20

Improving Real-World Performance Through Cursor Sharing

Cursor sharing can improve database application performance by orders of magnitude.

Overview of Cursor Sharing

Oracle Database can share cursors, which are pointers to private SQL areas in the shared pool.

About Cursors

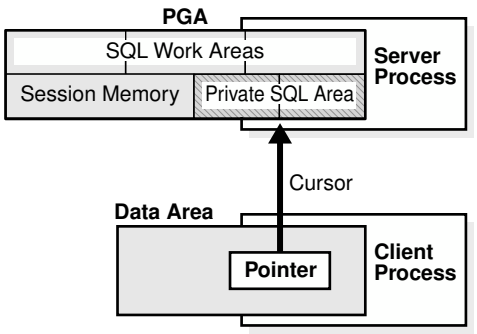
A **private SQL area** holds information about a parsed SQL statement and other session-specific information for processing.

When a server process executes SQL or PL/SQL code, the process uses the private SQL area to store bind variable values, query execution state information, and query execution work areas. The private SQL areas for each execution of a statement are not shared and may contain different values and data.

A cursor is a name or handle to a specific private SQL area. The cursor contains session-specific state information such as bind variable values and result sets.

As shown in the following graphic, you can think of a cursor as a pointer on the client side and as a state on the server side. Because cursors are closely associated with private SQL areas, the terms are sometimes used interchangeably.

Figure 20-1 Cursor



Private and Shared SQL Areas

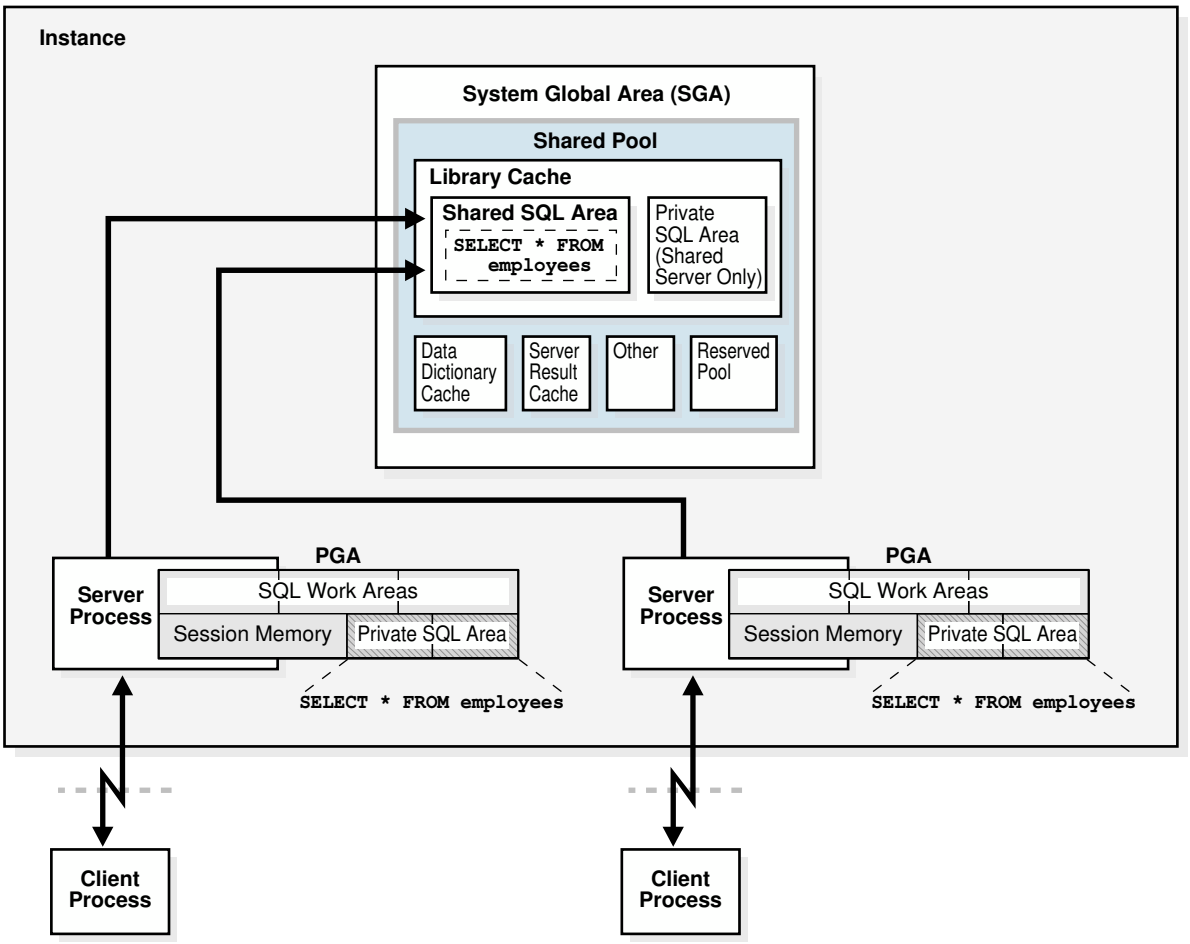
A cursor in the private SQL area points to a **shared SQL area** in the library cache.

Unlike the private SQL area, which contains session state information, the shared SQL area contains the parse tree and execution plan for the statement. For example, an execution of `SELECT * FROM employees` has a plan and parse tree stored in one shared SQL area. An

execution of `SELECT * FROM departments`, which differs both syntactically and semantically, has a plan and parse tree stored in a separate shared SQL area.

Multiple private SQL areas in the same or different sessions can reference a single shared SQL area, a phenomenon known as **cursor sharing**. For example, an execution of `SELECT * FROM employees` in one session and an execution of the `SELECT * FROM employees` (accessing the same table) in a different session can use the same parse tree and plan. A shared SQL area that is accessed by multiple statements is known as a shared cursor.

Figure 20-2 Cursor Sharing



Oracle Database automatically determines whether the SQL statement or PL/SQL block being issued is textually identical to another statement currently in the library cache, using the following steps:

1. The text of the statement is hashed.
2. The database looks for a matching hash value for an existing SQL statement in the shared pool. The following options are possible:
   - No matching hash value exists.

     In this case, the SQL statement does not currently exist in the shared pool, so the database performs a hard parse. This ends the shared pool check.
   - A matching hash value exists.

In this case, the database proceeds to the next step, which is a text match.

3. The database compares the text of the matched statement to the text of the hashed statement to determine whether they are identical. The following options are possible:

   - The textual match fails.

     In this case, the text match process stops, resulting in a hard parse.

   - The textual match succeeds.

     In this case, the database proceeds to the next step: determining whether the SQL can share an existing parent cursor.

     For a textual match to occur, the text of the SQL statements or PL/SQL blocks must be character-for-character identical, including spaces, case, and comments. For example, the following statements cannot use the same shared SQL area:

     ```
     SELECT * FROM employees;
     SELECT * FROM Employees;
     SELECT *  FROM employees;
     ```

     Usually, SQL statements that differ only in literals cannot use the same shared SQL area. For example, the following statements do not resolve to the same SQL area:

     ```
     SELECT count(1) FROM employees WHERE manager_id = 121;
     SELECT count(1) FROM employees WHERE manager_id = 247;
     ```

     The only exception to this rule is when the parameter `CURSOR_SHARING` has been set to `FORCE`, in which case similar statements can share SQL areas.

> **See Also:**
>
> - "Parent and Child Cursors"
> - "Do Not Use CURSOR_SHARING = FORCE as a Permanent Fix" to learn about the costs involved in using `CURSOR_SHARING`
> - *Oracle Database Reference* to learn more about the `CURSOR_SHARING` initialization parameter

Parent and Child Cursors

Every parsed SQL statement has a parent cursor and one or more child cursors.

The parent cursor stores the text of the SQL statement. If the text of two statements is identical, then the statements share the same parent cursor. If the text is different, however, then the database creates a separate parent cursor.

Example 20-1 Parent Cursors

In this example, the first two statements are syntactically different (the letter “c” is lowercase in the first statement and uppercase in the second statement), but semantically identical. Because of the syntactic difference, these statements have different parent cursors. The third statement is syntactically identical to the first statement (lowercase “c”), but semantically

different because it refers to a `customers` table in a different schema. Because of the syntactic identity, the third statement can share a parent cursor with the first statement.

```
SQL> CONNECT oe@inst1
Enter password: *******
Connected.
SQL> SELECT COUNT(*) FROM customers;

  COUNT(*)
----------
       319

SQL> SELECT COUNT(*) FROM Customers;

  COUNT(*)
----------
       319

SQL> CONNECT sh@inst1
Enter password:  *******
Connected.
SQL> SELECT COUNT(*) FROM customers;

  COUNT(*)
----------
    155500
```

The following query of `V$SQL` indicates the two parents. The statement with the SQL ID of `8h916vv2yw400`, which is the lowercase “c” version of the statement, has one parent cursor and two child cursors: child 0 and child 1. The statement with the SQL ID of `5rn2uxjtpz0wd`, which is the uppercase “c” version of the statement, has a different parent cursor and only one child cursor: child 0.

```
SQL> CONNECT SYSTEM@inst1
Enter password:  *******
Connected.

SQL> COL SQL_TEXT FORMAT a30
SQL> COL CHILD# FORMAT 99999
SQL> COL EXEC FORMAT 9999
SQL> COL SCHEMA FORMAT a6
SQL> SELECT SQL_ID, PARSING_SCHEMA_NAME AS SCHEMA, SQL_TEXT,
  2  CHILD_NUMBER AS CHILD#, EXECUTIONS AS EXEC FROM V$SQL
  3  WHERE SQL_TEXT LIKE '%ustom%' AND SQL_TEXT NOT LIKE '%SQL_TEXT%' ORDER
BY SQL_ID;

SQL_ID        SCHEMA SQL_TEXT                       CHILD# EXEC
------------- ------ ------------------------------ ------ -----
5rn2uxjtpz0wd OE     SELECT COUNT(*) FROM Customers      0     1
8h916vv2yw400 OE     SELECT COUNT(*) FROM customers      0     1
8h916vv2yw400 SH     SELECT COUNT(*) FROM customers      1     1
```

Parent Cursors and V$SQLAREA

The `V$SQLAREA` view contains one row for every parent cursor.

