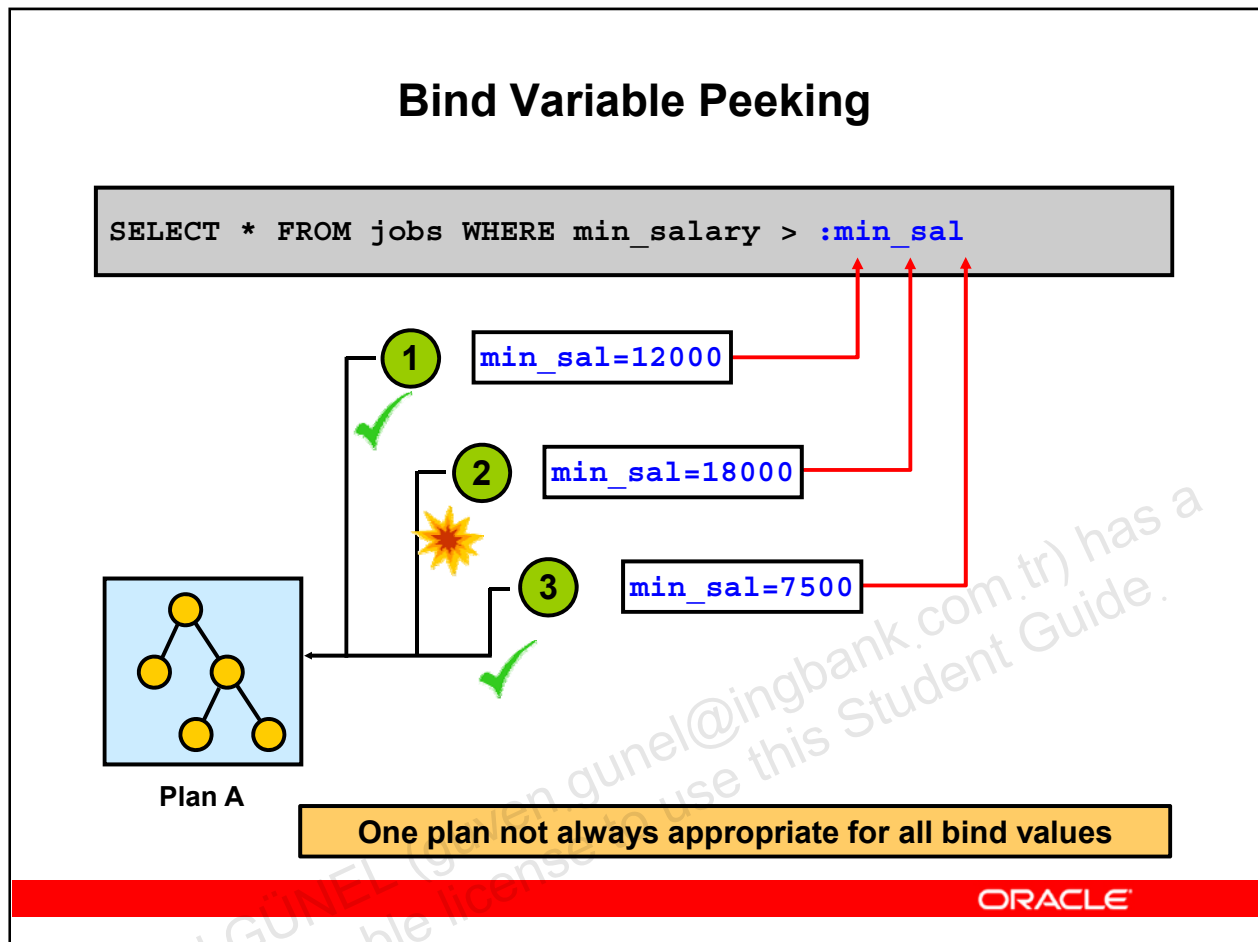


Bind Variable Peeking

When literals are used in a query, those literal values can be used by the optimizer to decide on the best plan. However, when bind variables are used, the optimizer still needs to select the best plan based on the values of the conditions in the query, but cannot see those values readily in the SQL text. That means, as a SQL statement is parsed, the system needs to be able to see the value of the bind variables, to ensure that a good plan that would suit those values is selected. The optimizer does this by *peeking* at the value in the bind variable. When the SQL statement is hard parsed, the optimizer evaluates the value for each bind variable, and uses that as input in determining the best plan. After the execution is determined the first time you parsed the query, it is reused when you execute the same statement regardless of the bind values used.

