Boost SQL Performance with cursor_sharing

An Oracle Technical White Paper

July 2001

EXECUTIVE OVERVIEW

This paper describes enhancements to the cursor sharing infrastructure, added in Oracle8*i* Release 2, and Oracle9*i*. These enhancements are controlled by a new parameter **cursor_sharing**.

The goal of **cursor_sharing** is to improve server performance for applications that don't use bind variables everywhere.

NEED FOR CURSOR_SHARING

This section explains why applications not using bind variables run into performance problems.

Applications execute similar SQL statements repeatedly

Applications that use an Oracle database to manage their data must access/modify it using SQL statements. These SQL statements are either directly issued by an application through OCI, OCCI, JDBC, PL/SQL etc., or indirectly issued through other utilities or libraries (e.g. dbms_sql).

An application typically provides a fixed collection of functions to the end user depending on the type of application, e.g. an HR application might provide functionality like adding a new employee, modifying personal information about an employee, etc. Every such functionality eventually accesses and/or modifies data using SQL. Since applications repeatedly execute such functionality, an application's interaction with an Oracle database consists of repeated execution of similar SQL statements.

Steps involved in executing SQL

To execute a SQL statement, a client program can go through a few different interfaces. For example, through OCI a client creates a statement handle, then prepares the statement, binds, defines, and executes the statement handle. Or, a SQL statement can be executed through a PL/SQL procedure by directly embedding it in the procedure text.

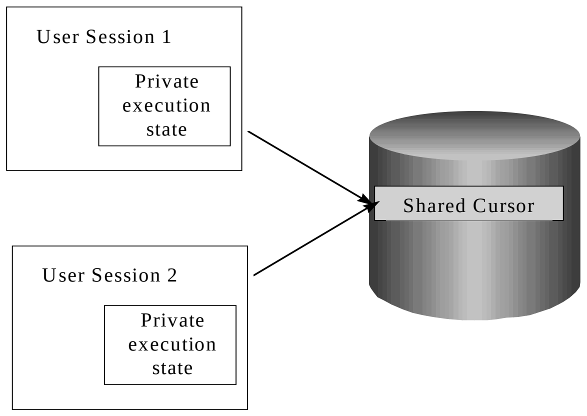
Regardless of the client interface, an Oracle database always goes through some fixed steps (by default):

1. open a user cursor - A user cursor is a handle to all other user state associated with any SQL statement like execution memory, reference to the shared cursor, current state of the user cursor, etc.
2. parse a SQL statement into the open user cursor - Doing this associates the SQL statement with the user cursor; it also creates a shared cursor, which corresponds to the parsed form of the SQL statement. The shared cursor may also be typechecked and optimized as a part of the parse, in some cases. The process of parsing, typechecking and optimizing a SQL statement is typically very resource intensive in terms of CPU, memory and latch contention.
3. bind variables, if necessary - This provides Oracle with the necessary information about the type, size, value, etc. of the bind variables in the statement.
4. typecheck and optimize the shared cursor, if not already done
5. execute the user cursor - This step does the actual work of executing the statement, and will use up CPU and session memory depending on the complexity of the statement.

Note that parse, typecheck and optimization (collectively called compilation in this document) comprise the bulk of the overhead in executing a SQL statement, and can limit capacity and scalability of the database.

Shared cursors

Given that a typical application repeatedly executes similar statements, a lot of optimizations in the Oracle database for SQL processing target repeated executions. The most important optimization is shared cursors, which tries to remove the overhead of compilation for an average execution, by sharing the result of compilation between different executions of the same statement (either happening concurrently, or at very different times). The picture below illustrates how this is done.



To enable sharing cursors, Oracle splits up the statement execution state into a shared cursor, and a per-execution instantiation. The shared cursor is the result of compilation and contains the execution plan; it is cached in the shared pool. Each session executing the statement has its private copy of per-execution state like the user cursor, values of runtime variables, etc.

In the parse step (step 2 mentioned earlier), Oracle first searches for an existing shared cursor that can be shared by the user session. Oracle breaks down this search into two steps: indexing based on the SQL text to find cursors built for the same SQL text, and selecting the right cursor based on other criteria like optimizer modes, base objects accessed, etc. If a sharable cursor is found then no compilation needs to happen, and the process is called a soft parse. Else, a new shared cursor is built by compiling the SQL statement, and the process is called a hard parse.

When most of the statements issued by an application can share the same set of cursors, most of the parses become soft parses, and improve the database server capacity/throughput (by reducing memory and CPU usage), response-time (by reducing the time taken by the parse phase), and scalability (by reducing latch contention).