In the following example, a query of `V$SQLAREA` shows two parent cursors, each identified with a different `SQL_ID`. The `VERSION_COUNT` indicates the number of child cursors.

```
COL SQL_TEXT FORMAT a30
SELECT SQL_TEXT, SQL_ID, VERSION_COUNT, HASH_VALUE
FROM   V$SQLAREA
WHERE  SQL_TEXT LIKE '%mployee%'
AND    SQL_TEXT NOT LIKE '%SQL_TEXT%';

SQL_TEXT                       SQL_ID        VERSION_COUNT HASH_VALUE
------------------------------ ------------- ------------- ----------
SELECT * FROM Employees        5bzhzpaa0wy9m             1 2483976499
SELECT * FROM employees        4959aapufrm1k             2 1961610290
```

In the preceding output, the `VERSION_COUNT` of `2` for `SELECT * FROM employees` indicates multiple child cursors, which were necessary because the statement was executed against two different objects. In contrast, the statement `SELECT * FROM Employees` (note the capital "E") was executed once, and so has one parent cursor, and one child cursor (`VERSION_COUNT` of `1`).

Child Cursors and V$SQL

Every parent cursor has one or more child cursors.

A **child cursor** contains the execution plan, bind variables, metadata about objects referenced in the query, optimizer environment, and other information. In contrast to the parent cursor, the child cursor does not store the text of the SQL statement.

If a statement is able to reuse a parent cursor, then the database checks whether the statement can reuse an existing child cursor. The database performs several checks, including the following:

- The database compares objects referenced in the issued statement to the objects referenced by the statement in the pool to ensure that they are all identical.

  References to schema objects in the SQL statements or PL/SQL blocks must resolve to the same object in the same schema. For example, if two users issue the following SQL statement, and if each user has its own `employees` table, then the following statement is not identical because the statement references different `employees` tables for each user:

  ```
  SELECT * FROM employees;
  ```

  > **Note:**
  >
  > The database can share cursors for a private temporary table, but only within the same session. The database associates the session identifier as part of the cursor context. During a soft parse, the database can share the child cursor only when the current session ID matches with the session ID in the cursor context.

- The database determines whether the optimizer mode is identical.

For example, SQL statements must be optimized using the same optimizer goal.

Example 20-2 Multiple Child Cursors

`V$SQL` describes the statements that currently reside in the library cache. It contains one row for every child cursor, as shown in the following example:

```
SELECT SQL_TEXT, SQL_ID, USERNAME AS USR, CHILD_NUMBER AS CHILD#,
       HASH_VALUE, PLAN_HASH_VALUE AS PLAN_HASHV
FROM   V$SQL s, DBA_USERS d
WHERE  SQL_TEXT LIKE '%mployee%'
AND    SQL_TEXT NOT LIKE '%SQL_TEXT%'
AND    d.USER_ID = s.PARSING_USER_ID;

SQL_TEXT               SQL_ID        USR CHILD# HASH_VALUE PLAN_HASHV
---------------------- ------------- --- ------ ---------- ----------
SELECT * FROM Employees 5bzhzpaa0wy9m  HR      0 2483976499 1445457117
SELECT * FROM employees 4959aapufrm1k  HR      0 1961610290 1445457117
SELECT * FROM employees 4959aapufrm1k  SH      1 1961610290 1445457117
```

In the preceding results, the `CHILD#` of the bottom two statements is different (`0` and `1`), even though the `SQL_ID` is the same. This means that the statements have the same parent cursor, but different child cursors. In contrast, the statement with the `SQL_ID` of `5bzhzpaa0wy9m` has one parent and one child (`CHILD#` of `0`). All three SQL statements use the same execution plan, as indicated by identical values in the `PLAN_HASH_VALUE` column.

Related Topics

- Types of Temporary Tables
  Temporary tables are classified as global, private, or cursor-duration.
- Choosing an Optimizer Goal
  The **optimizer goal** is the prioritization of resource usage by the optimizer.

Cursor Mismatches and V$SQL_SHARED_CURSOR

If a parent cursor has multiple children, then the `V$SQL_SHARED_CURSOR` view provides information about why the cursor was not shared. For several types of incompatibility, the `TRANSLATION_MISMATCH` column indicates a mismatch with the value `Y` or `N`.

Example 20-3 Translation Mismatch

In this example, the `TRANSLATION_MISMATCH` column shows that the two statements (`SELECT * FROM employees`) referenced different objects, resulting in a `TRANSLATION_MISMATCH` value of `Y` for the last statement. Because sharing was not possible, each statement had a separate child cursor, as indicated by `CHILD_NUMBER` of `0` and `1`.

```
SELECT S.SQL_TEXT, S.CHILD_NUMBER, s.CHILD_ADDRESS,
       C.TRANSLATION_MISMATCH
FROM   V$SQL S, V$SQL_SHARED_CURSOR C
WHERE  SQL_TEXT LIKE '%employee%'
AND    SQL_TEXT NOT LIKE '%SQL_TEXT%'
AND S.CHILD_ADDRESS = C.CHILD_ADDRESS;

SQL_TEXT                       CHILD_NUMBER    CHILD_ADDRESS T
------------------------------ ------------ ---------------- -
```

```
SELECT * FROM employees                    0 0000000081EE8690 N
SELECT * FROM employees                    1 0000000081F22508 Y
```

About Cursors and Parsing

If an application issues a statement, and if Oracle Database cannot reuse a cursor, then it must build a new executable version of the application code. This operation is known as a **hard parse**.

A soft parse is any parse that is not a hard parse, and occurs when the database can reuse existing code. Some soft parses are less resource-intensive than others. For example, if a parent cursor for the statement already exists, then Oracle Database can perform various optimizations, and then store the child cursor in the shared SQL area. If a parent cursor does not exist, however, then Oracle Database must also store the parent cursor in the shared SQL area, which creates additional memory overhead.

Effectively, a hard parse recompiles a statement before running it. Hard parsing a SQL statement before every execution is analogous to recompiling a C program before every execution. A hard parse performs operations such as the following:

- Checking the syntax of the SQL statement
- Checking the semantics of the SQL statement
- Checking the access rights of the user issuing the statement
- Creating an execution plan
- Accessing the library cache and data dictionary cache numerous times to check the data dictionary

An especially resource-intensive aspect of hard parsing is accessing the library cache and data dictionary cache numerous times to check the data dictionary. When the database accesses these areas, it uses a serialization device called a latch on required objects so that their definition does not change during the check. Latch contention increases statement execution time and decreases concurrency.

For all of the preceding reasons, the CPU and memory overhead of hard parses can create serious performance problems. The problems are especially evident in web applications that accept user input from a form, and then generate SQL statements dynamically. The Real-World Performance group strongly recommends reducing hard parsing as much as possible.

> **Video:**
>
> Video

Example 20-4 Finding Parse Information Using V$SQL

You can use various techniques to monitor hard and soft parsing. This example queries the session statistics to determine whether repeated executions of a `DBA_JOBS` query increase the hard parse count. The first execution of the statement increases the hard parse count to `49`, but the second execution does not change the hard parse count, which means that Oracle Database reused application code.

```
SQL> ALTER SYSTEM FLUSH SHARED_POOL;
```

```
System altered.

SQL> COL NAME FORMAT a18

SQL> SELECT s.NAME, m.VALUE
  2  FROM   V$STATNAME s, V$MYSTAT m
  3  WHERE  s.STATISTIC# = m.STATISTIC#
  4  AND    s.NAME LIKE '%(hard%';

NAME                    VALUE
------------------ ----------
parse count (hard)          48

SQL> SELECT COUNT(*) FROM DBA_JOBS;

  COUNT(*)
----------
     0

SQL> SELECT s.NAME, m.VALUE
  2  FROM   V$STATNAME s, V$MYSTAT m
  3  WHERE  s.STATISTIC# = m.STATISTIC#
  4  AND    s.NAME LIKE '%(hard%';

NAME                    VALUE
------------------ ----------
parse count (hard)          49

SQL> SELECT COUNT(*) FROM DBA_JOBS;

  COUNT(*)
----------
     0

SQL> SELECT s.NAME, m.VALUE
  2  FROM   V$STATNAME s, V$MYSTAT m
  3  WHERE  s.STATISTIC# = m.STATISTIC#
  4  AND    s.NAME LIKE '%(hard%';

NAME                    VALUE
------------------ ----------
parse count (hard)          49
```

Example 20-5 Finding Parse Information Using Trace Files

This example uses SQL Trace and the TKPROF utility to find parse information. You log in to the database with administrator privileges, and then query the directory location of the trace files (sample output included):

```
SET LINESIZE 120
COLUMN value FORMAT A80

SELECT value
FROM   v$diag_info
WHERE  name = 'Default Trace File';
```

```
VALUE
------------------------------------------------------------------------------
/disk1/oracle/log/diag/rdbms/orcl/orcl/trace/orcl_ora_23054.trc
```

You enable tracing, use the `TRACEFILE_IDENTIFIER` initialization parameter to give the trace file a meaningful name, and then query `hr.employees`:

```
EXEC DBMS_MONITOR.SESSION_TRACE_ENABLE(waits=>TRUE, binds=>TRUE);
ALTER SESSION SET TRACEFILE_IDENTIFIER = "emp_stmt";
SELECT * FROM hr.employees;
EXIT;
```

Search the default trace file directory for the trace file that you generated:

```
% ls *emp_stmt.trc
orcl_ora_17950_emp_stmt.trc
```

Use TKPROF to format the trace file, and then open the formatted file:

```
% tkprof orcl_ora_17950_emp_stmt.trc emp.out; vi emp.out
```

The formatted trace file contains the parse information for the query of `hr.employees`.

```
SQL ID: brmjpfs7dcnub Plan Hash: 1445457117

SELECT *
FROM
 hr.employees


call     count       cpu    lapsed       disk      query    current        rows
------- ------  -------- ---------- ---------- ----------  ----------  ----------
Parse        1      0.07       0.08          0          0          0           0
Execute      1      0.00       0.00          0          0          0           0
Fetch        9      0.00       0.00          3         12          0         107
------- ------  -------- ---------- ---------- ----------  ----------  ----------
total       11      0.07       0.08          3         12          0         107

Misses in library cache during parse: 1
Optimizer mode: ALL_ROWS
Parsing user id: SYSTEM
Number of plan statistics captured: 1

Rows (1st) Rows (avg) Rows (max)  Row Source Operation
---------- ---------- ----------  ---------------------------------------------
       107        107         107  TABLE ACCESS FULL EMPLOYEES (cr=12 pr=3
                                    pw=0 time=497 us starts=1 cost=2
                                    size=7383 card=107)
```