This feature was introduced in Oracle9i Database, Release 2. Oracle Database 11g changes this behavior.

Bind Variable Peeking



Bind Variable Peeking (continued)

Under some conditions, bind variable peeking can cause the optimizer to select the suboptimal plan. This occurs because the first value of the bind variable is used to determine the plan for all subsequent executions of the query. Therefore, even though subsequent executions provide different bind values, the same plan is used. It is possible that a different plan would be better for executions that have different bind variable values. An example is where the selectivity of a particular index varies extremely depending on the column value. For low selectivity, a full table scan may be faster. For high selectivity, an index range scan may be more appropriate. As shown in the slide, plan A may be good for the first and third values of `min_sal`, but it may not be the best for the second one. Suppose there are very few `MIN_SALARY` values that are above 18000, and plan A is a full table scan. It is probable that a full table scan is not a good plan for the second execution, in that case.

So bind variables are beneficial in that they cause more cursor sharing to happen, and thus reduce parsing of SQL. But, as in this case, it is possible that they cause a suboptimal plan to be chosen for some of the bind variable values. This is a good reason for not using bind variables for decision support system (DSS) environments, where the parsing of the query is a very small percentage of the work done when submitting a query. The parsing may take fractions of a second, but the execution may take minutes or hours. To execute with a slower plan is not worth the savings gained in parse time.

Cursor Sharing Enhancements

Cursor Sharing Enhancements

- Oracle8i introduced the possibility of sharing SQL statements that differ only in literal values.
- Oracle9i extends this feature by limiting it to similar statements only, instead of forcing it.
- Similar: Regardless of the literal value, same execution plan

```
SQL> SELECT * FROM employees  
2 WHERE employee_id = 153;
```

- Not similar: Possible different execution plans for different literal values

```
SQL> SELECT * FROM employees  
2 WHERE department_id = 50;
```

ORACLE

Cursor Sharing Enhancements

Oracle8i introduced the possibility of sharing SQL statements that differ only in literal values. Rather than developing an execution plan each time the same statement—with a different literal value—is executed, the optimizer generates a common execution plan used for all subsequent executions of the statement.

Because only one execution plan is used instead of potential different ones, this feature should be tested against your applications before you decide to enable it or not. That is why Oracle9i extends this feature by sharing only statements considered as similar. That is, only when the optimizer has the guarantee that the execution plan is independent of the literal value used. For example, consider a query, where `EMPLOYEE_ID` is the primary key:

```
SQL> SELECT * FROM employees WHERE employee_id = 153;
```

The substitution of any value would produce the same execution plan. It would, therefore, be safe for the optimizer to generate only one plan for different occurrences of the same statement executed with different literal values.

On the other hand, assume that the same `EMPLOYEES` table has a wide range of values in its `DEPARTMENT_ID` column. For example, department 50 could contain over one third of all employees and department 70 could contain just one or two.

See the two queries:

```
SQL> SELECT * FROM employees WHERE department_id = 50;
```

Copyright © 2010, Oracle and/or its affiliates. All rights reserved.

```
SQL> SELECT * FROM employees WHERE department_id = 70;
```

Using only one execution plan for sharing the same cursor would not be safe if you have histogram statistics (and there is skew in the data) on the `DEPARTMENT_ID` column. In this case, depending on which statement was executed first, the execution plan could contain a full table (or fast full index) scan, or it could use a simple index range scan.

GÜVEN GÜNEL (guven.gunel@ingbank.com.tr) has a non-transferable license to use this Student Guide.

The CURSOR_SHARING Parameter

The CURSOR_SHARING Parameter

- The CURSOR_SHARING parameter values:
 - FORCE
 - EXACT (default)
 - SIMILAR
- CURSOR_SHARING can be changed using:
 - ALTER SYSTEM
 - ALTER SESSION
 - Initialization parameter files
- The CURSOR_SHARING_EXACT hint

ORACLE

The CURSOR_SHARING Parameter

The value of the CURSOR_SHARING initialization parameter determines how the optimizer processes statements with bind variables:

- EXACT: Literal replacement disabled completely
- FORCE: Causes sharing for all literals
- SIMILAR: Causes sharing for safe literals only

In earlier releases, you could select only the EXACT or the FORCE option. Setting the value to SIMILAR causes the optimizer to examine the statement to ensure that replacement occurs only for safe literals. It can then use information about the nature of any available index (unique or nonunique) and statistics collected on the index or underlying table, including histograms.