Why cursors are not shared

Assuming that other factors like configurable instance/session/transaction level parameters are the same, cursors for two statements S1 and S2 can theoretically be shared if they perform the same operations on the same underlying set of rows/objects, using the same plan. This can be very hard and computation intensive to figure out, potentially destroying the benefits of sharing cursors in the first place! Hence, Oracle's cursor sharing criteria don't share cursors in all the possible scenarios, but they are designed to be efficient, without losing out on most of the common cases. Until 8i Release 2, two statements S1 and S2 could share the same cursor if both S1 and S2 were textually identical, and a few other conditions were met (any object names in the statements translate to the same base objects, optimizer modes for the sessions issuing the statements match, etc.)

This causes a cursor sharing problem when applications use literals instead of bind variables in statements. Such applications end up producing statements that differ in some of the literals even when the rest of the statement text is identical. For example, an application that does not use bind variables, might issue the following two statements, at different times or from different sessions:

INSERT INTO T VALUES(1, 'foo', 4)

INSERT INTO T VALUES(2, 'bar', 7)

Since the two statements aren't textually identical, they end up building separate cursors.

There are various reasons why some applications don't use bind variables:

- it's just easier to write SQL statements with literals, especially with some tools
- older Oracle releases didn't support bind variables (at least there was no cursor sharing advantage to using them until Oracle7), and it requires some work to retrofit bind variables to existing applications
- all database vendors don't support bind variables, or even if they do, the syntax varies; hence applications lose interoperability with other databases by using Oracle-only syntax/features
- If a statement uses bind variables, then it always uses the same plan. This can be a problem if the optimal plans for different bind values can be very different. For example, consider the following statements:

  *SELECT * FROM T1, T2 WHERE (T1.N <= 100) AND (T1.N1 = T2.N2)*

  *SELECT * FROM T1, T2 WHERE (T1.N <= 500) AND (T1.N1 = T2.N2)*

These two statements can have different optimal plans, depending on the distribution of values in the column N. Thus, using a bind variable, producing:

*SELECT * FROM T1, T2 WHERE (T1.N <= :X) AND (T1.N1 = T2.N2)*

will cause the plan to be sub-optimal for some values of the bind variable. This can force applications (especially, in a DSS environment) to use literals instead of bind variables.

CONCEPTS

This section describes some necessary concepts, before going into the solution.

Similar statements

A set of statements will be called similar if any two statements in the set differ only in the literals.

This is a purely syntactic criterion.

Example: The following statements are similar.

INSERT INTO T VALUES(1, 'foo', 4)

INSERT INTO T VALUES(2, 'bar', 7)

INSERT INTO T VALUES(5, 'alpha', 11)

INSERT INTO T VALUES(10, 'kappa', 17)

Optimally sharable statements

Similar statements may or may not have the same execution plans. For example, the following two statements will have the same execution plans:

INSERT INTO T VALUES(1, 'foo', 4)

INSERT INTO T VALUES(2, 'bar', 7)

Such statements will be called optimally sharable statements, in this document. Thus:

Optimally sharable statements are similar statements that have the same optimal plan.

This also implies that optimally **sharable** statements can share the same cursor, without any impact on the execution costs.

Sub-optimally sharable **statements**

On the other hand, the following two statements:

*SELECT * FROM T1, T2 WHERE (T1.N <= 100) AND (T1.N1 = T2.N2)*

*SELECT * FROM T1, T2 WHERE (T1.N <= 500) AND (T1.N1 = T2.N2)*

can have different optimal plans, depending on the rows that satisfy (N <= 100) and (N <= 500), the distribution of values in column N, the availability of an index on N, N1 or N2, etc. For instance, the first statement may use an index on T1 and the second statement may do a full table scan on T1. Or, the first statement may do a hash join and the second statement may do a nested loop join. Such statements will be referred to as suboptimally **sharable** statements. Thus:

Sub-optimally sharable statements are similar statements that can have different optimal plans.

This also implies that if sub-optimally **sharable** statements share the same cursor, then there may be a penalty in terms of execution costs.