A library cache miss indicates a hard parse. Performing the same steps for a second execution of the same statement produces the following trace output, which shows no library cache misses:

```
SQL ID: brmjpfs7dcnub Plan Hash: 1445457117

SELECT *
FROM
 hr.employees


call     count       cpu    elapsed       disk      query    current        rows
------- ------  -------- ---------- ---------- ---------- ----------  ----------
Parse        1      0.00       0.00          0          0          0           0
Execute      1      0.00       0.00          0          0          0           0
Fetch        9      0.00       0.00          3         12          0         107
------- ------  -------- ---------- ---------- ---------- ----------  ----------
total       11      0.00       0.00          3         12          0         107

Misses in library cache during parse: 0
Optimizer mode: ALL_ROWS
Parsing user id: SYSTEM
Number of plan statistics captured: 1

Rows (1st) Rows (avg) Rows (max)  Row Source Operation
---------- ---------- ----------  ---------------------------------------
       107        107         107  TABLE ACCESS FULL EMPLOYEES (cr=12 pr=3
                                   pw=0 time=961 us starts=1 cost=2
                                   size=7383 card=107)
```

> **See Also:**
>
> "Shared Pool Check"

About Literals and Bind Variables

Bind variables are essential to cursor sharing in Oracle database applications.

Literals and Cursors

When constructing SQL statements, some Oracle applications use literals instead of bind variables.

For example, the statement `SELECT SUM(salary) FROM hr.employees WHERE employee_id < 101` uses the literal value `101` for the employee ID. By default, when similar statements do not use bind variables, Oracle Database cannot take advantage of cursor sharing. Thus, Oracle Database sees a statement that is identical except for the value `102`, or any other random value, as a completely new statement, requiring a hard parse.

The Real-World Performance group has determined that applications that use literals are a frequent cause of performance, scalability, and security problems. In the real world, it is not uncommon for applications to be written quickly, without considering cursor sharing. A classic

example is a “screen scraping” application that copies the contents out of a web form, and then concatenates strings to construct the SQL statement dynamically.

Major problems that result from using literal values include the following:

- Applications that concatenate literals input by an end user are prone to SQL injection attacks. Only rewriting the applications to use bind variables eliminates this threat.
- If every statement is hard parsed, then cursors are not shared, and so the database must consume more memory to create the cursors.
- Oracle Database must latch the shared pool and library cache when hard parsing. As the number of hard parses increases, so does the number of processes waiting to latch the shared pool. This situation decreases concurrency and increases contention.

Video:

Video

Example 20-6 Literals and Cursor Sharing

Consider an application that executes the following statements, which differ only in literals:

```
SELECT SUM(salary) FROM hr.employees WHERE employee_id < 101;
SELECT SUM(salary) FROM hr.employees WHERE employee_id < 120;
SELECT SUM(salary) FROM hr.employees WHERE employee_id < 165;
```

The following query of `V$SQLAREA` shows that the three statements require three different parent cursors. As shown by `VERSION_COUNT`, each parent cursor requires its own child cursor.

```
COL SQL_TEXT FORMAT a30
SELECT SQL_TEXT, SQL_ID, VERSION_COUNT, HASH_VALUE
FROM   V$SQLAREA
WHERE  SQL_TEXT LIKE '%mployee%'
AND    SQL_TEXT NOT LIKE '%SQL_TEXT%';

SQL_TEXT                       SQL_ID VERSION_COUNT HASH_VALUE
------------------------------ ------------- ------------- ----------
SELECT SUM(salary) FROM hr.emp b1tvfcc5qnczb             1  191509483
loyees WHERE employee_id < 165
SELECT SUM(salary) FROM hr.emp cn5250y0nqpym             1 2169198547
loyees WHERE employee_id < 101
SELECT SUM(salary) FROM hr.emp au8nag2vnfw67             1 3074912455
loyees WHERE employee_id < 120
```

See Also:

"Do Not Use CURSOR_SHARING = FORCE as a Permanent Fix" to learn about SQL injection

Bind Variables and Cursors

You can develop Oracle applications to use bind variables instead of literals.

A bind variable is a placeholder in a query. For example, the statement `SELECT SUM(salary) FROM hr.employees WHERE employee_id < :emp_id` uses the bind variable `:emp_id` for the employee ID.

The Real-World Performance group has found that applications that use bind variables perform better, scale better, and are more secure. Major benefits that result from using bind variables include the following:

- Applications that use bind variables are not vulnerable to the same SQL injection attacks as applications that use literals.
- When identical statements use bind variables, Oracle Database can take advantage of cursor sharing, and share the plan and other information when different values are bound to the same statement.
- Oracle Database avoids the overhead of latching the shared pool and library cache required for hard parsing.

> **Note:**
>
> You cannot use more that 65535 bind variables in a query.

> **Video:**
>
> Video

Example 20-7 Bind Variables and Shared Cursors

The following example uses the `VARIABLE` command in SQL*Plus to create the `emp_id` bind variable, and then executes a query using three different bind values (`101`, `120`, and `165`):

```
VARIABLE emp_id NUMBER

EXEC :emp_id := 101;
SELECT SUM(salary) FROM hr.employees WHERE employee_id < :emp_id;
EXEC :emp_id := 120;
SELECT SUM(salary) FROM hr.employees WHERE employee_id < :emp_id;
EXEC :emp_id := 165;
SELECT SUM(salary) FROM hr.employees WHERE employee_id < :emp_id;
```

The following query of `V$SQLAREA` shows one unique SQL statement:

```
COL SQL_TEXT FORMAT a34
SELECT SQL_TEXT, SQL_ID, VERSION_COUNT, HASH_VALUE
FROM   V$SQLAREA
WHERE  SQL_TEXT LIKE '%mployee%'
AND    SQL_TEXT NOT LIKE '%SQL_TEXT%';
```

```
SQL_TEXT                          SQL_ID        VERSION_COUNT HASH_VALUE
--------------------------------- ------------- ------------- ----------
SELECT SUM(salary) FROM hr.employe 4318cbskba8yh             1 615850960
es WHERE employee_id < :emp_id
```

The `VERSION_COUNT` value of `1` indicates that the database reused the same child cursor rather than creating three separate child cursors. Using a bind variable made this reuse possible.

> **Note:**
>
> A maximum of 65535 bind variables can be used in a query. Also note that there are circumstances in which bind sensitivity is not used:
>
> - The bind is used in an equality or a range predicate.
> - The optimizer has peeked at the bind values to generate cardinality estimates.
> - The number of bind variables does not exceed internally-defined thresholds.

Bind Variable Peeking

In **bind variable peeking** (also known as **bind peeking**), the optimizer looks at the value in a bind variable when the database performs a hard parse of a statement.

The optimizer does not look at the bind variable values before every parse. Rather, the optimizer peeks only when the optimizer is *first* invoked, which is during the hard parse.

When a query uses literals, the optimizer can use the literal values to find the best plan. However, when a query uses bind variables, the optimizer must select the best plan without the presence of literals in the SQL text. This task can be extremely difficult. By peeking at bind values during the initial hard parse, the optimizer can determine the cardinality of a `WHERE` clause condition as if literals *had* been used, thereby improving the plan.

Because the optimizer only peeks at the bind value during the hard parse, the plan may not be optimal for all possible bind values. The following examples illustrate this principle.

Example 20-8 Literals Result in Different Execution Plans

Assume that you execute the following statements, which execute three different statements using different literals (`101`, `120`, and `165`), and then display the execution plans for each:

```
SET LINESIZE 167
SET PAGESIZE 0
SELECT SUM(salary) FROM hr.employees WHERE employee_id < 101;
SELECT * FROM TABLE(DBMS_XPLAN.DISPLAY_CURSOR());
SELECT SUM(salary) FROM hr.employees WHERE employee_id < 120;
SELECT * FROM TABLE(DBMS_XPLAN.DISPLAY_CURSOR());
SELECT SUM(salary) FROM hr.employees WHERE employee_id < 165;
SELECT * FROM TABLE(DBMS_XPLAN.DISPLAY_CURSOR());
```

The database hard parsed all three statements, which were not identical. The `DISPLAY_CURSOR` output, which has been edited for clarity, shows that the optimizer chose the same index range

scan plan for the first two statements, but a full table scan plan for the statement using literal 165:

```
SQL_ID    cn5250y0nqpym, child number 0
-------------------------------------
SELECT SUM(salary) FROM hr.employees WHERE employee_id < 101

Plan hash value: 2410354593

-----------------------------------------------------------------------------------------
|Id| Operation                             | Name         |Rows|Bytes|Cost(%CPU)|Time|
-----------------------------------------------------------------------------------------
| 0| SELECT STATEMENT                      |              |  |   |2 (100)|          |
| 1|  SORT AGGREGATE                       |              |1 | 8 |       |          |
| 2|   TABLE ACCESS BY INDEX ROWID BATCHED| EMPLOYEES     |1 | 8 |2 (0)   | 00:00:01 |
|*3|    INDEX RANGE SCAN                   | EMP_EMP_ID_PK |1 |   |1 (0)   | 00:00:01 |
-----------------------------------------------------------------------------------------
Predicate Information (identified by operation id):
---------------------------------------------------

   3 - access("EMPLOYEE_ID"<101)

SQL_ID    au8nag2vnfw67, child number 0
-------------------------------------
SELECT SUM(salary) FROM hr.employees WHERE employee_id < 120

Plan hash value: 2410354593

-----------------------------------------------------------------------------------------
|Id| Operation                             | Name         |Rows|Bytes|Cost(%CPU)|Time|
-----------------------------------------------------------------------------------------
| 0| SELECT STATEMENT                      |              |  |   |2 (100)|          |
| 1|  SORT AGGREGATE                       |              |1 | 8 |       |          |
| 2|   TABLE ACCESS BY INDEX ROWID BATCHED| EMPLOYEES     |20|160|2 (0)   | 00:00:01 |
|*3|    INDEX RANGE SCAN                   | EMP_EMP_ID_PK |20|   |1 (0)   | 00:00:01 |
-----------------------------------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------

   3 - access("EMPLOYEE_ID"<120)

SQL_ID    b1tvfcc5qnczb, child number 0
-------------------------------------
SELECT SUM(salary) FROM hr.employees WHERE employee_id < 165

Plan hash value: 1756381138

--------------------------------------------------------------------------
| Id  | Operation          | Name      |Rows| Bytes |Cost(%CPU)| Time    |
--------------------------------------------------------------------------
|   0 | SELECT STATEMENT   |           |    |       | 2 (100)|          |
|   1 |  SORT AGGREGATE    |           |  1 |     8 |        |          |
|*  2 |   TABLE ACCESS FULL| EMPLOYEES | 66 |   528 | 2   (0)| 00:00:01 |
--------------------------------------------------------------------------
```

```
Predicate Information (identified by operation id):
---------------------------------------------------

   2 - filter("EMPLOYEE_ID"<165)
```

The preceding output shows that the optimizer considers a full table scan more efficient than an index scan for the query that returns more rows.

