BIND VARIABLES AND CURSOR SHARING – NEW DIRECTIONS IN ORACLE9I

Bjørn Engsig, Miracle A/S

CONTENTS

SUMMARY

Good application programming practices have always been crucial for applications to perform and scale properly, and particularly for applications accessing a common centralized repository that gives access to hundreds or thousands of users concurrently, such as the Oracle database, understanding of the processing that takes place is important.

The database work that needs to be done by the application is expressed using the SQL language, and the Oracle database has been designed with features that allow sharing of information between multiple identical SQL statements executed concurrently by many users. A set of Oracle startup parameters control the behaviour of this, and also allowing sharing of SQL statements that are not fully identical.

The primary focus of this paper is to explain how Oracle processes SQL statements, thereby allowing the application developer to design and code well performing and scaling applications. Additionally, the database administrator will learn how Oracle can be configured to allow some non-optimally written applications to execute with reasonable performance.

SQL PROCESSING IN ORACLE

When Oracle processes SQL statements, there are generally four or five steps involved, depending on the type of SQL statement:

1. A *hard parse* step, where the Oracle engine parses the SQL statement, verifies its syntactical correctness and verifies the existence of the tables and columns named.
2. A *soft parse* step, where the hard parsed SQL statement is associated with the specific session and the access rights verified.
3. An optimization step, where Oracle finds the best way to process the SQL statement using information about the size of the various objects (tables, indexes, etc.), that will be accessed.
4. An *execution* step, which uses the result of the parse and optimization steps to actually access data.
5. For queries (i.e. select statements), the execution is followed by a fetch step, which actually returns the rows of the query to the application.

Understanding of Oracle SQL processing gives the application programmers the possibility to write fully scalable applications. In particular, avoiding hard and soft parses will greatly reduce CPU overhead and increase sharing of memory.

The first three steps, which may take considerable time and resources, are normally considered overhead, as no actual processing of user data takes place. Hence, applications should be written to minimize the amount of time spent during these steps. The most efficient way to do this is to avoid the parse and optimization steps as much as possible.

The major component of Oracle that copes with processing of SQL is the library cache, in which currently executing SQL statements are cached. During the parse step[1], Oracle first verifies whether or not the SQL statement is in the library cache. If it is, only little further processing is necessary, such as verification of access rights and association with the session environment, and the parse is a soft parse. If not found in the library cache, the statement will need to be hard parsed, checked for syntax errors, checked for correctness of table- and columnnames, etc. The former type of parse, the soft parse, is considerably faster than the hard parse. Hence applications should in particular avoid hard parses.

During the first execute of a SQL statement, the optimization takes place, i.e. Oracle finds the best path to access data[2]. Finally, the SQL statement is executed, and for queries, the resulting rows are then fetched. Once a statement is parsed and executed, it can efficiently be executed again, completely avoiding the parse and optimization steps.

The method used by Oracle to verify if a SQL statement is already found in the library cache is a hash lookup, followed by simply doing a string comparison. Hence, only completely identical SQL statements will be considered identical.

USING BIND VARIABLES

As is explained in the previous section, only completely identical SQL statements will be recognized as such when the library cache is consulted. Hence, the two statements:

---

[1] From an application programming point of view, there is a single *parse* operation; the distinction between the hard and soft parse is done in the Oracle Server depending on the circumstances as described.

[2] In some cases, the optimization is actually performed during the parse step. This is, however, irrelevant for the discussion in this paper.

```
select * from emp where empno=1234
```

and

```
select * from emp where empno=5678
```

are considered different, and executing these two will cause two hard parses and two separate entries in the library cache. It is therefore highly recommended to use bind variables in the SQL statement that are bound to values in the application program, and whose actual value is only filled in at statement execute time. The statement above would then be

```
select * from emp where empno=:x
```

which can be executed multiple times with different actual values for the bind variable `:x`. A well-designed application will do a single parse of this statement and repeat the execute as often as necessary with different actual values for the bind variable in the application program. This approach not only makes a single program perform better, it also ensures scalability by having multiple concurrently running programs share the same copy of the SQL statement. Oracle has always recommended this approach to application programming. (See, however, below in the section on special considerations for recommendations for data warehouse type applications).