The value of CURSOR_SHARING in the initialization file can be overridden with an ALTER SYSTEM SET CURSOR_SHARING or an ALTER SESSION SET CURSOR_SHARING command.

The CURSOR_SHARING_EXACT hint causes the system to execute the SQL statement without any attempt to replace literals by bind variables.

Forcing Cursor Sharing: Example

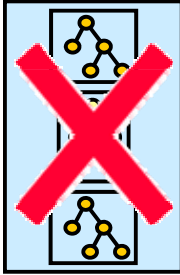
Forcing Cursor Sharing: Example

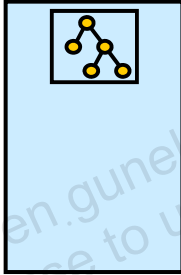
```
SQL> alter session set cursor_sharing = FORCE;
```

```
SELECT * FROM jobs WHERE min_salary > 12000;  
SELECT * FROM jobs WHERE min_salary > 18000;  
SELECT * FROM jobs WHERE min_salary > 7500;
```

↓

```
SELECT * FROM jobs WHERE min_salary > : "SYS_B_0"
```





↑
System-generated
bind variable

ORACLE

Forcing Cursor Sharing: Example

Because you forced cursor sharing with the `ALTER SESSION` command, all your queries that differ only with literal values are automatically rewritten to use the same system-generated bind variable called `SYS_B_0` in the example in the slide. As a result, you end up with only one child cursor instead of three.

Note: Adaptive cursor sharing may also apply, and might generate a second child cursor in this case.

Adaptive Cursor Sharing: Overview

Adaptive Cursor Sharing: Overview

Adaptive cursor sharing:

- Allows for intelligent cursor sharing for statements that use bind variables
- Is used to compromise between cursor sharing and optimization
- Has the following benefits:
 - Automatically detects when different executions would benefit from different execution plans
 - Limits the number of generated child cursors to a minimum
 - Provides an automated mechanism that cannot be turned off