Optimally sharable **vs. suboptimally** sharable **statements**

The distinction between optimally **sharable** and suboptimally **sharable** statements is not purely syntactic. It depends on factors like:

- the position of literals in the statement (e.g. in the VALUES clause, or the WHERE clause)
- the access paths available (e.g. presence of an index)
- the data distribution (statistics) and its availability, if a literal occurs in a predicate involving a column (e.g. N <= 100 with column statistics available on N)
- algorithms used by the optimizer, to exploit literals (e.g. constant folding, or partition pruning)

Non-sharable **statements**

There are cases where similar statements cannot share the same cursor, because using the same cursor would produce incorrect results. These are similar statements that mean different things, or do something very different during execution. The following statements illustrate the point:

*SELECT * FROM T ORDER BY 1, 4;*

*SELECT * FROM T ORDER BY 2, 3;*

In the above example, the literals 1, 2, 3, and 4 refer to items in the select list. Such statements will be called non-**sharable** statements. Thus:

Non-sharable statements are similar statements that cannot share the same execution plan.

The important point to note here is: if two non-**sharable** statements share the same cursor, one of them will get wrong results.

THE SOLUTION

This section describes the solution provided through **cursor_sharing**.

Overview

The new parameter **cursor_sharing** allows sharing cursors for similar statements whenever possible. Depending on the value of the parameter, similar statements can be always forced to share the same cursor (potentially using suboptimal plans), or can share the same cursor without compromising the optimality of the underlying plan.

With **cursor_sharing** set to either SIMILAR or FORCE, Oracle first searches for a cursor with exactly the same statement text. If such a cursor is not found, Oracle searches for a cursor with a similar statement text.

Usage

Parameter: cursor_sharing

A new dynamic parameter **cursor_sharing** has been introduced starting with 8i Release 2. In 8i, the parameter can have two possible values: FORCE and EXACT. Starting with 9i, a new value SIMILAR has been added.

The default value is EXACT. It only allows statements with the exact same text to share a cursor. This is the behavior of earlier releases.

The value SIMILAR causes similar statements to share the same cursors, without compromising execution plans, i.e. only optimally **sharable** statements share cursors.

Setting the parameter to FORCE will force Oracle to share cursors for similar statements, at the risk of suboptimal plans, i.e. both optimally **sharable** and suboptimally sharable statements can share the same cursor. The parameter should be set to FORCE only when the risk of suboptimal plans is outweighed by the improvements in cursor sharing, e.g. if there are severe performance problems caused by too many hard parses of suboptimally shareable statements.

SQL statements

A new hint CURSOR_SHARING_EXACT is allowed in SQL statements to control cursor sharing at the statement level. The hint behaves similar to the initialization parameter, cursor_sharing set to EXACT, and overrides the existing default behavior based on any setting of the parameter, i.e. it causes the statement to share a cursor built for a statement with an exact match.

ALTER SYSTEM and ALTER SESSION commands allow the new parameter cursor_sharing to be set or changed. The syntax is as follows:

ALTER SYSTEM SET **cursor_sharing** = {FORCE | SIMILAR | EXACT}

ALTER SESSION SET **cursor_sharing** = {FORCE | SIMILAR | EXACT}

Dynamic views

The following four dynamic views show information about bind variables:

- GV$SQL_BIND_METADATA
- V$SQL_BIND_METADATA
- GV$SQL_BIND_DATA
- V$SQL_BIND_DATA.

These views will also contain information about internal bind variables. Internal bind variables can be distinguished from user bind variables, based on the value of the column SHARED_FLAG2 in [G]V$SQL_BIND_DATA, by looking at flag value 256.

To see only the rows corresponding to internal binds, a user can issue the following statement:

*SELECT * FROM V$SQL_BIND_DATA WHERE*
BITAND(SHARED_FLAG2, 256) = 256

Key benefits and tradeoffs

Consider an application not using bind variables. Such an application will issue similar statements repeatedly, and most such executions will incur a hard parse.

A typical application not using binds can be expected to have all categories of statements: optimally shareable, suboptimally shareable and non-shareable. For optimally shareable statements, sharing cursors is clearly an improvement; non-shareable statements cannot share the same cursor.

There is no simple answer for suboptimally shareable statements: sharing cursors vs. getting the optimal plan represents a tradeoff that needs to be decided by weighing the acuteness of the hard parsing overhead on the system vs. the deterioration that can be caused by forcing such statements to use the same plan. Thus, the right answer will vary depending on the system load, application characteristics, resource constraints, etc. That's the reason Oracle leaves this decision up to the user, by exposing two different values for **cursor_sharing**: SIMILAR and FORCE. SIMILAR is a more conservative choice, which causes only optimally shareable statements to share cursors. With FORCE, both optimally shareable and suboptimally shareable statements are forced to share cursors, and the outcome is less predictable, since cursors may be shared but execution plans may also deteriorate. Hence, using FORCE makes sense in situations where performance is significantly affected due to hard parsing, and there is a very high percentage of suboptimally shareable statements. Another way to think about this is: it is better to try SIMILAR before resorting to FORCE.