SUPPORT FOR APPLICATIONS USING LITERALS

The values 1234 or 5678 in the two examples above are constants or literals, and using them in SQL statements causes overhead with parsing and imply that the SQL statements are not shareable in the library cache. Many applications have not, however, been developed with sufficient attention to this, and Oracle therefore has a method to reduce the impact of such programming. The feature will allow sharing of cursor information in the library cache, for SQL statements that actually are not identical, but only differ in literals. This is implemented by doing a simple scan of the SQL statement, which will replace any literals found by a bind variable. This scan is done before the actual parse step, and when the server is asked to parse the two statements above, they will therefore in both cases be replaced by a statement like:

```
select * from emp where empno=:SYS_B_0
```

In this statement, the literal has been replaced by the bind variable, and the statement will be shared and only incur the overhead of a hard parse once. Therefore, even applications that do not themselves use bind variables, can get the benefit of doing so. This is controlled with the startup (init.ora) parameter **cursor_sharing**, which can be set to any of three values:

exact	Causes the mechanism not be used, i.e. no cursor sharing for statements with different literals. This is the default value.
force	Causes unconditional sharing of SQL statements that only differ en literals.
similar	Causes cursor sharing to take place when this is known not to have any impact on optimization. This value is not available in Oracle8i.

The parameter may be set globally either in the startup file or by an alter system command, or it can be set for an

individual session with an alter session command. The default value is **exact**, although this may change in a future release of Oracle

HANDLING OF CURSORS

A cursor is a handle by which applications execute SQL statements. An application may open as many cursors as necessary (within limits of memory and the **open_cursors** startup parameter), and the cursors are subsequently associated with SQL statements. For the most efficient execution of SQL statements, applications should open as many cursors as is necessary, so that frequently executed statements are always readily available for execution. The application does so by opening multiple cursors and parsing these SQL statements with bind variables. Next, when actual data are available, only the execute step (and fetch step for queries) needs to be performed.

Even though Oracle has recommended this approach, many applications fail to follow it, or only follow it rarely. Hence, applications would either open and close cursors continuously or they would reuse a single cursor with different SQL statements, typically causing at least a soft parse; in many cases a hard parse. The Oracle startup parameter **session_cached_cursors** can be used to modify this behavior. The parameter can be set to an integer value and setting it will imply some of the server side context associated with SQL statements will be kept, even when the client side performs a cursor close or reuses a cursor with a new SQL statement. The value of the parameter tells how many SQL statements (per session) should be treated like this.

For well written applications, where the execute step is repeated for one or more cursors, that are opened and have SQL statements parsed only once, there is some processing taking place to allocate runtime memory, i.e. memory that is needed during the execute phase only. By setting the **cursor_space_for_time** parameter to **true** (the default being **false**), allocation of this memory is done only once at the first execute, which improves performance at the expense of using more memory. Due to the overhead in memory usage, this parameter should only be used if your application has a well-constrained set of SQL that fits in the shared pool.

RESULTS AND BEST PRACTICES

In the context of this white paper, applications that frequently execute similar SQL statements (i.e. SQL statements that are only different in actual values) are put into the three categories:

1.	Applications not using bind variables. This type of programming is frequently seen, when application developers have not been aware of the importance of reducing the number of parses, or when applications have been migrated from other environments.
2a.	Applications doing all steps: cursor open, parse with bind variables, execute (fetch) and close for each individual SQL statement. This is the behavior Oracle Forms when blocks are entered and exited, and in newer code, native dynamic SQL of the PL/SQL language uses this type of programming. The benefit of using bind variables in this way has not been obvious for OCI and Precompiler programmers, and is rarely therefore seen in these environment.
2b.	Applications opening cursors once, and repeatedly doing the subsequent steps: parse with bind variables, execute (and fetch) for each SQL statement. This type of application is often seen when using the Oracle Precompilers, with the options HOLD_CURSOR=NO and RELEASE_CURSOR=NO, which is the default, or when using the DBMS_SQL package from PL/SQL.
3.	Applications opening a cursor and parsing with bind variables only once for each distinct SQL statement, followed by repeated executes (and fetches for queries). This is the most efficient way to write applications repeatedly executing the same SQL statements, and it is typically seen in well designed applications using Oracle Call Interface or the Precompilers. Specifically for the Precompilers, using the options HOLD_CURSOR=YES, RELEASE_CURSOR=NO for specific SQL statements or generally setting MAXOPENCURSORS sufficiently high causes this behavior for implicit cursors. PL/SQL using cursor for loops or explicit cursors with static SQL are also using this approach.