Example 20-9 Bind Variables Result in Cursor Reuse

This example rewrites the queries executed in Example 20-8 to use bind variables instead of literals. You bind the same values (`101`, `120`, and `165`) to the bind variable `:emp_id`, and then display the execution plans for each:

```
VAR emp_id NUMBER

EXEC :emp_id := 101;
SELECT SUM(salary) FROM hr.employees WHERE employee_id < :emp_id;
SELECT * FROM TABLE(DBMS_XPLAN.DISPLAY_CURSOR());
EXEC :emp_id := 120;
SELECT SUM(salary) FROM hr.employees WHERE employee_id < :emp_id;
SELECT * FROM TABLE(DBMS_XPLAN.DISPLAY_CURSOR());
EXEC :emp_id := 165;
SELECT SUM(salary) FROM hr.employees WHERE employee_id < :emp_id;
SELECT * FROM TABLE(DBMS_XPLAN.DISPLAY_CURSOR());
```

The `DISPLAY_CURSOR` output shows that the optimizer chose exactly the same plan for all three statements:

```
SELECT SUM(salary) FROM hr.employees WHERE employee_id < :emp_id

Plan hash value: 2410354593

-------------------------------------------------------------------------------------------
| Id | Operation                            | Name       |Rows|Bytes|Cost (%CPU)|Time|
-------------------------------------------------------------------------------------------
|  0 | SELECT STATEMENT                     |            | |  |2 (100)|          |
|  1 |  SORT AGGREGATE                      |            |1|8 |       |          |
|  2 |   TABLE ACCESS BY INDEX ROWID BATCHED| EMPLOYEES  |1|8 |  2 (0)| 00:00:01 |
|* 3 |    INDEX RANGE SCAN                  | EMP_EMP_ID_PK |1|  |  1 (0)| 00:00:01 |
-------------------------------------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------

   3 - access("EMPLOYEE_ID"<:EMP_ID)
```

In contrast, when the preceding statements were executed with literals, the optimizer chose a lower-cost full table scan when the employee ID value was `165`. This is the problem solved by adaptive cursor sharing.

> **See Also:**
>
> "Adaptive Cursor Sharing"

About the Life Cycle of Shared Cursors

The database allocates a new shared SQL area when the optimizer parses a new SQL statement that is not DDL. The amount of memory required depends on the statement complexity.

The database can remove a shared SQL area from the shared pool even if this area corresponds to an open cursor that has been unused for a long time. If the open cursor is later used to run its statement, then the database reparses the statement and allocates a new shared SQL area. The database does not remove cursors whose statements are executing, or whose rows have not been completely fetched.

Shared SQL areas can become invalid because of changes to dependent schema objects or to optimizer statistics. Oracle Database uses two techniques to manage the cursor life cycle: invalidation and rolling invalidation.

> **See Also:**
>
> *Oracle Database Concepts* for an overview of memory allocation in the shared pool

Cursor Marked Invalid

When a shared SQL area is marked invalid, the database can remove it from the shared pool, along with valid cursors that have been unused for some time.

In some situations, the database must execute a statement that is associated with an invalid shared SQL area in the shared pool. In this case, the database performs a hard parse of the statement before execution.

The database immediately marks dependent shared SQL areas invalid when the following conditions are met:

- `DBMS_STATS` gathers statistics for a table, table cluster, or index when the `NO_INVALIDATE` parameter is `FALSE`.
- A SQL statement references a schema object, which is later modified by a DDL statement that uses immediate cursor invalidation (default).

  You can manually specify immediate invalidation on statements such as `ALTER TABLE ... IMMEDIATE VALIDATION` and `ALTER INDEX ... IMMEDIATE VALIDATION`, or set the `CURSOR_INVALIDATION` initialization parameter to `IMMEDIATE` at the session or system level.

> **Note:**
>
> A DDL statement using the `DEFERRED VALIDATION` clause overrides the `IMMEDIATE` setting of the `CURSOR_INVALIDATION` initialization parameter.

When the preceding conditions are met, the database reparses the affected statements at next execution.

When the database invalidates a cursor, the `V$SQL.INVALIDATIONS` value increases (for example, from `0` to `1`), and `V$SQL.OBJECT_STATUS` shows `INVALID_UNAUTH`.

Example 20-10 Forcing Cursor Invalidation by Setting NO_INVALIDATE=FALSE

This example logs in as user `sh`, who has been granted administrator privileges. The example queries `sales`, and then gathers statistics for this table with `NO_INVALIDATE=FALSE`. Afterward, the `V$SQL.INVALIDATIONS` value changes from `0` to `1` for the cursor, indicating that the database flagged the cursor as invalid.

```
SQL> SELECT COUNT(*) FROM sales;

  COUNT(*)
----------
    918843

SQL> SELECT PREV_SQL_ID SQL_ID FROM V$SESSION WHERE SID =
SYS_CONTEXT('userenv', 'SID');

SQL_ID
-------------
1y17j786c7jbh

SQL> SELECT CHILD_NUMBER, EXECUTIONS,
  2  PARSE_CALLS, INVALIDATIONS, OBJECT_STATUS
  3  FROM V$SQL WHERE SQL_ID = '1y17j786c7jbh';

CHILD_NUMBER EXECUTIONS PARSE_CALLS INVALIDATIONS OBJECT_STATUS
------------ ---------- ----------- ------------- -------------
           0          1           1             0 VALID

SQL> EXEC DBMS_STATS.GATHER_TABLE_STATS(null,'sales',no_invalidate => FALSE);

PL/SQL procedure successfully completed.

SQL> SELECT CHILD_NUMBER, EXECUTIONS,
  2  PARSE_CALLS, INVALIDATIONS, OBJECT_STATUS
  3  FROM V$SQL WHERE SQL_ID = '1y17j786c7jbh';

CHILD_NUMBER EXECUTIONS PARSE_CALLS INVALIDATIONS OBJECT_STATUS
------------ ---------- ----------- ------------- --------------
           0          1           1             1 INVALID_UNAUTH
```

> **See Also:**
>
> - "About Optimizer Initialization Parameters"
> - *Oracle Database SQL Language Reference* to learn more about `ALTER TABLE ... IMMEDIATE VALIDATION` and other DDL statements that permit immediate validation
> - *Oracle Database Reference* to learn more about `V$SQL` and `V$SQLAREA` dynamic views
> - *Oracle Database Reference* to learn more about the `CURSOR_INVALIDATION` initialization parameter

Cursors Marked Rolling Invalid

When cursors are marked rolling invalid (`V$SQL.IS_ROLLING_INVALID` is `Y`), the database gradually performs hard parses over an extended time.

> **Note:**
>
> When `V$SQL.IS_ROLLING_REFRESH_INVALID` is `Y`, the underlying object has changed, but recompilation of the cursor is not required. The database updates metadata in the cursor.

Purpose of Rolling Invalidation

Because a sharp increase in hard parses can significantly degrade performance, rolling invalidation—also called *deferred invalidation*—is useful for workloads that simultaneously invalidate many cursors. The database assigns each invalid cursor a randomly generated time period. SQL areas invalidated at the same time typically have different time periods.

A hard parse occurs only if a query accessing the cursor executes *after* the time period has expired. In this way, the database diffuses the overhead of hard parsing over time.

> **Note:**
>
> If parallel SQL statements are marked rolling invalid, then the database performs a hard parse at next execution, regardless of whether the time period has expired. In an Oracle Real Application Clusters (Oracle RAC) environment, this technique ensures consistency between execution plans of parallel execution servers and the query coordinator.

An analogy for rolling invalidation might be the gradual replacement of worn-out office furniture. Instead of replacing all the furniture at once, forcing a substantial financial outlay, a company assigns each piece a different expiration date. Over the course of a year, a piece stays in use until it is replaced, at which point a cost is incurred.

Specification of Deferred Invalidation

By default, DDL specifies that statements accessing the object use immediate cursor invalidation. For example, if you create a table or an index, then cursors that reference this table or index use immediate invalidation.

If a DDL statement supports deferred cursor invalidation, then you can override the default behavior by using statements such as `ALTER TABLE ... DEFERRED INVALIDATION`. The options depend on the DDL statement. For example, `ALTER INDEX` only supports `DEFERRED INVALIDATION` when the `UNUSABLE` or `REBUILD` option is also specified.

An alternative to DDL is setting the `CURSOR_INVALIDATION` initialization parameter to `DEFERRED` at the session or system level. A DDL statement using the `IMMEDIATE INVALIDATION` clause overrides the `DEFERRED` setting of the `CURSOR_INVALIDATION` initialization parameter.

