Designing applications for performance and scalability

An Oracle White Paper
July 2005

Designing applications for performance and scalability

Designing applications for performance and scalability

OVERVIEW

The Oracle database has been designed to allow developers to design and build applications that will provide very high performance and scale to thousands of users. Much of the work done by the database engine is of repetitive nature: Multiple users continue to perform the same type of transactions, and Oracle can efficiently share information between these, leading to reductions in resource usage on database and application servers. In a similar manner, batching network operations reduces network resource consumption.

This white paper discusses how applications can be written to fully utilize the scalability and performance features of the Oracle Database engine.

This white paper discusses how applications can utilize the Oracle features available to handle SQL statements and PL/SQL blocks providing high performance and scalability and at the same time reduce server and network resource consumption.

INTRODUCTION

Application developers writing applications against relational databases like Oracle have always had rules on how to implement the application, in order to achieve performance and scalability. The database work that needs to be done by the application is expressed using the SQL language, and writing applications correctly to minimize both the actual execution time and the associated overhead, have been key to performant applications. This white paper lays the groundwork for application developers to use in the design phases of application development.

Most online, multi-user applications, if not all, have a need to execute the same SQL statements repeatedly and by multiple users concurrently, and sharing of such SQL statements is imperative for applications to scale to thousands of users. Oracle is able to share identical SQL statements executed by many users concurrently by keeping these ready for subsequent execution in a cache that is part of the database server.[1]

When applications are processing many rows at once, such as inserting a batch of rows or returning multiple rows from a query, these rows need to be transferred between the application and the Oracle server. By default, many interfaces do this one row at a time, which has the implication of using many network roundtrips.

[1] This paper discusses *handling* of SQL statements, and how application programs should deal with SQL statements (and PL/SQL blocks). The topics of tuning of individual SQL statements or of modeling database structures such as tables and indexes are outside the scope of this paper.

Performance can be highly improved by using arrays of rows, which can be transferred in a single network package.

Applications that fail to follow the recommendations in this paper will show lack of scalability, and/or excessive CPU and network usage.

SQL PROCESSING IN ORACLE

Understanding of Oracle SQL processing allows application programmers writing fully scalable applications. In particular, avoiding hard and soft parses will greatly reduce CPU overhead and increase sharing of memory and processing of arrays of rows rather than single rows will reduce the number of network roundtrips.

When a client application[2] sends SQL statements[3] to the Oracle server for processing, there are generally three or four steps involved, depending on the type of SQL statement. Most of these steps are directly available to the application program through an application programming interface (API). A section of this paper will discuss the most frequently used of these APIs.

The four steps are:

1. A parse step, where the Oracle engine parses the SQL statement, verifies its syntactical correctness, and verifies access rights. Subsequently, Oracle does an optimization, where it finds the best access plan for the SQL statement using information about the size of the various objects (tables, indexes, etc.) that will be accessed. In many cases, the SQL statement given to Oracle contains parameters in the form of placeholders, as actual values are not known until runtime. Note, that if the statement is a DDL statement (such as CREATE TABLE), which is outside the scope of this paper, processing is completed during the parse step.

2. A bind step, where actual values from the application are provided to the placeholders of the SQL statement. This can be done by reference (memory address) or by value.

3. An execution step, which uses the actual values of placeholders and the result of the optimization step to perform access to data in the database. For DML[4] statements, multiple rows may be processed during a single execute.

4. For queries[5], the execution is followed by one or more fetch steps, which actually return the row(s) of the query to the application.

A careful understanding of these steps will show that real user data are being processed in the steps 2 through 4; and that the step 1 merely is present for the Oracle engine to deal with the SQL statement. This first step may take considerable time and resources, and as it is overhead seen from the data processing point of view, applications should be written to minimize the amount of

[2] In this white paper, the term "client" refers to application code executing in an application server as well as to traditional clients in a client/server environment.
[3] PL/SQL blocks are processed in the same way. Therefore, whenever the term "SQL statement" is used in the paper, it may as well be a PL/SQL block.
[4] DML is Data Manipulation Language and DML statements are statements that manipulate data in the database. The most frequenly used DML statements are INSERT, UPDATE and DELETE.
[5] Queries are statemens that query the database, i.e. SELECT statements.

time spent during this step. The most efficient way to do this is to avoid the parse/optimization step as much as possible.

Inside the Oracle server engine, all currently executing SQL statements are cached in the library cache, and each application program (or more precisely, each session[6] connected to Oracle) can have any number of SQL statements concurrently in some state of processing[7].

During the parse step, 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. If not, the statement will need to be parsed, checked for syntax errors, checked for correctness of table- and column-names, and optimized to find the best access plan, etc. The former type of parse is called a soft parse and it is considerably faster than the latter, a hard parse.

After the hard parse, all information about how to process the SQL statement is saved together with the SQL statement itself in the library cache.[8]

Applications only indirectly control if the hard parse or the soft parse is being used, and there is only one "parse" function available to the application programmer. The hard parse is performed just once, by the session that happens to be the first to call the parse function, all subsequent parses from any session of the same SQL statement will be a soft parse, that will utilize the information stored in the library cache.[9] The method used by Oracle to verify if a SQL statement is already found in the library cache is relatively complex; it includes a simple string comparison, verification that names refer to the same tables and other database objects, verification that optimizer settings are identical, etc. Hence, only completely identical SQL statements will be considered identical.

Hard parses as well as soft parses include processing in the Oracle server, that applications should be written to avoid. As hard parses involve considerable processing work, applications should in particular avoid hard parses.