When similar statements share cursors as a result of **cursor_sharing**, hard parses get converted to soft parses. Note that these soft parses are a little more expensive than a soft parse for an application already using bind variables, due to the additional cost of determining the similarity of the statement (done internally by replacing literals with bind variables). However, the net savings in CPU, memory and latch contention will still be considerable.

Note that with **cursor_sharing**, Oracle still searches for an exact match first. Only when a cursor with exactly the same statement text is not found, Oracle searches for a cursor with a similar statement text. This is done to ensure that when the same SQL text is being issued with no hard coded literals, there is no impact on performance.

Since replacement of literals is done before looking for a cursor, other Oracle optimizations like session_cached_cursors, and cursor_space_for_time can be advantageously combined with **cursor_sharing**. For example, with both cursor_sharing and session_cached_cursors set to a reasonable value, a similar statement will be able to use a cached open cursor, after literals are replaced with internal bind variables.

A summary of the key benefits follows:

- No application change is needed.
- There is no negative impact on those statements, that already use bind variables.
- With SIMILAR, cursors are shared more often without impacting execution plans.
- All similar statements can be forced to share cursors with FORCE, as a final resort.

Caveats

Mixed statements

Mixed statements are statements that have both bind variables and hard coded literals. For example:

INSERT INTO T VALUES(5, 'alpha', :X)

Similar statements that are mixed don't share cursors through **cursor_sharing**, if they are issued by a client using Oracle7 OCI; they share cursors with later versions (starting with Oracle8 OCI). In particular, this also applies to SQL issued from PL/SQL stored procedures in the server, since PL/SQL in the server uses an older client interface.

Static SQL through PL/SQL

Cursor_sharing does not have any effect on static (embedded) SQL in PL/SQL.

Stored outlines

Any stored outlines created without **cursor_sharing** set to FORCE or SIMILAR, don't get picked up when **cursor_sharing** is set (to FORCE or SIMILAR). That's because stored outlines are indexed by the SQL text, and the current implementation of **cursor_sharing** modifies the statement text. To use stored outlines with **cursor_sharing**, they must be recreated using the create_stored_outlines parameter (and NOT using the create outline statement).

Overhead

There is an overhead involved with FORCE or SIMILAR, which consists of searching for a cursor built for a similar statement. As mentioned earlier, this comprises of:

- searching for a cursor with the original statement text
- replacing literals with internal bind variables, and searching based on the new text

This overhead will not matter when **cursor_sharing** works, since a large percentage of hard parses will be replaced by soft parses, which are slightly more expensive. However, when there isn't a significant increase in cursor sharing, these overheads can negatively impact performance. There are three scenarios when this can happen:

a. an application not using binds, issuing the same statements, and not having any similar statements

   This can happen if an application always executes the same statements with the same literals hard coded in them. Such an application does soft parses by default, and setting cursor_sharing to FORCE or SIMILAR, will make the soft parses more expensive.

   There is a trick that can be used in the case of such an application: cursor_sharing can be set to FORCE or SIMILAR, after the shared pool is warmed up, i.e. after all the statements that have the same literals are already compiled. That way, Oracle will always find cursors for those statements right away, avoiding the extra overhead.

This can be particularly useful, if there are some statements in an application that always use the same literals, and others that keep changing literals.

b. an application issuing structurally different statements, and thus not having any similar statements

   Such an application does hard parses by default, and setting cursor_sharing to FORCE or SIMILAR, will make the hard parses slightly more expensive.

c. an application not using binds, with most of the similar statements being suboptimally shareable, using cursor_sharing set to SIMILAR

   Such an application does hard parses by default, and mostly soft parses with cursor_sharing set to FORCE. Setting cursor_sharing to SIMILAR, will make the hard parses slightly more expensive.

Using FORCE

Using FORCE can potentially cause a very bad execution plan to be used. The difference between a good plan and a bad plan can be crucial in some situations, e.g. a DSS environment. Hence, Oracle does not recommend using FORCE is such scenarios.

When should you use cursor_sharing?

This section makes some recommendations on using cursor_sharing.