When Rolling Invalidation Occurs

If the `DEFERRED INVALIDATION` attribute applies to an object, either as a result of DDL or an initialization parameter setting, then statements that access the object may be subject to deferred invalidation. The database marks shared SQL areas as rolling invalid in either of the following circumstances:

- `DBMS_STATS` gathers statistics for a table, table cluster, or index when the `NO_INVALIDATE` parameter is set to `DBMS_STATS.AUTO_INVALIDATE`. This is the default setting.
- One of the following statements is issued with `DEFERRED INVALIDATION` in circumstances that do not prevent the use of deferred invalidation:
  - `ALTER TABLE` on partitioned tables
  - `ALTER TABLE ... PARALLEL`
  - `ALTER INDEX ... UNUSABLE`
  - `ALTER INDEX ... REBUILD`
  - `CREATE INDEX`
  - `DROP INDEX`
  - `TRUNCATE TABLE` on partitioned tables

  A subset of DDL statements require immediate cursor invalidation for DML (`INSERT`, `UPDATE`, `DELETE`, or `MERGE`) but not `SELECT` statements. Many factors relating to the specific DDL statements and affected cursors determine whether Oracle Database uses deferred invalidation.

See Also:

- "About Optimizer Initialization Parameters"
- *Oracle Database SQL Language Reference* to learn more about `ALTER TABLE ... DEFERRED INVALIDATION` and other DDL statements that permit deferred invalidation
- *Oracle Database Reference* to learn more about `V$SQL` and `V$SQLAREA` dynamic views
- *Oracle Database Reference* to learn more about the `CURSOR_INVALIDATION` initialization parameter

CURSOR_SHARING and Bind Variable Substitution

This topic explains what the `CURSOR_SHARING` initialization parameter is, and how setting it to different values affects how Oracle Database uses bind variables.

CURSOR_SHARING Initialization Parameter

The `CURSOR_SHARING` initialization parameter controls how the database processes statements with bind variables.

In Oracle Database 12c, the parameter supports the following values:

- `EXACT`

  This is the default value. The database enables only textually identical statements to share a cursor. The database does not attempt to replace literal values with system-generated bind variables. In this case, the optimizer generates a plan for each statement based on the literal value.

- `FORCE`

  The database replaces all literals with system-generated bind variables. For statements that are identical after the bind variables replace the literals, the optimizer uses the same plan.

Note:

The `SIMILAR` value for `CURSOR_SHARING` is deprecated.

You can set `CURSOR_SHARING` at the system or session level, or use the `CURSOR_SHARING_EXACT` hint at the statement level.

See Also:

"Do Not Use CURSOR_SHARING = FORCE as a Permanent Fix"

Parsing Behavior When CURSOR_SHARING = FORCE

When SQL statements use literals rather than bind variables, setting the `CURSOR_SHARING` initialization parameter to `FORCE` enables the database to replace literals with system-generated bind variables. Using this technique, the database can sometimes reduce the number of parent cursors in the shared SQL area.

> **Note:**
>
> If a statement uses an `ORDER BY` clause, then while the database may perform literal replacement in the clause, the cursor will not necessarily be sharable because different values of the column number as a literal imply different query results and potentially a different execution plan. The column number in the `ORDER BY` clause affects the query plan and execution, so the database cannot share two cursors having different column numbers.

When `CURSOR_SHARING` is set to `FORCE`, the database performs the following steps during the parse:

1. Copies *all* literals in the statement to the PGA, and replaces them with system-generated bind variables

   For example, an application could process the following statement:

   ```
   SELECT SUBSTR(last_name, 1, 4), SUM(salary)
   FROM   hr.employees
   WHERE  employee_id < 101 GROUP BY last_name
   ```

   The optimizer replaces literals, including the literals in the `SUBSTR` function, as follows:

   ```
   SELECT SUBSTR(last_name, :"SYS_B_0", :"SYS_B_1"), SUM(salary)
   FROM   hr.employees
   WHERE  employee_id < :"SYS_B_2" GROUP BY last_name
   ```

2. Searches for an identical statement (same SQL hash value) in the shared pool

   If an identical statement is *not* found, then the database performs a hard parse. Otherwise, the database proceeds to the next step.

3. Performs a soft parse of the statement

As the preceding steps indicate, setting the `CURSOR_SHARING` initialization parameter to `FORCE` does *not* reduce the parse count. Rather, in some cases, `FORCE` enables the database to perform a soft parse instead of a hard parse. Also, `FORCE` does not the prevent against SQL injection attacks because Oracle Database binds the values after any injection has already occurred.

Example 20-11 Replacement of Literals with System Bind Variables

This example sets `CURSOR_SHARING` to `FORCE` at the session level, executes three statements containing literals, and displays the plan for each statement:

```
ALTER SESSION SET CURSOR_SHARING=FORCE;
SET LINESIZE 170
SET PAGESIZE 0
SELECT SUM(salary) FROM hr.employees WHERE employee_id < 101;
SELECT * FROM TABLE(DBMS_XPLAN.DISPLAY_CURSOR());
SELECT SUM(salary) FROM hr.employees WHERE employee_id < 120;
SELECT * FROM TABLE(DBMS_XPLAN.DISPLAY_CURSOR());
SELECT SUM(salary) FROM hr.employees WHERE employee_id < 165;
SELECT * FROM TABLE(DBMS_XPLAN.DISPLAY_CURSOR());
```

The following `DISPLAY_CURSOR` output, edited for readability, shows that all three statements used the same plan. The optimizer chose the plan, an index range scan, because it peeked at the *first* value (`101`) bound to the system bind variable, and picked this plan as the best for all values. In fact, this plan is not the best plan for all values. When the value is `165`, a full table scan is more efficient.

```
SQL_ID    cxx8n1cxr9khn, child number 0
-------------------------------------
SELECT SUM(salary) FROM hr.employees WHERE employee_id < :"SYS_B_0"

Plan hash value: 2410354593

-----------------------------------------------------------------------------------------
| Id  | Operation                            | Name       |Rows|Bytes|Cost(%CPU)|Time|
-----------------------------------------------------------------------------------------
|  0 | SELECT STATEMENT                     |              |  |   |2 (100)|        |
|  1 |  SORT AGGREGATE                      |              |1 | 8 |       |        |
|  2 |   TABLE ACCESS BY INDEX ROWID BATCHED| EMPLOYEES    |1 | 8 |2 (0)  |00:00:01|
|* 3 |    INDEX RANGE SCAN                  | EMP_EMP_ID_PK |1 |   |1 (0)  |00:00:01|
-----------------------------------------------------------------------------------------
Predicate Information (identified by operation id):
---------------------------------------------------

   3 - access("EMPLOYEE_ID"<101)
```

A query of `V$SQLAREA` confirms that Oracle Database replaced with the literal with system bind variable `:"SYS_B_0"`, and created one parent and one child cursor (`VERSION_COUNT=1`) for all three statements, which means that all executions shared the same plan.

```
COL SQL_TEXT FORMAT a36
SELECT SQL_TEXT, SQL_ID, VERSION_COUNT, HASH_VALUE
FROM   V$SQLAREA
WHERE  SQL_TEXT LIKE '%mployee%'
AND    SQL_TEXT NOT LIKE '%SQL_TEXT%';

SQL_TEXT                             SQL_ID        VERSION_COUNT HASH_VALUE
------------------------------------ ------------- ------------- ----------
```

```
SELECT SUM(salary) FROM hr.employees cxx8n1cxr9khn              1   997509652
 WHERE employee_id < :"SYS_B_0"
```

> **See Also:**
>
> - "Private and Shared SQL Areas" for more details on the various checks performed
> - *Oracle Database Reference* to learn about the `CURSOR_SHARING` initialization parameter

Adaptive Cursor Sharing

The **adaptive cursor sharing** feature enables a single statement that contains bind variables to use multiple execution plans.

Cursor sharing is "adaptive" because the cursor adapts its behavior so that the database does not always use the same plan for each execution or bind variable value.

Purpose of Adaptive Cursor Sharing

With bind peeking, the optimizer peeks at the values of user-defined bind variables on the first invocation of a cursor.

The optimizer determines the cardinality of any `WHERE` clause condition as if literals had been used instead of bind variables. If a column in a `WHERE` clause has skewed data, however, then a histogram may exist on this column. When the optimizer peeks at the value of the user-defined bind variable and chooses a plan, this plan may not be good for all values.

In adaptive cursor sharing, the database monitors data accessed over time for different bind values, ensuring the optimal choice of cursor for a specific bind value. For example, the optimizer might choose one plan for bind value `10` and a different plan for bind value `50`. Cursor sharing is "adaptive" because the cursor adapts its behavior so that the optimizer does not always choose the same plan for each execution or bind variable value. Thus, the optimizer automatically detects when different execution of a statement would benefit from different execution plans.

> **Note:**
>
> Adaptive cursor sharing is independent of the `CURSOR_SHARING` initialization parameter. Adaptive cursor sharing is equally applicable to statements that contain user-defined and system-generated bind variables. Adaptive cursor sharing does not apply to statements that contain only literals.

How Adaptive Cursor Sharing Works: Example

Adaptive cursor sharing monitors statements that use bind variables to determine whether a new plan is more efficient.

Assume that an application executes the following statement five times, binding different values every time:

```
SELECT * FROM employees WHERE salary = :sal AND department_id = :dept
```

Also assume in this example that a histogram exists on at least one of the columns in the predicate. The database processes this statement as follows:

1. The application issues the statement for the first time, which causes a hard parse. During the parse, the database performs the following tasks:
   - Peeks at the bind variables to generate the initial plan.
   - Marks the cursor as bind-sensitive. A bind-sensitive cursor is a cursor whose optimal plan may depend on the value of a bind variable. To determine whether a different plan is beneficial, the database monitors the behavior of a bind-sensitive cursor that uses different bind values.
   - Stores metadata about the predicate, including the cardinality of the bound values (in this example, assume that only 5 rows were returned).
   - Creates an execution plan (in this example, index access) based on the peeked values.
2. The database executes the cursor, storing the bind values and execution statistics in the cursor.
3. The application issues the statement a second time, using different bind variables, causing the database to perform a soft parse, and find the matching cursor in the library cache.
4. The database executes the cursor.
5. The database performs the following post-execution tasks:
   a. The database compares the execution statistics for the second execution with the first-execution statistics.
   b. The database observes the pattern of statistics over all previous executions, and then decides whether to mark the cursor as a bind-aware cursor. In this example, assume that the database decides the cursor is bind-aware.
6. The application issues the statement a third time, using different bind variables, which causes a soft parse. Because the cursor is bind-aware, the database does the following:
   - Determines whether the cardinality of the new values falls within the same range as the stored cardinality. In this example, the cardinality is similar: 8 rows instead of 5 rows.
   - Reuses the execution plan in the existing child cursor.
7. The database executes the cursor.
8. The application issues the statement a fourth time, using different bind variables, causing a soft parse. Because the cursor is bind-aware, the database does the following:
   - Determines whether the cardinality of the new values falls within the same range as the stored cardinality. In this example, the cardinality is vastly different: 102 rows (in a table with 107 rows) instead of 5 rows.
   - Does not find a matching child cursor.
9. The database performs a hard parse. As a result, the database does the following:
   - Creates a new child cursor with a *second* execution plan (in this example, a full table scan)

- Stores metadata about the predicate, including the cardinality of the bound values, in the cursor

10. The database executes the new cursor.
11. The database stores the new bind values and execution statistics in the new child cursor.
12. The application issues the statement a fifth time, using different bind variables, which causes a soft parse. Because the cursor is bind-aware, the database does the following:
    - Determines whether the cardinality of the new values falls within the same range as the stored cardinality. In this example, the cardinality is 20.
    - Does not find a matching child cursor.
13. The database performs a hard parse. As a result, the database does the following:
    a. Creates a new child cursor with a *third* execution plan (in this example, index access)
    b. Determines that this index access execution plan is the same as the index access execution plan used for the first execution of the statement
    c. Merges the two child cursors containing index access plans, which involves storing the combined cardinality statistics into one child cursor, and deleting the other one
14. The database executes the cursor using the index access execution plan.