[6] A session is loosely speaking a connection to the Oracle server authenticated by a specific username and password. All SQL processing takes place within a session. Using connection pooling, it is possible that multiple sessions use the same connection; this distinction is however not relevant for the discussions in this white paper.
[7] The number of SQL statements concurrently available to each session is limited by the Oracle startup parameter, open_cursors, which typically is configured by the database administrator.
[8] The optimization does in many cases not actually take place until during the first execute step. This is however not relevant for the discussion in this white paper, and the reader can therefore safely assume, that (hard) parsing also includes optimization.
[9] The size of the library cache storing SQL statements is controlled by Oracle startup parameters. In Oracle Database 10g, the parameter SGA_TARGET indirectly controls the size; in previous releases of Oracle, the library cache was part of the shared pool controlled by the SHARED_POOL_SIZE parameter. SQL statements not recently being actually executed may get aged out of the cache to make room for other SQL statements.

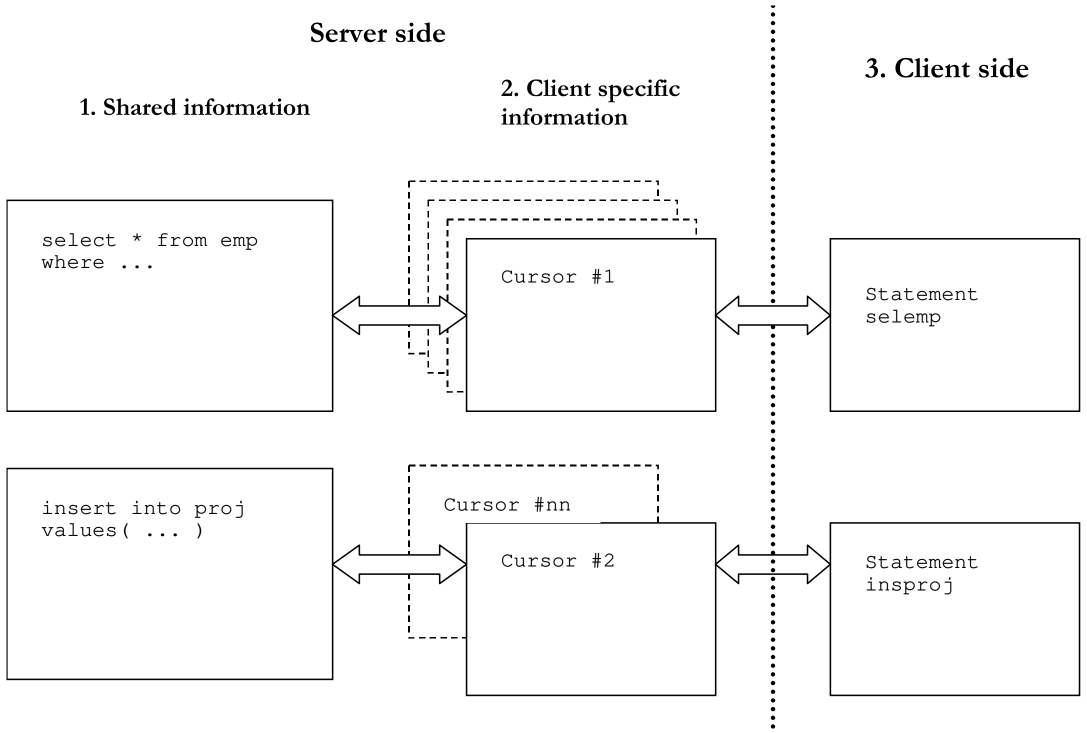
When a session has parsed a certain SQL statement, it will bind actual run-time values for the placeholders in the statement, and finally, the statement is executed, and for queries, the resulting rows are fetched. Once a session has parsed a statement, it can efficiently be executed multiple times by binding new actual values, completely avoiding the parse step.

During the execute of DML or during the fetch of a query, multiple rows may be processed. This processing should be done using an array interface to reduce the number of network roundtrips between the client and the server.

The need for cursors

Cursors – which are references to active SQL statements – are the core mechanism that application programmers use to interface to Oracle.

Typically, applications will need to process many SQL statements concurrently and intermix processing of the different statements. This is precisely what *cursors* are provided for. A cursor can be thought of as a pointer to a SQL statement that is in some state of processing.



Picture 1: Cursor representation on the server and the client side

Picture 1 shows schematically, that there are three different parts to a cursor:

1. On the server side, there is information that can be shared between all concurrent users of a specific SQL statement; this is stored in the library cache as explained earlier.
2. Also on the server side, there is a set of cursors for each individual client. These cursors are simply numbered for each client from 1 and up and are simply assigned in sequence. Hence, different clients (shown with dashed

lines) do in general not have the same number associated with the same SQL statement.

3. Each client application has client side representations of the SQL statements it currently processes, and there will always be a one-to-one correspondence between the client and the server side cursor representation. The actual representation depends on the API used, but it is often referred to as "statement handle" or "cursor".

The processing steps mentioned in the previous section are always directly associated with a specific server side cursor.

As an example, an application may have a loop fetching rows from a query, and for each row fetched, an SQL update statement is performed. This will require two cursors, one associated with the query and one associated with the update. One cursor would be controlling the loop via the fetch step, and another cursor would need to bind values from the row fetched and then execute the update. As this simple example shows, different cursors can (and in many cases will) be at different steps in processing.

Using bind variables

Properly using bind variables is one of the most efficient ways to achieve scalability of application programs.

The concepts of placeholders and of binding actual run-time values to SQL statements have been used loosely in the previous section. We will now take a closer look at this.