The categories 2a and 2b are for all practical purposes identical, as the actual open and close of cursors has very little overhead – the real overhead is in the parse.

Application performance can generally be viewed as response time performance, i.e. the actual time taken for each execution of a SQL statement, and as throughput performance or scalability, i.e. the ability to run multiple copies of the application concurrently. Both of these performance metrics have been tested with the different type of application categories, and with different scenarios of the **cursor_sharing** and **session_cached_cursors** parameters. Advises for each of the three categories of applications listed above in combination with the three startup parameters mentioned are shown in the table below. The appendix contains some actual results of tests that have been designed specifically to demonstrate this.

Application category	General behavior	**Cursor_sharing force** or **similar**	**session_cached_-cursors** sufficiently high	**cursor_space_-for_time true**
1. Not using bind variables at all.	Generally poor response time and scalability. Is, however, recommended for DSS applications, that have little or no repetition of SQL statements	Good improvement of response time and scalability; although not quite as good as writing applications using category 2 or 3.	Minor extra benefit	no effect
2. Parse with bind variables, execute (and fetch) for each SQL statement execution.	Response time relatively good due to cursor sharing in library cache, limited scalability, due to repeated open/close and soft parse	No extra benefit	Improves response time and scalability as the server keeps cursors cached.	no effect
3. Cursor open and parse with bind variables once, repeated executes (fetch)	Highest possible response time and scalability	No extra benefit	No extra benefit	Additional improvement to scalability

BEST PRACTICES

The tests (detailed in appendix A) demonstrate that the parameter, **cursor_sharing**, has a good performance impact on applications written without bind variables. In particular the response time of applications that frequently repeat SQL statements only differing in literals can have a dramatic increase. In addition to the performance improvements shown, using this parameter also decreases memory usage as more SQL statement sharing takes place. However, as soft parsing still takes place, applications without bind variables will still see scalability problems albeit at a higher level of throughput.

Additionally, the tests show that setting the **session_cached_cursors** parameter to an appropriate value can have some scalability improvements on certain types of applications, in particular those that do use bind variables, but do it with a cursor open/close or a (soft) parse for each SQL statement. Appropriate values for this parameter is typically between the number of frequently executed SQL statements per application program and the value of the **open_cursors** startup parameter.

For well designed applications (that do multiple executes of a parsed SQL statement), setting **cursor_space_for_time** can make some further improvement to scalability at the expense of some more memory overhead.

The Oracle startup parameters **cursor_sharing** and **session_cached_cursors** allow application programmers some relaxation on the guidelines for proper application design. However, to fully get the scalability and performance of the Oracle database, applications should be designed according to the guidelines.

The following general recommendations should be followed for application programming:

- SQL statements, that are repeatedly executed within the same module of the application should be associated with specific cursors, which are opened once, have the SQL statement with bind variables parsed once, and are repeatedly executed. This provides the best possible response time and scalability. This can be achieved by using explicit cursors in Oracle Precompiler programs and by properly using statement handles in Oracle Call Interface programs. Oracle Precompiler programs using implicit cursors, should run with HOLD_CURSOR=YES, RELEASE_CURSOR=NO for specific SQL statements or generally with MAXOPENCURSORS set to a number sufficiently high to keep all frequently executed cursors open; the default value of 10 is typically too low. PL/SQL code has this behavior when it uses cursor for loops or declare cursors with static SQL statements.
- SQL statements that are repeatedly executed from anywhere within the application, and for which the above method cannot be used, should be executed using bind variables. If this is not possible, or if an already written application need to use the benefits of bind variable, the **cursor_sharing** parameter can be set to force or similar. See however the comment below on decision support type applications.
- Frequent soft parsing of SQL statements or open/close of cursors should generally be avoided. For applications that nevertheless do this, setting the **session_cached_cursors** parameter can bring some benefit to scalability, but the parameter should not be used as a substitute for proper application programming.
- Applications should not stop using bind variables altogether and completely rely on using the **cursor_sharing** parameter. The major benefit of this parameter is to improve performance of existing applications that were not using bind variables by avoiding the expensive hard parse. Note that scalability problems with SQL processing also comes from the soft parse, and the soft parse is not avoided when using this parameter. Also note, that applications that do not actually have any repeated execution of SQL statements only differing in literals, or applications that do not have actual problems with parsing do not benefit from using this parameter.
- As an exception, SQL statements, in particular complex SQL statements that are rarely executed more than once, should be parsed using literals and not using bind variables, to allow the optimizer to get full benefit of the available statistics. Please refer to the special considerations section below for details.

Note, that in addition to the performance benefit of using the **cursor_sharing** parameter, the actual sharing of SQL statements, that would otherwise be different due to differences in literals, implies a memory saving as well.

Also note, that some applications or some parts of an application using literals may not want to run with this

parameter set to a value different from the default. This is in particular true, if your application uses SQL statements with literals that actually do not change from one invocation to the next. An example of such a SQL statement may be one with an IN-list such as:

```
select * from elements
where status in (1,4)
```

where each invocation of the statement would actually use the same values for the literals, 1 and 4.

SPECIAL CONSIDERATIONS

The use of bind variables in SQL statements, either directly coded in the application or indirectly via the **cursor_sharing** parameter, does generally improve performance. However, there are cases where using it either does not actually have any impact at all on performance or memory consumption, or where its use would actually have a risk of decreasing performance. This section discusses these issues and discusses how they are solved in Oracle9i.

CONSIDERATIONS FOR DECISION SUPPORT SYSTEMS

Decision support systems, such as data warehouse applications, are characterized by small numbers of users executing complex SQL statements with little or no repetition, in which case there is no actual sharing in the library cache possible. For such complex SQL statements, it is also important that the optimizer has all necessary information available to choose the right access plan, and in Oracle8i this is only possible if bind variables are not being used. In Oracle8i, using a bind variable hides the actual value to the optimizer, and it will therefore use heuristics to generate the best access plan, in stead of potentially using information about data distribution. For this type of application, the major amount of resources spent is not in the parse or optimization step, and hence, reducing these steps have little impact on the overall performance. Rather, it is important that the optimization step is done with the best possible data available, in order for the subsequent execute step, which can very well run for minutes or hours, to be as efficient as possible. This is in particular true, if you have collected column statistics that make detailed information about the data4 available to the optimizer.

Therefore, for this type of applications running with Oracle8i, bind variables should not be used and the **cursor_sharing** parameter should be set to exact, to not have the cursor, and therefore the execution plan, shared.

BIND VARIABLE PEEKING IN ORACLE9I

In Oracle9i, the optimizer will peek at actual values of bind variables if these are available. By doing this, the optimizer can use generated table/index- or column-level statistics when bind variables are used. This will benefit both applications that themselves are written to use bind variables, and applications that have had literals replaced by bind variables using the **cursor_sharing** parameter.

The tests that we have conducted show that the bind variable peeking works as expected: the optimizer will make its decision based on actual values of bind variables. However, once an execution plan is generated by the optimizer, no further peeking into bind variables will take place; neither from the same session or from a different session parsing the same SQL statement. In both cases, the same SQL statement and associated execution plan is found in the library cache.