Bind-Sensitive Cursors

A **bind-sensitive cursor** is a cursor whose optimal plan may depend on the value of a bind variable.

The database has examined the bind value when computing cardinality, and considers the query “sensitive” to plan changes based on different bind values. The database monitors the behavior of a bind-sensitive cursor that uses different bind values to determine whether a different plan is beneficial.

The optimizer uses the following criteria to decide whether a cursor is bind-sensitive:

- The optimizer has peeked at the bind values to generate cardinality estimates.
- The bind is used in an equality or a range predicate.

For each execution of the query with a new bind value, the database records the execution statistics for the new value and compares them to the execution statistics for the previous value. If execution statistics vary greatly, then the database marks the cursor bind-aware.

Example 20-12 Column with Significant Data Skew

This example assumes that the `hr.employees.department_id` column has significant data skew. `SYSTEM` executes the following setup code, which adds 100,000 employees in department 50 to the `employees` table in the sample schema, for a total of 100,107 rows, and then gathers table statistics:

```
DELETE FROM hr.employees WHERE employee_id > 999;

ALTER TABLE hr.employees DISABLE NOVALIDATE CONSTRAINT emp_email_uk;

DECLARE
v_counter NUMBER(7) := 1000;
BEGIN
 FOR i IN 1..100000 LOOP
 INSERT INTO hr.employees
```

```
   VALUES (v_counter, null, 'Doe', 'Doe@example.com', null,'07-JUN-02',
     'AC_ACCOUNT', null, null, null, 50);
 v_counter := v_counter + 1;
 END LOOP;
END;
/
COMMIT;

BEGIN
  DBMS_STATS.GATHER_TABLE_STATS (ownname = 'hr',tabname => 'employees');
END;
/

ALTER SYSTEM FLUSH SHARED_POOL;
```

The following query shows a histogram on the `employees.department_id` column:

```
COL TABLE_NAME FORMAT a15
COL COLUMN_NAME FORMAT a20
COL HISTOGRAM FORMAT a9

SELECT TABLE_NAME, COLUMN_NAME, HISTOGRAM
FROM   DBA_TAB_COLS
WHERE  OWNER = 'HR'
AND    TABLE_NAME = 'EMPLOYEES'
AND    COLUMN_NAME = 'DEPARTMENT_ID';

TABLE_NAME      COLUMN_NAME          HISTOGRAM
--------------- -------------------- ---------
EMPLOYEES       DEPARTMENT_ID        FREQUENCY
```

Example 20-13 Low-Cardinality Query

This example continues the example in Example 20-12. The following query shows that the value `10` has extremely low cardinality for the column `department_id`, occupying .00099% of the rows:

```
VARIABLE dept_id NUMBER
EXEC :dept_id := 10;
SELECT COUNT(*), MAX(employee_id) FROM hr.employees WHERE department_id
= :dept_id;

  COUNT(*) MAX(EMPLOYEE_ID)
---------- ----------------
         1              200
```

The optimizer chooses an index range scan, as expected for such a low-cardinality query:

```
SQL> SELECT * FROM TABLE(DBMS_XPLAN.DISPLAY_CURSOR);

PLAN_TABLE_OUTPUT
------------------------------------
SQL_ID    a9upgaqqj7bn5, child number 0
------------------------------------
```

```
select COUNT(*), MAX(employee_id) FROM hr.employees WHERE department_id = :dept_id

Plan hash value: 1642965905

-------------------------------------------------------------------------------------------
| Id| Operation                             | Name       |Rows|Bytes|Cost (%CPU)|Time |
-------------------------------------------------------------------------------------------
| 0| SELECT STATEMENT                       |                  |  |  |2(100)|        |
| 1|  SORT AGGREGATE                        |                  |1 |8 |      |        |
| 2|   TABLE ACCESS BY INDEX ROWID BATCHED| EMPLOYEES          |1 |8 |2  (0)|00:00:01|
|*3|    INDEX RANGE SCAN                    | EMP_DEPARTMENT_IX |1 |  |1  (0)|00:00:01|
-------------------------------------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------

   3 - access("DEPARTMENT_ID"=:DEPT_ID)
```

The following query of `V$SQL` obtains information about the cursor:

```
COL BIND_AWARE FORMAT a10
COL SQL_TEXT FORMAT a22
COL CHILD# FORMAT 99999
COL EXEC FORMAT 9999
COL BUFF_GETS FORMAT 999999999
COL BIND_SENS FORMAT a9
COL SHARABLE FORMAT a9

SELECT SQL_TEXT, CHILD_NUMBER AS CHILD#, EXECUTIONS AS EXEC,
       BUFFER_GETS AS BUFF_GETS, IS_BIND_SENSITIVE AS BIND_SENS,
       IS_BIND_AWARE AS BIND_AWARE, IS_SHAREABLE AS SHARABLE
FROM   V$SQL
WHERE  SQL_TEXT LIKE '%mployee%'
AND    SQL_TEXT NOT LIKE '%SQL_TEXT%';

SQL_TEXT               CHILD#  EXEC  BUFF_GETS BIND_SENS BIND_AWARE SHARABLE
---------------------- ------ ----- ---------- --------- ---------- --------
SELECT COUNT(*), MAX(e      0     1        196         Y          N        Y
mployee_id) FROM hr.em
ployees WHERE departme
nt_id = :dept_id
```

The preceding output shows one child cursor that has been executed once for the low-cardinality query. The cursor has been marked bind-sensitive because the optimizer believes the optimal plan may depend on the value of the bind variable.

When a cursor is marked bind-sensitive, Oracle Database monitors the behavior of the cursor using different bind values, to determine whether a different plan for different bind values is more efficient. The database marked this cursor bind-sensitive because the optimizer used the histogram on the `department_id` column to compute the selectivity of the predicate `WHERE department_id = :dept_id`. Because the presence of the histogram indicates that the column is skewed, different values of the bind variable may require different plans.

Example 20-14 High-Cardinality Query

This example continues the example in Example 20-13. The following code re-executes the same query using the value `50`, which occupies 99.9% of the rows:

```
EXEC :dept_id := 50;
SELECT COUNT(*), MAX(employee_id) FROM hr.employees WHERE department_id
= :dept_id;

  COUNT(*) MAX(EMPLOYEE_ID)
---------- ---------------
    100045          100999
```

Even though such an unselective query would be more efficient with a full table scan, the optimizer chooses the same index range scan used for `department_id=10`. This reason is that the database assumes that the existing plan in the cursor can be shared:

```
SQL> SELECT * FROM TABLE(DBMS_XPLAN.DISPLAY_CURSOR);

PLAN_TABLE_OUTPUT
------------------------------------
SQL_ID    a9upgaqqj7bn5, child number 0
------------------------------------
SELECT COUNT(*), MAX(employee_id) FROM hr.employees WHERE department_id = :dept_id

Plan hash value: 1642965905

-------------------------------------------------------------------------------------------
| Id| Operation                            | Name       |Rows|Bytes|Cost (%CPU)|Time |
-------------------------------------------------------------------------------------------
| 0| SELECT STATEMENT                      |                  |  |  |2(100)|        |
| 1|  SORT AGGREGATE                       |                  |1 |8 |      |        |
| 2|   TABLE ACCESS BY INDEX ROWID BATCHED| EMPLOYEES         |1 |8 |2  (0)|00:00:01|
|*3|    INDEX RANGE SCAN                   | EMP_DEPARTMENT_IX |1 |  |1  (0)|00:00:01|
-------------------------------------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------

   3 - access("DEPARTMENT_ID"=:DEPT_ID)
```

A query of `V$SQL` shows that the child cursor has now been executed twice:

```
SELECT SQL_TEXT, CHILD_NUMBER AS CHILD#, EXECUTIONS AS EXEC,
       BUFFER_GETS AS BUFF_GETS, IS_BIND_SENSITIVE AS BIND_SENS,
       IS_BIND_AWARE AS BIND_AWARE, IS_SHAREABLE AS SHARABLE
FROM   V$SQL
WHERE  SQL_TEXT LIKE '%mployee%'
AND    SQL_TEXT NOT LIKE '%SQL_TEXT%';

SQL_TEXT               CHILD#  EXEC  BUFF_GETS BIND_SENS BIND_AWARE SHARABLE
---------------------- ------ ----- ---------- --------- ---------- --------
SELECT COUNT(*), MAX(e      0     2       1329         Y          N        Y
mployee_id) FROM hr.em
```

```
ployees WHERE departme
nt_id = :dept_id
```

At this stage, the optimizer has not yet marked the cursor as bind-aware.

> **Note:**
>
> There are circumstances in which bind sensitivity is not used. Specifically:
>
> - If collection bind variables are used, such as PL/SQL arrays.
> - When the number of bind variables exceed internal limits.