As 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

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

```
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. The SQL statements do however, really do the same thing – they only differ in a value. Applications should therefore use placeholders in the SQL statement that are bound to actual run-time values in the application program. Instead of the above two SQL statements, 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.[10] A well-designed application will do a single parse of this statement

[10] Oracle generally uses a colon followed by a number or an identifier for bind variables. You can have as many bind variables as needed, and each should have a different name such as :1, :2, :a, :xyz, :b03, … You can refer to these by position in the SQL statement counting from left or by their names. ODBC and JDBC use '?' for all bind variables, and you refer to them by position.

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 in the library cache. Oracle has always recommended this approach to application programming.

There are two ways of binding, that the different APIs are using; binding *by value* takes an actual value before each execute, binding *by reference* uses the value found at a specific memory address. Some APIs support both methods; others only support one of them.

Bind by value

When binding is done by value, the actual value(s) must explicitly be provided for all placeholders in the SQL statement before each execute. The API will typically have a number of bind calls, which take arguments of different data types such as integer, string, etc. The sample pseudo-code fragments shown in the next section use bind by value.

Bind by reference

When binding is done by reference, the address of variables containing the actual values of placeholders is given, and the variables must therefore contain the actual values before each execute. If the address does not change between executes (which is normally the case), the application only needs to use the bind calls once for each placeholder. Depending on the programming language, APIs may have a single bind call that has one argument specifying the data type and another specifying the variable address, or it may have an individual bind call for each data type that take the variable address as argument.

THREE CATEGORIES OF APPLICATION CODING

The three major steps in execution of SQL statements are the parse separated into a hard parse and a soft parse and the execute. Let us use an example of an application that repeatedly needs to execute the type of SQL statement presented in the previous section – this could be query screen in an application where end users type in an employee number and the application subsequently displays all information about the employee.

Three categories of application coding practices are discussed. Each of these has different behavior with respect to cursor processing, and they allow for increasing sharing of resources and increased scalability and performance.

In the samples shown, there is vague representation of a cursor that is implicitly opened and explicitly closed. Most APIs work similarly.

Category 1 – parsing with literals

The first category of application will repeat all steps for each single input provided by the end user. The application will do this by creating the complete SQL statement at run-time including actual values. The application code will look somewhat similar to:

```
loop
  cursor cur;
  parse(cur, "select * from emp where empno=<some value>");
  execute(cur);
  fetch(cur);
  close(cur);
end loop;
```

Effectively, each time the loop is processed, a new SQL statement is being parsed, which will cause a hard parse to take place in the Oracle server engine. As we have mentioned above, this is a resource intensive process, and it is *not* recommended for any operation that need to be repeated.

Category 2 – continued soft parsing

The second category of application is coded such that the hard parse is replaced by a soft parse. The application will do this by specifying the SQL statement using a bind variable at run-time including the actual value. The application code will now look somewhat similar to:

```
loop
  cursor cur;
  number eno := <some value>;
  parse(cur, "select * from emp where empno=:x");
  bind(cur, ":x", eno);
  execute(cur);
  fetch(cur);
  close(cur);
end loop;
```

Only the first execution of this loop (by the first of potentially many sessions doing so) will cause the parse to be a hard parse; all subsequent executions by any session will perform a soft parse as the SQL statement will be found in the library cache. After parsing, the placeholder is assigned a value by means of the bind call. However, there is still a soft parse, and although less resource intensive than the hard parse, it is still recommended to avoid the parse all together.

Category 3 – repeating execute only

To avoid the parse all together, it needs to be taken out of the loop as shown in this application fragment:

```
cursor cur;
number eno;
parse(cur, "select * from emp where empno=:x");
loop
  eno := <some value>;
  bind(cur, ":x", eno);
  execute(cur);
  fetch(cur);
end loop;
close(cur);
```

By having the parse operation completely outside the loop, only the really necessary steps are being repeated; the bind[11] is necessary to specify the actual value of the placeholder, and the execute and fetch steps are necessary to actually execute the SQL statement and fetch the resulting row.

In this example, coding the application "properly" seems quite simple. In practical application programming, it does however often require considerable application coding to keep track of parsed cursors. Several APIs have a mechanism to automatically do this, by providing an application side cache of SQL statements identified by a key value or by the full SQL text.

Comparison of the categories

Graphs directly comparing performance of the three categories of applications clearly show the benefit of proper application coding.

The parse call – hard or soft – has overhead due to processing requirements, i.e. actual CPU work needed by the Oracle server engine. However, there is additionally considerable scalability overhead. During the hard parse, Oracle needs to lock several internal resources, for example to make sure the structure of the tables involved does not change. Operations on the library cache also require locking of internal resources. These locks (controlled by so-called latches) are taken for very short durations of time, and have little influence on applications supporting single or few users. However, for applications that need to scale many concurrent users, any such lock – as short as it may be – will prevent scalability. Only the execute and fetch steps require no locking and will therefore scale well, the soft and in particular the hard parse require substantial internal locking.

To demonstrate the effect of both the CPU and the locking involved in hard and soft parsing, we have used a sample application that performs a loop very similar to the one used in the examples shown above, and we have made 2000 roundtrips in the loop using the three different application categories. The actual application code has been executed a number of times concurrently ranging from 1 to 24 to inspect scalability. The results show both the effect of using the proper coding category, and the scalability of the different categories.

[11] If the API used does bind by reference, the call to do the binding can also be put outside the loop. However, from a performance point of view, virtually nothing is gained by doing so.

The charts below show processing times executing 1 to 24 concurrently running copies of the three categories of application coding.

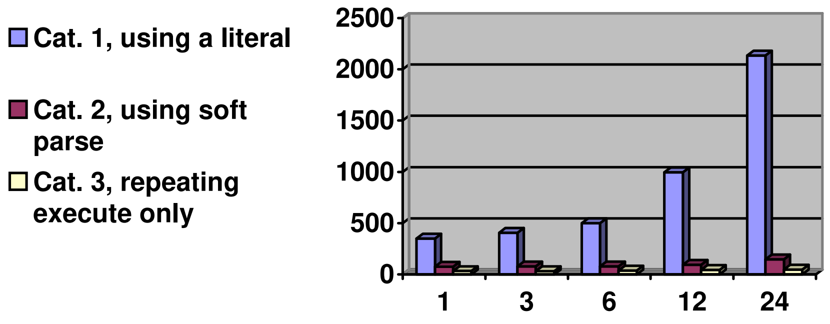
The *horizontal axis* is the number of concurrent executions, and comparing horizontally therefore shows the scalability of the three categories.

The *vertical axis* shows the actual processing time and will therefore show the actual performance of the three categories. The absolute numbers for the time spent has no significance, but the numbers in the two charts are immediately comparable.

Note, that in all charts (pictures 2 through 4), identically colored bars refer to the same results.

Comparing categories 1 and 2

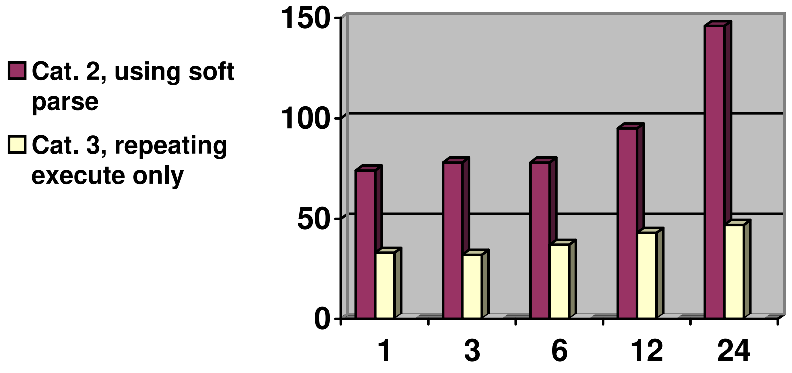
In category 1, an expensive hard parse is being done for each execution. Picture 2 clearly shows that the overhead of doing this is high; the actual performance is five times better using category 2 compared to using category 1. As shown in the chart, scalability of the code is also severely affected.



Picture 2: Category 1 performance

Comparing categories 2 and 3

Picture 3 has the category 1 results omitted and the performance and scalability of the categories 2 and 3 can therefore be seen. The first columns, where a single copy of the test application was executing, show that the overhead of doing a (soft) parse is similar to the actual time spent doing executes. For simple SQL statements like the one being tested, using category 3 can therefore be expected to have roughly twice the performance as category 2. The scalability of the category 2 is clearly not as good as of category 3.



Picture 3: Category 2 and 3 performance

Decision support applications

There are cases, where placeholders should not be used, and where the separation of parse and execute is not necessary.

Decision support applications are in several ways different from transaction processing applications:

- The actual execution of the SQL statement can take considerable time compared to the parsing time
- There is virtually no repetition of SQL statements within a session.
- Concurrent or near concurrent execution of the same SQL statement from multiple sessions is rare.
- The optimizer needs all possible information (including data like column statistics, histograms, etc) to create the best possible execution plan.

Most of this white paper explains how applications should be coded to properly handle repetitive SQL statements, in particular by avoiding repetition of the parse call. However, when SQL statements are *not* repeated, neither within a session nor between sessions, the parse call will need to take place regardless, and the optimization (which is part of parsing), should be as good as possible. This is exactly achieved by using category 1 coding.

When coding the complex, non-repetitive SQL of decision support applications, (including data warehousing, online analytical processing, etc.) the SQL statements including actual values as literals should be parsed before each execute. It is not recommended to use bind variables and not relevant to separate parse and execute.[12]

[12] Oracle will actually peek the value of the bind variable when doing optimization. However, when a SQL statement found in the library cache is being executed, it will not be re-optimized even if the actual value of the bind variable has changed.

Initialization code and other non-repetitive code

Many applications require some code to be executed just once during an initialization or other code that is only called once or not very frequently. Such code has the following characteristics:

- There is little or no repetition of SQL statements within a session
- Multiple sessions *will* execute the same code and are likely to execute it at almost the same time.

As there is no repetition within a single session, there is no need to keep cursors parsed and ready for execution in the application code. However, as multiple sessions do execute the same SQL statements, the library cache in the Oracle kernel can be effectively utilized. This is achieved exactly by using category 2 coding.

Combining placeholders and literals

There are cases, where using literals should be considered. If a column in a table only has few distinct values with uneven distribution, the optimizer is likely to choose different execution plans for where clauses on this column if column statistics have been gathered. As an example, consider the following two queries:

```
select * from … where col=:x and status='YES'
```

and

```
select * from … where col=:x and status='NO'
```

If these two queries had been coded using a placeholder instead of 'YES' or 'NO', they would be identical and the execution plan for both would also be the same. If the number of rows with 'YES and 'NO' values in the status column is very different, a single execution plan is likely to be non-optimal for one of the two queries. Therefore, using two separate cursors for each of the two SQL statements will ensure optimal execution plan for both. Please note, that application program logic is then required to execute the relevant cursor depending on the status value. Also note, that each of these cursors should be parsed only once, and repeatedly executed for different values of the :x placeholder.

Similar considerations should be done in cases where a column contains relatively few (but more than two) different values, and the optimizer is likely to choose different execution plans for where clauses using different values.

Closing unused cursors

Cursors are a resource just as any other resource, and cursors should be closed when no longer in use; otherwise, memory used by the cursors will not be freed. When applications are using category 3, cursors should probably be closed when switching between parts of the application. When applications are using categories 1 or 2, the same cursor can be used repeatedly with different SQL statements and closed when no longer needed. When all cursors from all sessions using the same SQL statement are closed, the server side memory in the library cache will not be freed until it is needed for newer SQL statements. This ensures soft parses (rather than hard parses) are used as long as library cache memory is sufficient, even if there is considerable time between the individual parses.[9]

USING ORACLE PARAMETERS TO IMPROVE CURSOR PROCESSING PERFORMANCE

Several Oracle parameters can be used to influence the behavior of cursor processing in the Oracle server.

Applications are not always coded with sufficient attention to the recommendations in this paper, and Oracle therefore has a set of parameters that will provide reasonable performance for such applications. In fact, there is a parameter that will improve the performance of each of the three application coding categories mentioned.

In [1], the use of these parameters is discussed in detail, and the reader is referred to this white paper for the detailed explanation. We will briefly mention these parameters and additionally provide an updated list of recommendations.

Improving category 1 performance

For applications of category 1, where literals are being used instead of bind variables, leading to unnecessary hard parsing, the parameter **cursor_sharing** can be used. The performance of such applications is primarily influenced by the hard parse necessary. By default, Oracle will behave as explained previously doing a hard parse for each new SQL statement. If, however, these SQL statements could be shared had the literal(s) been replaced by a bind variable, setting the parameter to one of the non-default values will change this behavior. Doing this will bring some of the benefit of coding applications with bind variables in stead of literals.

The **cursor_sharing** parameter can be set to one of three values:

exact	SQL statements are only shared if they are exactly identical as described previously. This is the default
force	If SQL statements only differ in literal values, they will be shared as if the literals had been bind variables. This will be done unconditionally.
similar	Causes cursor sharing to take place when this is known not to have any impact on optimization. The white paper [1] explains this behaviour in detail.

The parameter can be set at the system level using the alter system command or at the session level using the alter session command.

It is highly recommended **not** to write applications dependent on this parameter, and to use the parameter only when needed for existing applications incorrectly using literals.

Improving category 2 performance

Applications of category 2 are identified by repeated (soft) parse of identical SQL statements. On the Oracle server side, this has the implication that the same values repeatedly are assigned to the server side information about cursors. This behavior can be modified by allowing the server to keep information available for frequently parsed SQL statement, at the expense of the need to lookup such SQL statements. The parameter **session_cached_cursors** can be used to do exactly that: If set to an integer value, the Oracle server engine will attempt to keep that many cursors in each session parsed and ready for execution. Appropriate values for the **session_cached_cursors** parameter depend on the Oracle release. In Oracle9*i* Release 2 and later, values as high as several hundreds can be used, in earlier releases, values above 10 to 20 are not likely to be useful as more CPU usage has a tendency to negate the effect of increased scalability using the parameter.

Improving category 3 performance

Associated with category 3 applications, there is a small overhead associated with finishing one execute and starting the next one (of the same cursor). This overhead can be minimized using the parameter **cursor_space_for_time**, which by default is **false**, but can be set to **true** at the system level by the database administrator. Doing this can provide some extra scalability for properly coded applications, however, the extra memory used must be taken into account, and the database administrator should inspect memory usage.

Parameter usage recommendations

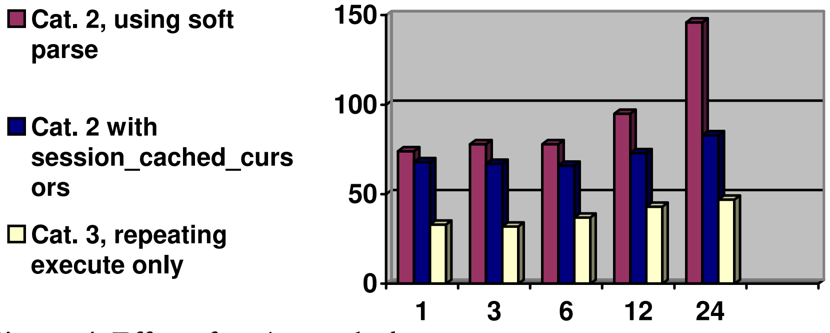
The white paper [1] has a list of recommendations on using these parameters with different types of applications. The following is an updated version of these recommendations with some changes: 1) This paper only distinguishes between three application categories, paper [1] mentions four. The distinction between the categories 2 and 3 in [1] is not really relevant and they are therefore merged. 2) A few recommendations on using the parameters have been corrected. These changes are marked with †