CURSOR_SHARING IN ORACLE9I

In Oracle9i, the cursor_sharing parameter can have the value similar, which will cause the cursor sharing facility to handle decision support queries well. Consider the two SQL statements below, assuming the emp table has a primary key index on the empno column and an ordinary index on the deptno columns:

```
select * from emp where empno=1234
```

and

```
select * from emp where deptno<400
```

In the first case with the primary key lookup, the decisions made by the optimizer would not change if another literal such as 5678 replaced the literal 1234. Hence, cursors can be safely shared without any effect on best optimization, and literals like these are called safe literals. In the second case with the inequality condition, however, the optimizer is likely to generate a different execution plan, if the literal 400 is replaced by the literal 10. Hence, although the two SQL statements parse identical, they should not have a shared execution plan, and such literals are called unsafe literals. In Oracle9i, such distinctions are done by the cursor sharing mechanism, when the value of the **cursor_sharing** parameter is **similar**.

Using the value **similar** for **cursor_sharing** also has the benefit of making cursor sharing behave well, when SQL statements are using literals that do not actually change from one invocation to another as in the example with an IN-list above.

Although it would seem attractive to set **cursor_sharing** to **similar** in Oracle9i for both OLTP and decision support systems, there are cases where this would not be beneficial. When **similar** is used in cases where SQL statements have literals, whose actual value does influence optimization, and whose value does change with most or each statement, the performance will be equivalent to the performance without cursor sharing (i.e. the default value of **cursor_sharing=exact**), which does a hard parse for each statement. Therefore cursor sharing should not be used in such cases. In the case, where the actual value of literals only change among few different ones, **cursor_sharing=similar** can have the effect of increasing scalability and response time. However, this appears to be the rare cases, and using **cursor_sharing=similar** does generally not appear to have any positive effect compared to proper application coding. The results quoted in the appendix clearly show, that **cursor_sharing=similar** actually causes slightly worse performance when the actual value of a literal (influencing optimization) changes between invocation.

CONCLUSION

The Oracle database engine is written to support applications, that need to scale to tens of thousands of users, which can very well be achieved, if certain design recommendations are being followed. Certain Oracle startup parameters can be used to modify the behaviour of the Oracle kernel – both to provide some enhancements for incorrectly written applications, and to make correctly written applications perform even better.

This paper explains

- How applications should be designed to fully explore the possibilities of the Oracle database engine.
- How the database engine can be adjusted to handle applications that have been programmed with more or less focus on scalability.

The three parameters that have been discussed in this paper should be used according to these guidelines:

Cursor_sharing	This parameter will improve scalability and performance of applications written using literals instead of bind variables in SQL statements. Using the value **force** can give good improvements in throughput performance and some improvements in scalability. For decision support and data warehouse applications it should be used with care or not at all in Oracle8i. In Oracle9i the value **similar** is possible for this parameter, and this value allows cursor sharing to be used in many cases of decision support and data warehouse applications. Setting this parameter is typically decided by the Database Administrator, when repeated hard parsing of SQL statements only differing in literals is taking place.
session_cached_cursors	Setting this parameter to a value between the actual number of cursors used and the value of the **open_cursors** parameter, can provide some improvement in scalability and throughput performance. There is some overhead in CPU, effectively making this parameter useless for values higher than 50-100. Also, there is some minor memory overhead, which normally can be ignored. The Database Administrator would typically decide to set this parameter when too much soft parsing is taking place.
Cursor_space_for_time	Well written application can further increase scalability at the expense of using more memory by setting this parameter to true.

ORACLE9I

In Oracle9i, the new features available (the value **similar** for **cursor_sharing** and bind variable peeking), may have some good performance impact on poorly written applications. However, it must be stressed, the best possible performance arises from using proper application design practices, which really have not changed with Oracle9i. These good practices can be put very simple:

- Applications (typically OLTP) frequently repeating the same operation should parse SQL statements only once, use bind variables, and only actually repeat the execute step.
- Applications (typically decision support) doing operations that rarely are repeated should parse SQL statements using literals to give the optimizer as much information as possible.

APPENDIX: TEST RESULTS

This paper discusses the effects of doing SQL processing in different ways, with different programming models and with different Oracle startup parameters. Tests have been designed specifically to see the effects of this, and have been executed on a test system with 24 CPUs. The core of each test is a loop that performs one thousand executions of a SQL statement, and each of the tests have been executed running as a single program or as 3, 6, 12, or 24 concurrently executing copies. In all cases, the average value of the sum of the CPU time used and the time spent waiting for latches has been measured[3]. The absolute value of the results has no significance, but results can be compared, showing the benefit of specific application programming models or Oracle startup parameters.