Using cursor_sharing=SIMILAR

As mentioned earlier, **cursor_sharing** does not hurt the performance of applications written using bind variables. Setting **cursor_sharing** to SIMILAR improves performance of applications not using bind variables, in most cases (two exceptions mentioned in the previous section). Hence, **cursor_sharing** can be set to SIMILAR with minimal risk, in case of performance problems with applications that don't use bind variables everywhere. Parts of the application that use bind variables continue to share cursors, and those parts that issue statements with hard coded literals benefit from some cursor sharing.

Whether **cursor_sharing**=SIMILAR will improve performance depends on the answers to the following questions:

- Is performance bad due to a very high number of hard parses?

  This can be inferred by monitoring several metrics like the average number of hard parses, the number of parses/number of executions, average response times, wait events for sessions, etc.

- Are there lots of similar statements in the shared pool, with hard coded literals?

  This can be checked through dynamic views like v$sql or v$sqlarea.

If the answer to both the above questions is positive, then it is very likely that **cursor_sharing**=SIMILAR will improve performance.

Using cursor_sharing=FORCE

Using **cursor_sharing**=FORCE can be considered in the following cases:

- The percentage of suboptimally shareable statements is very high, so that SIMILAR isn't useful enough.

  There is no easy way to find the percentage of suboptimally shareable statements, short of examining all the statements. The only other way of determining this might be to try setting cursor_sharing to SIMILAR; if hard parsing problems due to similar statements not sharing cursors persist, then there are many suboptimally shareable statements, and FORCE is the only solution.

- The application has hard coded literals, and there is little deterioration in execution time by forcing similar statements to use the same cursor.

  Again, this is not easy to tell. When an application uses bind variables, it implicitly makes this assumption; to some extent such an application ensures that the plan works reasonably well for all values of bind variables by structuring the statement in certain ways, using optimizer hints, creating the right indexes, etc. There is some likelihood of the reverse working; viz. for an application not using binds, using the rule-based optimizer, with indexes on most of the interesting columns, forcing similar statements to share the same cursor may produce reasonable execution times. Nevertheless, this is a risky approach that may remove the hard parsing bottleneck, and introduce a new problem of tuning SQL.

Thus, it is useful to think of FORCE as a last resort, when using SIMILAR doesn't help.

When should you not use cursor_sharing?

There are three scenarios, mentioned earlier (in "**Caveats**"), when using **cursor_sharing** can hurt. These are cases where there aren't any similar statements that can share cursors with a certain value of **cursor_sharing**, and using it only adds to the parsing overhead.

The other thing to keep in mind is the following: **cursor_sharing** provides a solution for the DBA faced with an application using literals. It is not a substitute, however, for writing applications using bind variables, which also allows exploiting other optimizations provided by Oracle. For example, an application can keep frequently executed statements parsed in some open cursors, and only issue executes on them, as needed. Such optimizations are based on deeper application knowledge, and can't be matched by using **cursor_sharing**.

CONCLUSION

The use of **cursor_sharing** can solve performance problems caused by hard parsing, in case of applications not using bind variables everywhere. The parameter should be set judiciously, based on application and database characteristics and system resources.

APPENDIX: SOME PERFORMANCE MEASUREMENTS

This section describes an experiment done with Oracle8*i* Release 2, to validate **cursor_sharing**.

Description

The purpose of this experiment was to do a basic validation. The maximum throughput of the server was measured by repeatedly issuing a single statement by a few clients (order of 100s). The experiment was done 3 times, with the following characteristics:

1. using only bind variables

   The purpose was twofold: to establish a baseline, and to ensure that using cursor_sharing did not impact the performance for such a statement.

2. using only literals, with each iteration having a different literal

   This scenario issued similar statements, and was expected to benefit the most by cursor_sharing.

3. using literals with the same literal with each iteration.

   This scenario wasn't expected to benefit from cursor_sharing; on the contrary, it was expected to deteriorate. The reason for testing it out was to measure the overhead of cursor_sharing on a soft parse.

Only the parse throughput (number of parses per second) was measured in each case.

Results

The results are shown below.

Type	*cursor_sharing=EXACT*	*cursor_sharing=FORCE*
Binds only	2650	2650
Similar statements	860	2500
1 statement with literals	3300	2600

Oracle8.1.7 Parse Throughput numbers with cursor_sharing (parses/sec)

ORACLE

Boost SQL Performance with cursor_sharing
July 2001

Author: Sanjay Kaluskar
Contributing Authors: Brajesh Goyal, Namit Jain

Oracle Corporation
World Headquarters
500 Oracle Parkway
Redwood Shores, CA 94065
U.S.A.

Worldwide Inquiries:
Phone: +1.650.506.7000
Fax: +1.650.506.7200
www.oracle.com