Note that the table below specifically refers to response time as the time needed for individual execution, and to scalability as the ability to execute many application programs concurrently.

	Application category	General behavior	cursor_sharing force or similar	Session_cached_ cursors sufficiently high	cursor_space_fo r_time true
1	Not using bind variables at all.	Generally poor response time and scalability.	Good improvement of response time and scalability; although not quite as good as writing applications using category 2 or 3.	Some extra benefit to scalability†	No effect
2	Parse with bind variables, execute (and fetch) for each SQL statement.	Response time relatively good due to cursor sharing in library cache, limited scalability, due to repeated (soft) parse	No benefit	Improves scalability as the server keeps cursors cached. Only marginally improves response time†.	No effect
3	Parse with bind variables once, only repeating executes (and fetch)	Highest possible response time and scalability	No benefit	No benefit	Additional improvement to scalability

Effect of session_cached_cursors

The chart in picture 4, which can be directly compared to the previously shown



Picture 4: Effect of session_cached_cursors

charts, shows the effect of using the **session_cached_cursors** parameter.

You can see that scalability of applications repeating the soft parse does improve when using the **session_cached_cursors** parameter, but that the improvement to throughput is only marginal. You can also see, that using **session_cached_cursors** is *not* a replacement for properly coding applications to only repeat the execute (and fetch) calls.

In Oracle Database 10g and in 9i releases starting with 9.2.0.5, the parameter **session_cached_cursors** also limits the number of cursors implicitly opened by PL/SQL.

Effect of cursor_sharing

For details of the effect of using the **cursor_sharing** parameter, please see [1]. As explained earlier, using **cursor_sharing** is recommended *only* in cases, where applications incorrectly have been coded using literals instead of bind variables.

Effect of cusor_space_for_time

For details of the effect of using **cursor_space_for_time**, please see [1].

USING THE ARRAY INTERFACE

When batches of rows are being sent between the application and the server, an array interface can be used to optimize network performance.

When more than one row is being sent between the client and the server, performance can be greatly enhanced by batching these rows together in a single network roundtrip rather than having each row sent in individual network roundtrips. This is in particular useful for INSERT and SELECT statements, which frequently process multiple rows and the feature is commonly known as the *array interface*.

To use the array interface, the client application will need to represent data as arrays rather than individual variables containing the data of a single row. For queries, most APIs perform automated array processing such that a client side fetch will return rows from an automatically buffered array fetch. This is known as prefetching. For INSERT statements, most APIs require the application code actually contain the array in the client side and use an array version of the bind and execute calls.

USING STATEMENT CACHING

Several APIs support some form of statement caching, which allows the application programmer to ignore the fact that repeated parse of the same SQL statement has negative performance implications. The general issue that application designers face is that SQL statements are not known until runtime; typically SQL statements are generated based on requests from the user interface. Questions often asked by application designers are: How does the application know, that a SQL statement is likely to be repeatedly executed? And if this information is available, where in the application should the parsed SQL statement be kept ready for execution? In order to keep parse and execute separate, the cursor representation and SQL statement text would often be required to be controlled in a layer outside the layer that actually deals with the data in the SQL statement, which can be hard to code. Statement caching solves these issues by automating the process of identified repeated SQL. The statement cache is set to a certain size, and as long as the size is sufficient, the cache will keep SQL statements parsed and ready for execution.

Statement caching, which is found in several APIs, takes off some of the burden of the application programmer, by providing a automated mechanism to recognize repeated parse of the same SQL statement.

The caching mechanism implements a least recently used mechanism, such that statements that are frequently used by the application code are likely to stay in the cache. However, there is no automated tuning on the size of the statement cache. If statement caching is being considered as an alternative to keep track of frequent execution in the application code, it is preferable for the application programmer to identify statements that do not need caching (e.g. in initialization code, that is executed once only), and treat these as such. Otherwise, there is a risk, that the cache will contain SQL statements that really are not repeated. All APIs that support statement caching have a way to specify, that a certain statement should not be cached.

ANALYZING CURSOR OPERATIONS

Using the SQL_TRACE mechanism, applications can effectively provide data that precisely show the statement processing steps.

During development of a new application or when existing applications are being investigated for scalability concerns, it is very useful to actually understand when cursors are being parsed and executed and whether array interface is being used. Oracle has a tool to do this, SQL_TRACE, which you can enable/disable at the session level using an alter session command. It is strongly recommended that applications always can do this, as it is a very important tool for diagnosing performance and scalability issues.

To use SQL_TRACE a SQL statement like