The first test is using bind variables as advised by Oracle for best performance of OLTP type applications, and it demonstrates the effect of the programming models that do parse and execute for each statement or only do parse once, and the effects of using the **session_cached_cursors** and **cursor_space_for_time startup** parameters.

The second and third tests demonstrate the use of the **cursor_sharing** parameter. The second test is a typical OLTP example with safe literals, where no literal values influence the optimization and **cursor_sharing=force** is used. The third test is a case with an unsafe literal, whose value does influence optimization and **cursor_sharing=similar** is used.

Within each of the first two tests, all values can be directly compared, as the actually executed SQL statement is the same. However, the parts of the third test all use different SQL statements and the results of this can therefore not be compared among each other or to those of the first and second test.

TESTING SESSION_CACHED_CURSORS, CURSOR_SPACE_FOR_TIME AND PROGRAMMING MODEL

Test program using bind variables to do a simple primary key based lookup with different combinations of programming model and startup parameters. In all tests, open/close of the cursor is outside the loop.

	1	3	6	12	24
Program doing repeated parse and execute and not using **session_cached_cursors**	74	78	78	95	146
Program doing repeated parse and execute and using **session_cached_cursors=1**	68	67	66	73	83
Program doing parse only once, repeating execute and not using **cursor_space_for_time**	33	32	37	43	47
Program doing parse only once, repeating execute and using **cursor_space_for_time=true**	28	28	31	35	33

TESTING CURSOR_SHARING FOR STATEMENTS WITH SAFE LITERALS

Test program doing a simple primary key based lookup effectively the same as in the first tests, but using literals in stead of bind variables. All tests do cursor open/close only once outside the loop.

	1	3	6	12	24
Test using safe literal and **cursor_sharing=exact** (i.e. no cursor sharing)	349	407	499	995	2135

[3] The CPU time is retrieved from the Oracle statistics 'CPU used by this session', and the time waiting for latches from the 'latch free' event.

Test using safe literal and **cursor_sharing=force**	85	95	97	102	164

TESTING CURSOR_SHARING FOR STATEMENTS WITH UNSAFE LITERALS

Test program doing a query on an indexed column of a table with skew data distribution and with column statistics gathered. Hence, some queries will use full table scan, and some will use index lookup. The tests show the effect of having unsafe literals that change more or less rapidly between invocations in combination with **cursor_sharing=similar**. All tests were done with cursor open/close outside the loop and repeating parse and execute of SQL statements. Also, all tests actually do use the correct access plan (full table scan or index lookup); differences are therefore not due to incorrect access plans.

Note, that results can only be compared within each subgroup (i.e. among results separated by a dotted line) as the actual test executed is different between the different subgroups having different rates of change of the unsafe literal. Results can also not be compared to those of the previous two tests.

	1	3	6	12	24
Test using rarely changing unsafe literal and no **cursor_sharing**	557	607	670	1058	2355
Test using rarely changing unsafe literal and **cursor_sharing=similar**	318	350	352	396	554
Test with unsafe literal changing every 10 calls and no **cursor_sharing**	321	389	489	938	1993
Test with unsafe literal changing every 10 calls and **cursor_sharing=similar**	120	139	133	158	222
Test with unsafe literal changing every 3 calls and no **cursor_sharing**	330	365	485	900	1949
Test with unsafe literal changing every 3 calll and **cursor_sharing=similar**	172	194	210	264	524
Test with continuously changing unsafe literal and no **cursor_sharing**	316	358	468	846	1838
Test with continuously changing unsafe literal and **cursor_sharing=similar**	369	417	505	824	1724

ACKNOWLEDGMENTS

The results quoted in this paper were measured while Mr. Engsig was an employee of Oracle Corporation. Mr. Engsig would like to thank his collegues at Oracle, in particular Sanjay Kaluskar, Graham Wood, Debu Chatterjee, Lex De Haan and Cecilia Gervasio for assistance in ideas.