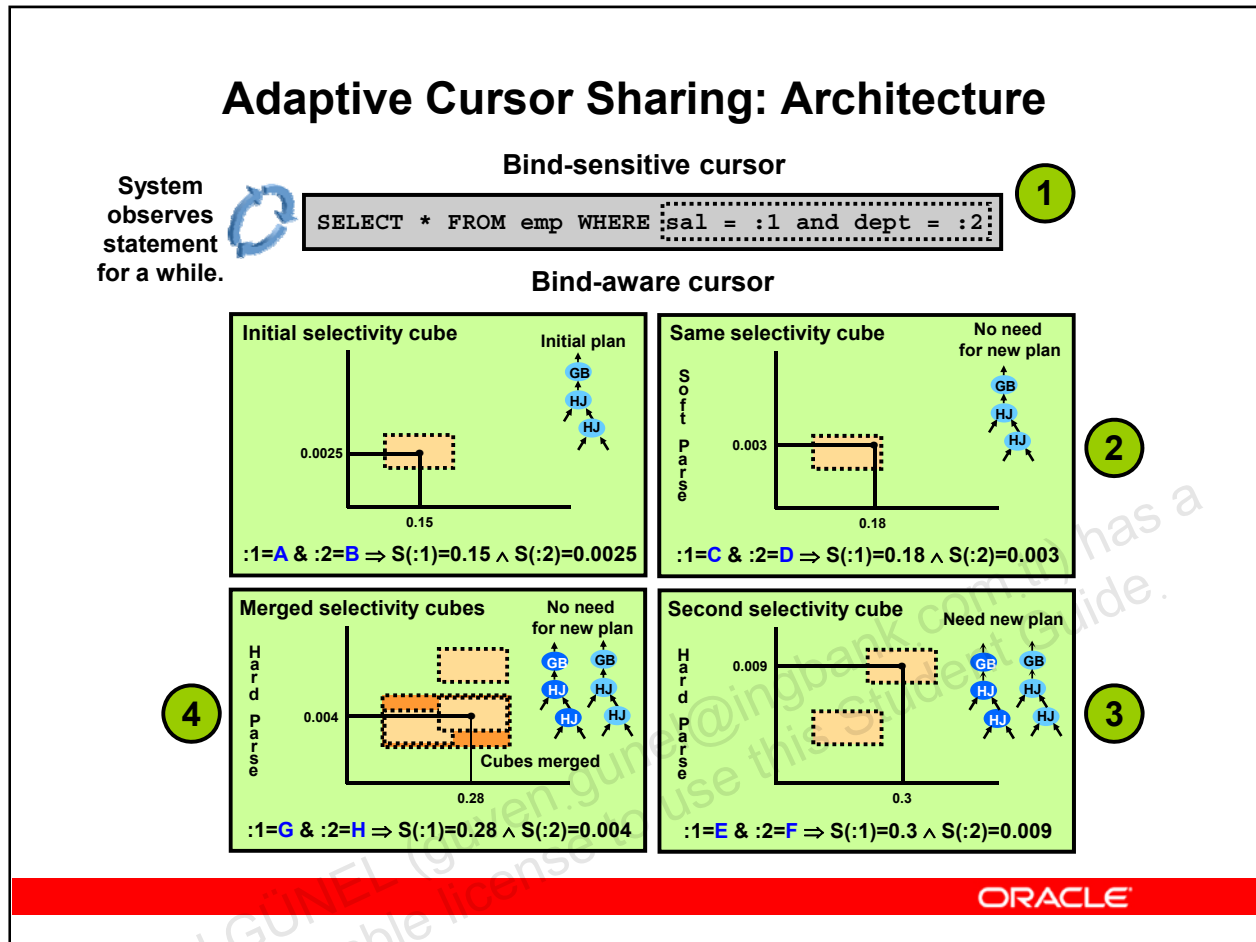
ORACLE

Adaptive Cursor Sharing: Overview

Bind variables were designed to allow the Oracle Database to share a single cursor for multiple SQL statements to reduce the amount of shared memory used to parse SQL statements. However, cursor sharing and SQL optimization are conflicting goals. Writing a SQL statement with literals provides more information for the optimizer and naturally leads to better execution plans, while increasing memory and CPU overhead caused by excessive hard parses. Oracle9i Database was the first attempt to introduce a compromise solution by allowing similar SQL statements using different literal values to be shared. For statements using bind variables, Oracle9i also introduced the concept of bind peeking. To benefit from bind peeking, it is assumed that cursor sharing is intended and that different invocations of the statement are supposed to use the same execution plan. If different invocations of the statement would significantly benefit from different execution plans, bind peeking is of no use in generating good execution plans.

To address this issue as much as possible, Oracle Database 11g introduces adaptive cursor sharing. This feature is a more sophisticated strategy designed to not share the cursor blindly, but generate multiple plans per SQL statement with bind variables if the benefit of using multiple execution plans outweighs the parse time and memory usage overhead. However, because the purpose of using bind variables is to share cursors in memory, a compromise must be found regarding the number of child cursors that need to be generated.

Adaptive Cursor Sharing: Architecture



Adaptive Cursor Sharing: Architecture

When you use adaptive cursor sharing, the following steps take place in the scenario illustrated in the slide:

1. The cursor starts its life with a hard parse, as usual. If bind peeking takes place, and a histogram is used to compute selectivity of the predicate containing the bind variable, then the cursor is marked as a bind-sensitive cursor. In addition, some information is stored about the predicate containing the bind variables, including the predicate selectivity. In the slide example, the predicate selectivity that would be stored is a cube centered around (0.15,0.0025). Because of the initial hard parse, an initial execution plan is determined using the peeked binds. After the cursor is executed, the bind values and the execution statistics of the cursor are stored in that cursor.

During the next execution of the statement when a new set of bind values is used, the system performs a usual soft parse, and finds the matching cursor for execution. At the end of execution, execution statistics are compared with the ones currently stored in the cursor. The system then observes the pattern of the statistics over all the previous runs (see V\$SQL_CS_... views in the slide that follows) and decides whether or not to mark the cursor as bind aware.

2. On the next soft parse of this query, if the cursor is now bind aware, bind-aware cursor matching is used. Suppose the selectivity of the predicate with the new set of bind values is now (0.18,0.003). Because selectivity is used as part of bind-aware cursor

matching, and because the selectivity is within an existing cube, the statement uses the existing child cursor's execution plan to run.