```
alter session set sql_trace=true
```

needs to be executed as any normal SQL statement. Ideally, applications should be coded, such that the application user is able to switch on SQL_TRACE, e.g. by setting an environment variable. When applications are executed with SQL_TRACE set, a trace file will be generated on the Oracle server, which has details about the processing of the SQL statement. A short example of information from such a trace file is shown in the following extract:

```
PARSING IN CURSOR #3
select * from aa where a>:1
END OF STMT
PARSE #3: … ,mis=1,r=0,
EXEC #3: … r=0,
FETCH #3: … ,r=4,
EXEC #3: … ,mis=0,r=0,
FETCH #3: … ,r=3,
```

The example shows the following:

- The actual SQL statement text (between PARSING IN CURSOR #3 and END OF STMT)
- The cursor number, #3 in this case. The numerical value has little meaning on the client side, but uniquely identifies the SQL statements on the server side. These numbers are the same used in picture 1 identfying the client specific cursor information on the server side.
- The fact, that the statement was parsed (PARSE #3). As explained, the parse can be a hard parse or a soft parse; this is distinguished by the "mis=1" – meaning a hard parse. A soft parse has "mis=0". "r=0" tells the number of rows processed, which is always zero for parse operations.
- The cursor was executed (EXEC #3) and "r=0" shows that no rows were processed. For queries, this is always the case, for DML statements such as UPDATE, the actual number of rows processed will be shown. For

INSERT, the number of rows is larger than one if the array interface has been used.

- Rows were fetched (FETCH #3), which is only of relevance for queries and “r=4” shows that four rows were returned, i.e. that the array interface was being used. Had the array interface not been used, there would have been multiple FETCH lines each with “r=1”.
- The same cursor was executed and fetched once more.

In [2], a complete description of using SQL_TRACE and the associated tool tkprof is found, and [3] provides more technical background about SQL_TRACE and interpretation of the trace file generated.

SUMMARY

Following the five rules shown in the summary is a very good start to scalable and performing applications.

The following is a brief summary of the recommended application coding practices discussed in this paper.

1. Make sure all SQL statements are coded using placeholders and are being parsed only once by each session. This can be done using either appropriate calls of the API in use, or by using the statement cache available in some APIs.[13]
2. As an exception to the first recommendation, complex, long-running statements, such as those seen in decision support type applications, should not use placeholders and should be parsed before each execute to allow the best possible optimization.
3. Always utilize array fetching, as this greatly reduced the number of network roundtrips.
4. If available, use array insert for all insert operations.
5. Make sure the application can be running using SQL_TRACE by means of some external mechanism.

The remainder of this paper lists a number of popular APIs and explains how each of the can be used to properly implement scalable applications.

[13] Queries for one of very few different values of a status code or similar, where the execution plan may depend on the actual value, should be considered as individual SQL statements, and the status value clause should be using a literal.

HANDLING CURSORS WITH SPECIFIC PROGRAMMING INTERFACES

The discussion in this white paper has generally considered the four major operations, parse, bind, execute and fetch. When using various standard application programming interfaces, it is therefore important to understand how these map to the actual calls found. The following frequently used API's will be discussed:

All application-programming interfaces work with cursors. However, different interfaces have different ways of representing a cursor to the application programmer.

- Oracle Call Interface
- Oracle C++ Call Interface
- Oracle Precompilers
- PL/SQL
- JDBC
- ODBC

The term *cursor* is not always being used directly with these APIs; instead, the term *statement handle* or similar is being used as a generic term. However, all these APIs will eventually map to server side cursors, and Oracle does provide methods to show exactly how cursors are being used on the Oracle kernel side.

In the following discussion, we will *not* give a detailed description of all possibilities of the various APIs – only those calls that directly influence the cursor processing and array handling will be discussed.

Several APIs have a concept of statement caching, which will allow application developers some freedom in their coding practices. The next session contains some hints and tips on using these features.

For each API, the following will be discussed:

Cursor representation	Explains how a cursor is being represented in the API.
Parse call	Explains when a parse call is being done at the Oracle kernel side.
Binding	Explains if the API does binding by reference, where you can repeat execute by simply having the application store different values in the bind variables, or by value, where a call is required to set values of placeholders before each execute. In some API's binding is done implicitly and the application developer does not need an explicit bind call.
Execute call	Explains how the execute call can be repeated without repeating the parse.
Array fetch	Explains how array fetching can be used, either by automated buffering or by declaring arrays in the application program

Array insert	Explains how array insert can be implemented. Other DML operations such as update or delete can also utilize the same array interface as insert, although this is in practice not frequently useful.
Best practice	Briefly explains the best practices for coding scalable applications.
Special considerations	Any special considerations for the API. It is strongly recommended to consult the documentation for details.

Oracle Call Interface, OCI

OCI – The Oracle Call Interface for C

Cursor representation	A cursor is associated with a Statement Handle (OCIStmt *).
Parse call	The SQL statement will be parsed during the first call to OCIStmtExecute() after an OCIStmtPrepare() call. The OCIStmtPrepare() call is a local client side call, the does not actually do any server side parsing of the SQL statement.
Binding	OCI does by default bind by reference, and it only needs to be repeated between executes if the address of bind variables change. There are several other ways to do binding, which are fully documented in the Oracle Call Interface manual.
Execute call	A statement is executed using the OCIStmtExecute() call. The first call done after OCIStmtPrepare() will actually also perform the server side parse.
Array fetch	OCI supports automated buffering for array fetching. The size of the buffer can be controlled by number of rows or by memory used.
Array insert	To perform array inserts, the application program must contain arrays of C structs or arrays of individual values that contain the data to be inserted. The OCIBindXXX() calls are used to specify array insert, and the OCIStmtExecute() call take an argument that is the actual size of the array.
Best practice	Statement handles for frequently executed SQL statements should be prepared only once using OCIStmtPrepare() followed by binding of placeholders using any of the OCIBindXXX() calls. Exection should be done using OCIStmtExecute() as often as necessary.

Special considerations	There is a newer call, OCIStmtPrepare2() with an associated OCIStmtRelease() that includes a client side statement cache. Using this, OCI will do a comparison of SQL statements between multiple calls to OCIStmtPrepare2(), such that a subsequent server side parse will be avoided if a SQL statement is already found in the client side statement cache.

Oracle C++ Call Interface, OCCI

OCCI – The Oracle Call Interface for C++

Cursor representation	A cursor is associated with a Statement class. The SQL statement is specified when the Statement class is initialized using the createStatement() method of the Connection class or using the setSQL() method.
Parse call	The SQL statement will be parsed during the first call to any of the ExecuteXXX() methods after a new SQL statement is associated with it.
Binding	Binding is done by value in OCCI and hence, the setXXX() methods must be called before each execute[14].
Execute call	A statement is executed using any of the ExecuteXXX() methods. During the first of these, the statement will also be parsed in the Oracle server.
Array fetch	Array fetching is automatically done.
Array insert	Array insert is implemented using the addIterator() method of the Statement class.