> **See Also:**
>
> *Oracle Database Reference* to learn about `V$SQL`

Bind-Aware Cursors

A **bind-aware cursor** is a bind-sensitive cursor that is eligible to use different plans for different bind values.

After a cursor has been made bind-aware, the optimizer chooses plans for future executions based on the bind value and its cardinality estimate. Thus, “bind-aware” means essentially “best plan for the current bind value.”

When a statement with a bind-sensitive cursor executes, the optimizer uses an internal algorithm to determine whether to mark the cursor bind-aware. The decision depends on whether the cursor produces significantly different data access patterns for different bind values, resulting in a performance cost that differs from expectations.

If the database marks the cursor bind-aware, then the *next* time that the cursor executes the database does the following:

- Generates a new plan based on the bind value
- Marks the original cursor generated for the statement as not sharable (`V$SQL.IS_SHAREABLE` is `N`). The original cursor is no longer usable and is eligible to age out of the library cache

When the same query repeatedly executes with different bind values, the database adds new bind values to the “signature” of the SQL statement (which includes the optimizer environment, NLS settings, and so on), and categorizes the values. The database examines the bind values, and considers whether the current bind value results in a significantly different data volume, or whether an existing plan is sufficient. The database does *not* need to create a new plan for each new value.

Consider a scenario in which you execute a statement with 12 distinct bind values (executing each distinct value twice), which causes the database to trigger 5 hard parses, and create 2 additional plans. Because the database performs 5 hard parses, it creates 5 new child cursors, even though some cursors have the same execution plan as existing cursors. The database marks the superfluous cursors as not usable, which means these cursors eventually age out of the library cache.

During the initial hard parses, the optimizer is essentially mapping out the relationship between bind values and the appropriate execution plan. After this initial period, the database eventually reaches a steady state. Executing with a new bind value results in picking the best child cursor in the cache, without requiring a hard parse. Thus, the number of parses does *not* scale with the number of different bind values.

Example 20-15 Bind-Aware Cursors

This example continues the example in "Bind-Sensitive Cursors". The following code issues a second query `employees` with the bind variable set to `50`:

```
EXEC :dept_id := 50;
SELECT COUNT(*), MAX(employee_id) FROM hr.employees WHERE department_id
= :dept_id;

  COUNT(*) MAX(EMPLOYEE_ID)
---------- ---------------
    100045           100999
```

During the first two executions, the database was monitoring the behavior of the queries, and determined that the different bind values caused the queries to differ significantly in cardinality. Based on this difference, the database adapts its behavior so that the same plan is not always shared for this query. Thus, the optimizer generates a new plan based on the current bind value, which is `50`:

```
SQL> SELECT * FROM TABLE(DBMS_XPLAN.DISPLAY_CURSOR);

PLAN_TABLE_OUTPUT
------------------------------------
SQL_ID    a9upgaqqj7bn5, child number 1
------------------------------------
SELECT COUNT(*), MAX(employee_id) FROM hr.employees WHERE department_id
= :dept_id

Plan hash value: 1756381138

-------------------------------------------------------------------------------
| Id  | Operation          | Name      | Rows  | Bytes | Cost(%CPU)| Time     |
-------------------------------------------------------------------------------
|   0 | SELECT STATEMENT   |           |       |       |254 (100)|          |
|   1 |  SORT AGGREGATE    |           |     1 |     8 |         |          |
|* 2 |   TABLE ACCESS FULL| EMPLOYEES |  100K | 781K |254   (15)| 00:00:01 |
-------------------------------------------------------------------------------
Predicate Information (identified by operation id):
---------------------------------------------------

   2 - filter("DEPARTMENT_ID"=:DEPT_ID)
```

The following query of `V$SQL` obtains information about the cursor:

```
SELECT SQL_TEXT, CHILD_NUMBER AS CHILD#, EXECUTIONS AS EXEC,
       BUFFER_GETS AS BUFF_GETS, IS_BIND_SENSITIVE AS BIND_SENS,
       IS_BIND_AWARE AS BIND_AWARE, IS_SHAREABLE AS SHAREABLE
FROM   V$SQL
```

```
WHERE  SQL_TEXT LIKE '%mployee%'
AND    SQL_TEXT NOT LIKE '%SQL_TEXT%';

SQL_TEXT               CHILD#  EXEC  BUFF_GETS BIND_SENS BIND_AWARE SHAREABLE
---------------------- ------ ----- ---------- --------- ---------- ---------
SELECT COUNT(*), MAX(e      0     2       1329         Y          N         N
mployee_id) FROM hr.em
ployees WHERE departme
nt_id = :dept_id

SELECT COUNT(*), MAX(e      1     1        800         Y          Y         Y
mployee_id) FROM hr.em
ployees WHERE departme
nt_id = :dept_id
```

The preceding output shows that the database created an additional child cursor (`CHILD#` of `1`). Cursor `0` is now marked as not shareable. Cursor `1` shows a number of buffers gets lower than cursor `0`, and is marked both bind-sensitive and bind-aware. A bind-aware cursor may use different plans for different bind values, depending on the selectivity of the predicates containing the bind variable.

Example 20-16 Bind-Aware Cursors: Choosing the Optimal Plan

This example continues the example in "Example 20-15". The following code executes the same `employees` query with the value of `10`, which has extremely low cardinality (only one row):

```
EXEC :dept_id := 10;
SELECT COUNT(*), MAX(employee_id) FROM hr.employees WHERE department_id
= :dept_id;

  COUNT(*) MAX(EMPLOYEE_ID)
---------- ----------------
         1              200
```

The following output shows that the optimizer picked the best plan, which is an index scan, based on the low cardinality estimate for the current bind value of `10`:

```
SQL> SELECT * from TABLE(DBMS_XPLAN.DISPLAY_CURSOR);

PLAN_TABLE_OUTPUT
------------------------------------
SQL_ID    a9upgaqqj7bn5, child number 2
------------------------------------
select COUNT(*), MAX(employee_id) FROM hr.employees WHERE department_id = :dept_id

Plan hash value: 1642965905

-------------------------------------------------------------------------------------
| Id| Operation                           | Name        |Rows|Bytes|Cost (%CPU)|Time |
-------------------------------------------------------------------------------------
| 0| SELECT STATEMENT                     |             |  |  |2(100)|        |
| 1|  SORT AGGREGATE                      |             |1 |8 |      |        |
| 2|   TABLE ACCESS BY INDEX ROWID BATCHED| EMPLOYEES   |1 |8 |2  (0)|00:00:01|
|*3|    INDEX RANGE SCAN                  | EMP_DEPARTMENT_IX | 1|  |1  (0)|00:00:01|
-------------------------------------------------------------------------------------
```

```
Predicate Information (identified by operation id):
---------------------------------------------------

   3 - access("DEPARTMENT_ID"=:DEPT_ID)
```

The `V$SQL` output now shows that three child cursors exist:

```
SELECT SQL_TEXT, CHILD_NUMBER AS CHILD#, EXECUTIONS AS EXEC,
       BUFFER_GETS AS BUFF_GETS, IS_BIND_SENSITIVE AS BIND_SENS,
       IS_BIND_AWARE AS BIND_AWARE, IS_SHAREABLE AS SHAREABLE
FROM   V$SQL
WHERE  SQL_TEXT LIKE '%mployee%'
AND    SQL_TEXT NOT LIKE '%SQL_TEXT%';

SQL_TEXT               CHILD#  EXEC  BUFF_GETS BIND_SENS BIND_AWARE SHAREABLE
---------------------- ------ ----- ---------- --------- ---------- ---------
SELECT COUNT(*), MAX(e      0     2       1329         Y          N         N
mployee_id) FROM hr.em
ployees WHERE departme
nt_id = :dept_id

SELECT COUNT(*), MAX(e      1     1        800         Y          Y         Y
mployee_id) FROM hr.em
ployees WHERE departme
nt_id = :dept_id

SELECT COUNT(*), MAX(e      2     1          3         Y          Y         Y
mployee_id) FROM hr.em
ployees WHERE departme
nt_id = :dept_id
```

The database discarded the original cursor (`CHILD#` of `0`) when the cursor switched to bind-aware mode. This is a one-time overhead. The database marked cursor `0` as not shareable (`SHAREABLE` is `N`), which means that this cursor is unusable and will be among the first to age out of the cursor cache.

> **See Also:**
>
> *Oracle Database Reference* to learn about `V$SQL`

Cursor Merging

If the optimizer creates a plan for a bind-aware cursor, and if this plan is the same as an existing cursor, then the optimizer can perform **cursor merging**.

In this case, the database merges cursors to save space in the library cache. The database increases the selectivity range for the cursor to include the selectivity of the new bind value.

When a query uses a new bind variable, the optimizer tries to find a cursor that it thinks is a good fit based on similarity in the selectivity of the bind value. If the database cannot find such a cursor, then it creates a new one. If the plan for the new cursor is the same as one of the

existing cursors, then the database merges the two cursors to save space in the library cache. The merge results in the database marking one cursor as not sharable. If the library cache is under space pressure, then the database ages out the non-sharable cursor first.

> **See Also:**
>
> "Example 20-12"

Adaptive Cursor Sharing Views

You can use the `V$` views for adaptive cursor sharing to see selectivity ranges, cursor information (such as whether a cursor is bind-aware or bind-sensitive), and execution statistics.

Specifically, use the following views:

- `V$SQL` shows whether a cursor is bind-sensitive or bind-aware.
- `V$SQL_CS_HISTOGRAM` shows the distribution of the execution count across a three-bucket execution history histogram.
- `V$SQL_CS_SELECTIVITY` shows the selectivity ranges stored for every predicate containing a bind variable if the selectivity was used to check cursor sharing. It contains the text of the predicates, and the low and high values for the selectivity ranges.
- `V$SQL_CS_STATISTICS` summarizes the information that the optimizer uses to determine whether to mark a cursor bind-aware. For a sample of executions, the database tracks the rows processed, buffer gets, and CPU time. The `PEEKED` column shows `YES` when the bind set was used to build the cursor; otherwise, the value is `NO`.

> **See Also:**
>
> *Oracle Database Reference* to learn about `V$SQL` and its related views

Real-World Performance Guidelines for Cursor Sharing

The Real-World Performance team has created guidelines for how to optimize cursor sharing in Oracle database applications.