3. On the next soft parse of this query, suppose that the selectivity of the predicate with the new set of bind values is now (0.3,0.009). Because that selectivity is not within an existing cube, no child cursor match is found. So the system does a hard parse, which generates a new child cursor with a second execution plan in that case. In addition, the new selectivity cube is stored as part of the new child cursor. After the new child cursor executes, the system stores the bind values and execution statistics in the cursor.
4. On the next soft parse of this query, suppose the selectivity of the predicate with the new set of bind values is now (0.28,0.004). Because that selectivity is not within one of the existing cubes, the system does a hard parse. Suppose that this time, the hard parse generates the same execution plan as the first one. Because the plan is the same as the first child cursor, both child cursors are merged. That is, both cubes are merged into a new bigger cube, and one of the child cursors is deleted. The next time there is a soft parse, if the selectivity falls within the new cube, the child cursor matches.

GÜVEN GÜNEL (guven.gunel@ingbank.com.tr) has a non-transferable license to use this Student Guide.

Adaptive Cursor Sharing: Views

Adaptive Cursor Sharing: Views

The following views provide information about adaptive cursor sharing usage:

V\$SQL	Two new columns show whether a cursor is bind sensitive or bind aware.
V\$SQL_CS_HISTOGRAM	Shows the distribution of the execution count across the execution history histogram
V\$SQL_CS_SELECTIVITY	Shows the selectivity cubes stored for every predicate containing a bind variable and whose selectivity is used in the cursor sharing checks
V\$SQL_CS_STATISTICS	Shows execution statistics of a cursor using different bind sets

ORACLE

Adaptive Cursor Sharing: Views

These views determine whether a query is bind aware or not, and is handled automatically, without any user input. However, information about what goes on is exposed through v\$ views so that you can diagnose problems, if any. New columns have been added to V\$SQL:

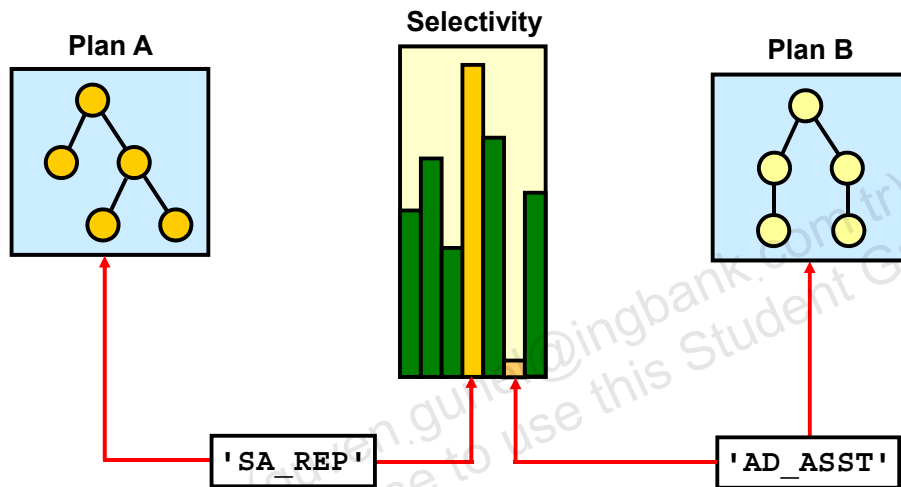
- **IS_BIND_SENSITIVE:** Indicates if a cursor is bind sensitive; value YES | NO. A query for which the optimizer peeked at bind variable values when computing predicate selectivities and where a change in a bind variable value may lead to a different plan is called *bind sensitive*.
- **IS_BIND_AWARE:** Indicates if a cursor is bind aware; value YES | NO. A cursor in the cursor cache that has been marked to use bind-aware cursor sharing is called *bind aware*.
- **V\$SQL_CS_HISTOGRAM:** Shows the distribution of the execution count across a three-bucket execution history histogram.
- **V\$SQL_CS_SELECTIVITY:** Shows the selectivity cubes or ranges stored in a cursor for every predicate containing a bind variable and whose selectivity is used in the cursor sharing checks. It contains the text of the predicates and the selectivity range low and high values.

- `V$SQL_CS_STATISTICS`: Adaptive cursor sharing monitors execution of a query and collects information about it for a while, and uses this information to decide whether to switch to using bind-aware cursor sharing for the query. This view summarizes the information that it collects to make this decision. For a sample of executions, it keeps track of the rows processed, buffer gets, and CPU time. The `PEEKED` column has the value `YES` if the bind set was used to build the cursor, and `NO` otherwise.