Develop Applications with Bind Variables for Security and Performance

The Real-World Performance group strongly suggests that all enterprise applications use bind variables.

Oracle Database applications were intended to be written with bind variables. Avoid application designs that result in large numbers of users issuing dynamic, unshared SQL statements. Whenever Oracle Database fails to find a match for a statement in the library cache, it must perform a hard parse. Despite the dangers of developing applications with literals, not all real-world applications use bind variables. Developers sometimes find that it is faster and easier to write programs that use literals. However, decreased development time does not lead to better performance and security after deployment.

Video:

Video

The primary benefits of using bind variables are as follows:

- Resource efficiency

  Compiling a program before every execution does not use resources efficiently, but this is essentially what Oracle Database does when it performs a hard parse. The database server must expend significant CPU and memory to create cursors, generate and evaluate execution plans, and so on. By enabling the database to share cursors, soft parsing consumes far fewer resources. If an application uses literals instead of bind variables, but executes only a few queries each day, then DBAs may not perceive the extra overhead as a performance problem. However, if an application executes hundreds or thousands of queries per second, then the extra resource overhead can easily degrade performance to unacceptable levels. Using bind variables enables the database to perform a hard parse only once, no matter how many times the statement executes.

- Scalability

  When the database performs a hard parse, the database spends more time acquiring and holding latches in the shared pool and library cache. Latches are low-level serialization devices. The longer and more frequently the database latches structures in shared memory, the longer the queue for these latches becomes. When multiple statements share the same execution plan, the requests for latches and the durations of latches go down. This behavior increases scalability.

- Throughput and response time

  When the database avoids constantly reparsing and creating cursors, more of its time is spent in user space. The Real-World Performance group has found that changing literals to use binds often leads to orders of magnitude improvements in throughput and user response time.

  See the following video:

  Video:

  Video

- Security

  The only way to prevent SQL injection attacks is to use bind variables. Malicious users can exploit application that concatenate strings by “injecting” code into the application.

See Also:

Oracle Database PL/SQL Language Reference for an example of an application that fixes a security vulnerability created by literals

Do Not Use CURSOR_SHARING = FORCE as a Permanent Fix

The best practice is to write sharable SQL and use the default of `EXACT` for `CURSOR_SHARING`.

However, for applications with many similar statements, setting `CURSOR_SHARING` to `FORCE` can sometimes significantly improve cursor sharing. The replacement of literals with system-generated bind values can lead to reduced memory usage, faster parses, and reduced latch contention. However, `FORCE` is not meant to be a permanent development solution.
As a general guideline, the Real-World Performance group recommends *against* setting `CURSOR_SHARING` to `FORCE` exception in rare situations, and then only when all of the following conditions are met:

- Statements in the shared pool differ only in the values of literals.
- Response time is suboptimal because of a very high number of library cache misses.
- Your existing code has a serious security and scalability bug—the absence of bind variables—and you need a *temporary* band-aid until the source code can be fixed.
- You set this initialization parameter at the session level and not at the instance level.

Setting `CURSOR_SHARING` to `FORCE` has the following drawbacks:

- It indicates that the application does not use user-defined bind variables, which means that it is open to SQL injection. Setting `CURSOR_SHARING` to `FORCE` does *not* fix SQL injection bugs or render the code any more secure. The database binds values only after any malicious SQL text has already been injected.

  Video:

  Video

- The database must perform extra work during the soft parse to find a similar statement in the shared pool.
- The database removes every literal, which means that it can remove useful information. For example, the database strips out literal values in `SUBSTR` and `TO_DATE` functions. The use of system-generated bind variables where literals are more optimal can have a negative impact on execution plans.
- There is an increase in the maximum lengths (as returned by `DESCRIBE`) of any selected expressions that contain literals in a `SELECT` statement. However, the actual length of the data returned does not change.
- Star transformation is not supported.

See Also:

- "CURSOR_SHARING and Bind Variable Substitution"
- *Oracle Database Reference* to learn about the `CURSOR_SHARING` initialization parameter

Establish Coding Conventions to Increase Cursor Reuse

By default, any variation in the text of two SQL statements prevents the database from sharing a cursor, including the names of bind variables. Also, changes in the size of bind variables can cause cursor mismatches. For this reason, using bind variables in application code is not enough to *guarantee* cursor sharing.

The Real-World Performance group recommends that you standardize spacing and capitalization conventions for SQL statements and PL/SQL blocks. Also establish conventions for the naming and definitions of bind variables. If the database does not share cursors as expected, begin your diagnosis by querying `V$SQL_SHARED_CURSOR`.

Example 20-17 Variations in SQL Text

In this example, an application that uses bind variables executes 7 statements using the same bind variable value, but the statements are not textually identical:

```
VARIABLE emp_id NUMBER
EXEC :emp_id := 101;

SELECT SUM(salary) FROM hr.employees WHERE employee_id < :emp_id;
SELECT SUM(salary) FROM hr.employees WHERE employee_id < :EMP_ID;
SELECT SUM(salary) FROM hr.employees WHERE employee_id < :Emp_Id;
SELECT SUM(salary)  FROM hr.employees WHERE employee_id < :emp_id;
select sum(salary) from hr.employees where employee_id < :emp_id;
Select sum(salary) From hr.employees Where employee_id < :emp_id;
Select sum(salary) From hr.employees Where employee_id< :emp_id;
```

A query of `V$SQLAREA` shows that no cursor sharing occurred:

```
COL SQL_TEXT FORMAT a35
SELECT SQL_TEXT, SQL_ID, VERSION_COUNT, HASH_VALUE
FROM   V$SQLAREA
WHERE  SQL_TEXT LIKE '%mployee%'
AND    SQL_TEXT NOT LIKE '%SQL_TEXT%';

SQL_TEXT                            SQL_ID        VERSION_COUNT HASH_VALUE
----------------------------------- ------------- ------------- ----------
SELECT SUM(salary) FROM hr.employee bkrfu3ggu5315             1 3751971877
s WHERE employee_id < :EMP_ID
SELECT SUM(salary) FROM hr.employee 70mdtwh7xj9gv             1  265856507
s WHERE employee_id < :Emp_Id
Select sum(salary) From hr.employee 18tt4ny9u5wkt             1 2476929625
s Where employee_id< :emp_id
SELECT SUM(salary)  FROM hr.employe b6b21tbyaf8aq             1 4238811478
es WHERE employee_id < :emp_id
SELECT SUM(salary) FROM hr.employee 4318cbskba8yh             1  615850960
s WHERE employee_id < :emp_id
select sum(salary) from hr.employee 633zpx3xm71kj             1 4214457937
s where employee_id < :emp_id
Select sum(salary) From hr.employee 1mqbbbnsrrw08             1  830205960
s Where employee_id < :emp_id
```

```
7 rows selected.
```

Example 20-18 Bind Length Mismatch

The following code defines a bind variable with different lengths, and then executes textually identical statements with the same bind values:

```
VARIABLE lname VARCHAR2(20)
EXEC :lname := 'Taylor';
SELECT SUM(salary) FROM hr.employees WHERE last_name = :lname;
VARIABLE lname VARCHAR2(100)
EXEC :lname := 'Taylor';
SELECT SUM(salary) FROM hr.employees WHERE last_name = :lname;
```

The following query shows that the database did not share the cursor:

```
COL SQL_TEXT FORMAT a35
SELECT SQL_TEXT, SQL_ID, VERSION_COUNT, HASH_VALUE
FROM   V$SQLAREA
WHERE  SQL_TEXT LIKE '%mployee%'
AND    SQL_TEXT NOT LIKE '%SQL_TEXT%';

SQL_TEXT                            SQL_ID           VERSION_COUNT HASH_VALUE
----------------------------------- ------------- ------------- ----------
SELECT SUM(salary) FROM hr.employee buh8j4557r0h1              2 1249608193
s WHERE last_name = :lname
```

The reason is because of the bind lengths:

```
COL BIND_LENGTH_UPGRADEABLE FORMAT a15
SELECT s.SQL_TEXT, s.CHILD_NUMBER,
       c.BIND_LENGTH_UPGRADEABLE
FROM   V$SQL s, V$SQL_SHARED_CURSOR c
WHERE  SQL_TEXT LIKE '%employee%'
AND    SQL_TEXT NOT LIKE '%SQL_TEXT%'
AND    s.CHILD_ADDRESS = c.CHILD_ADDRESS;

SQL_TEXT                            CHILD_NUMBER BIND_LENGTH_UPG
----------------------------------- ------------ ---------------
SELECT SUM(salary) FROM hr.employee            0 N
s WHERE last_name = :lname
SELECT SUM(salary) FROM hr.employee            1 Y
s WHERE last_name = :lname
```

Minimize Session-Level Changes to the Optimizer Environment

A best practice is to prevent users of the application from changing the optimization approach and goal for their individual sessions. Any changes to the optimizer environment can prevent otherwise identical statements from sharing cursors.

Example 20-19 Environment Mismatches

This example shows two textually identical statements that nevertheless do not share a cursor:

```
VARIABLE emp_id NUMBER

EXEC :emp_id := 110;

ALTER SESSION SET OPTIMIZER_MODE = FIRST_ROWS;
SELECT salary FROM hr.employees WHERE employee_id < :emp_id;
ALTER SESSION SET OPTIMIZER_MODE = ALL_ROWS;
SELECT salary FROM hr.employees WHERE employee_id < :emp_id;
```

A query of `V$SQL_SHARED_CURSOR` shows a mismatch in the optimizer modes:

```
SELECT S.SQL_TEXT, S.CHILD_NUMBER, s.CHILD_ADDRESS,
       C.OPTIMIZER_MODE_MISMATCH
FROM   V$SQL S, V$SQL_SHARED_CURSOR C
WHERE  SQL_TEXT LIKE '%employee%'
AND    SQL_TEXT NOT LIKE '%SQL_TEXT%'
AND S.CHILD_ADDRESS = C.CHILD_ADDRESS;

SQL_TEXT                          CHILD_NUMBER CHILD_ADDRESS    O
--------------------------------- ------------ ---------------- -
SELECT salary FROM hr.employees WHE            0 0000000080293040 N
RE employee_id < :emp_id
SELECT salary FROM hr.employees WHE            1 000000008644E888 Y
RE employee_id < :emp_id
```