GÜVEN GÜNEL (guven.gunel@ingbank.com.tr) has a non-transferable license to use this Student Guide.

Adaptive Cursor Sharing: Example

```
SQL> variable job varchar2(6)
SQL> exec :job := 'AD_ASST'
SQL> select count(*), max(salary) from emp where job_id=:job;
```



Adaptive Cursor Sharing: Example

Consider the data in the slide. There are histogram statistics on the `JOB_ID` column, showing that there are many thousands times more occurrences of `SA_REP` than `AD_ASST`. In this case, if literals were used instead of a bind variable, the query optimizer would see that the `AD_ASST` value occurs in less than 1% of the rows, whereas the `SA_REP` value occurs in approximately a third of the rows. If the table has over a million rows in it, the execution plans are different for each of these values' queries. The `AD_ASST` query results in an index range scan because there are so few rows with that value. The `SA_REP` query results in a full table scan because so many of the rows have that value, it is more efficient to read the entire table. But, as it is, using a bind variable causes the same execution plan to be used for both of the values, at first. So, even though there exist different and better plans for each of these values, they use the same plan.

After several executions of this query using a bind variable, the system considers the query bind aware, at which point it changes the plan based on the bound value. This means the best plan is used for the query, based on the bind variable value.

Interacting with Adaptive Cursor Sharing

- **CURSOR_SHARING:**
 - If `CURSOR_SHARING <> EXACT`, statements containing literals may be rewritten using bind variables.
 - If statements are rewritten, adaptive cursor sharing may apply to them.
- **SQL Plan Management (SPM):**
 - If `OPTIMIZER_CAPTURE_SQL_PLAN_BASELINES` is set to `TRUE`, only the first generated plan is used.
 - As a workaround, set this parameter to `FALSE`, and run your application until all plans are loaded in the cursor cache.
 - Manually load the cursor cache into the corresponding plan baseline.

ORACLE

Interacting with Adaptive Cursor Sharing

- Adaptive cursor sharing is independent of the `CURSOR_SHARING` parameter. The setting of this parameter determines whether literals are replaced by the system-generated bind variables. If they are, adaptive cursor sharing behaves just as it would if the user supplied binds to begin with.
- When using the SPM automatic plan capture, the first plan captured for a SQL statement with bind variables is marked as the corresponding SQL plan baseline. If another plan is found for that same SQL statement (which maybe the case with adaptive cursor sharing), it is added to the SQL statements plan history and marked for verification. It will not be used immediately. So even though adaptive cursor sharing has come up with a new plan based on a new set of bind values, SPM does not let it be used until the plan has been verified. Thus reverting to 10g behavior, only the plan generated based on the first set of bind values is used by all subsequent executions of the statement. One possible workaround is to run the system for some time with automatic plan capture set to `False`, and after the cursor cache has been populated with all the plans a SQL statement with bind has, load the entire plan directly from the cursor cache into the corresponding SQL plan baseline. By doing this, all the plans for a single SQL statement are marked as SQL baseline plans by default. Refer to the lesson titled "SQL Plan Management" for more information.

Quiz

Quiz

Which three statements are true about applications that are coded with literals in the SQL statements rather than bind variables?

- a. More shared pool space is required for cursors.
- b. Less shared pool space is required for cursors.
- c. Histograms are used if available.
- d. Histograms are not used.
- e. No parsing is required for literal values.
- f. Every different literal value requires parsing.

ORACLE

Answer: a, c, f

Quiz

Quiz

The `CURSOR_SHARING` parameter should be set to _____ for systems with large tables and long-running queries such as a data warehouse.

- a. Similar
- b. Force
- c. Exact
- d. Literal
- e. True
- f. False

ORACLE

Answer: c

Quiz

Quiz

Adaptive Cursor Sharing can be turned off by setting the `CURSOR_SHARING` parameter to `FALSE`.

- a. True
- b. False

ORACLE

Answer: b

Summary

Summary

In this lesson, you should have learned how to:

- List the benefits of using bind variables
- Use bind peeking
- Use adaptive cursor sharing

ORACLE

Practice 11: Overview

Practice 11: Overview

This practice covers the following topics:

- Using adaptive cursor sharing and bind peeking
- Using the `CURSOR_SHARING` initialization parameter

ORACLE

GÜVEN GÜNEL (guven.gunel@ingbank.com.tr) has a
non-transferable license to use this Student Guide.