Best practice	Statement classes for frequently executed SQL statements should be associated with SQL statements only once, preferably using the SQL parameter to the createStatement() method of the Connection class. Subsequently, binding of placeholders using any of the setXXX() methods and executes using any ExecuteXXX() method should be done as often as necessary.

[14] Binding by reference is also possible in OCCI if data are available in C format rather than C++ format; please see the documentation for details.

Special considerations	In Oracle Database 10g, a client side statement cache is introduced. Using this, the statement can be searched in the cache when the createStatement() method of the Connection class is being called. Statements being executed only once (or very infrequently) can use the setSQL() method of the Statement class before being executed.

Oracle Precompilers

Oracle Precompilers

Cursor representation	Using the Oracle precompilers, there is a cache of client side cursors, which is used by all implicit SQL statements and by cursors that are declared using an EXEC SQL DECLARE CURSOR operation. The size of the client side cursor cache is controlled by the MAXOPENCURSORS parameter and the precompilers will by default keep that many SQL statements parsed and ready for execution.
Parse call	If the parameter HOLD_CURSOR=YES is specified, the precompilers will save parsed cursors even if the limit of MAXOPENCURSORS is exceeded. If RELEASE_CURSOR=YES the precompilers will not save parsed cursors, even if there are less than MAXOPENCURSORS currently open and parsed. The RELEASE_CURSOR and HOLD_CURSOR parameters can be specified on a statement basis.
Binding	The precompilers perform binding implicitly for static SQL.
Execute call	Executing takes place using the appropriate EXEC SQL.
Array fetch	The precompilers have a PREFETCH option that automates the buffering for array fetching.
Array insert	Array inserts can be achieved by declaring arrays of structures or arrays of individual values in the application program.
Best practice	The HOLD_CURSOR parameter should be set to YES for anything that may be repeated frequently. SQL statements that are done once or very infrequently such as initialization code should specify RELEASE_CURSOR=YES.
Special considerations	Please see the precompiler documentation for more details, including details of processing of dynamic SQL statements.

Static SQL in PL/SQL

Cursor representation

There is no explicit representation of cursors for static SQL directly embedded within PL/SQL.

Cursors associated with static SQL can explicitly be declared using the CURSOR declaration.

Parse call

During the first execute of any static SQL statement within a PL/SQL anonymous block or stored procedure call, the SQL statement will be parsed and the parsed SQL statement will be kept.

During the first OPEN of any declared cursor, the associated SQL statement will be parsed and kept parsed for subsequent OPEN calls.

The anonymous PL/SQL block or the stored procedure call is itself associated with a cursor in the calling environment, which typically is the client application. When this cursor is being closed or reparsed, all cursors within the PL/SQL code that were kept parsed will be closed and therefore need a reparse.

Binding

PL/SQL does binding implicitly when static SQL is being used.

Execute call

Static SQL statements are executed as they are found in the PL/SQL code.

Declared cursors are executed at the OPEN call.

Array processing

Array processing is done in PL/SQL using the bulk mechanism in combination with PL/SQL VARRAY or TABLE OF types. The FORALL operator is used to provide array insert, and the BULK COLLECT operator is used to provide array fetch. As of Oracle Database 10g, PL/SQL provides automatic array fetch for implicit cursors loops. Please see the PL/SQL documentation for further information on using FORALL and BULK COLLECT.

Best practice

By default, processing of all static SQL either embedded within PL/SQL or using declared cursors is done "correctly". As a PL/SQL block may have many cursors open and parsed, it is important that the calling environment correctly keeps the cursor associated with the PL/SQL call parsed. A (re-)parse of the cursor in the calling environment will cause (re-)parsing of all cursors in the PL/SQL block or stored procedure.

Special considerations	In Oracle Database 10g and release 9.2.0.5 and later, the Oracle startup parameter session_cached_cursors limits the number of PL/SQL cursors open. Hence, it is important to set this parameter sufficiently high when PL/SQL execution takes place. PL/SQL packages can be used to store session state – including open cursor – at the package level, and such information is kept independent of the client side cursors as long as the session persists.

PL/SQL native dynamic SQL

Cursor representation	Dynamic SQL statements (i.e. SQL statements contained in a string variable) can be associated with a cursor using OPEN FOR USING.
Parse call	A parse will always take place during the EXECUTE IMMEDIATE or OPEN FOR USING call.
Binding	Binding is done using the USING clause.
Execute call	Execution is done immediately following the parse.
Array processing	Array processing for both insert and fetch is supported using bulk operations.
Best practice	PL/SQL native dynamic SQL should primarily be used for statements that are infrequently executed as the parse and execute cannot be separated. Efficient extract and load processing, which is typically seen in data warehouse type applications, where parsing overhead is neglectible, can be coded using the bulk operations.
Special considerations	Please refer to the PL/SQL documentation for details on binding and IN OUT variables.

PL/SQL, dynamic SQL using the DBMS_SQL package

Cursor representation	The DBMS_SQL package represents a cursor by an integer, which is returned from the OPEN function.
Parse call	A parse is done explicitly using the PARSE procedure.
Binding	Binding is done by value using the BIND_XXX procedures.
Execute call	Execution is done explicitly using the EXECUTE function.

Array processing	Is supported for inserts via the BIND_ARRAY function.
Best practice	For frequently executed dynamic SQL, the cursor should be kept parsed by calling the PARSE procedure only once. For each execution, placeholders should be set using the BIND_XXX procedures and the EXECUTE function should be called to do the actual execute.

JDBC

Oracle JDBC – The Java Database Connectivity interface

Cursor representation	A cursor is associated both with the PreparedStatement and the Statement class, however, only the PreparedStatement class can be used to separate the parse from the execute.
Parse call	During the first call of the execute() or executeQuery() method, the statement (of class PreparedStatement) will be parsed. Subsequent calls to execute() or executeQuery() will not do parsing.
Binding	Placeholders in SQL statements are specified using '?' and binding is done by value. Therefore, appropriate setXXX() methods must be called before each execute.
Execute call	Execution is done with the execute() or executeQuery() method.
Array processing	JDBC automatically buffers rows for fetch operations. Array inserts are not supported.
Best practice	You should open the PreparedStatement object using the SQL statement, and repeat the execute() or executeQuery() method as often as needed. If the close() method is called, it must be re-opened and hence, a parse of the SQL statement will take place. Using statement caching, you can alternatively call the close() method and repeat the open() call with an identical SQL statement text. As long as the statement cache is not exhausted, any open() with a SQL statement found in the cache will actually use the parsed statement in the cache and the subsequent call to the execute() or executeQuery() method will not do a parse.
Special considerations	The Statement class should only be used for SQL statements that are very infrequently or not at all repeated. The parse and execute steps will always be done at the same time when the Statement class is used.

ODBC

Oracle ODBC – The Open Database Connectivity interface for C++

Cursor representation	ODBC has three classes associated with cursors. The simple Statement class, which will always do parse and execute at the same time, and the PreparedStatement and CallableStatement classes, that both allow separation of parse and execute.
Parse call	Parsing is done during the first call to one of the executeXXX() methods of the PreparedStatement or CallableStatement class.
Binding	In ODBC, '?' is used as the placeholder in SQL statements and binding is done by value. Applications therefore need to use the setXXX() methods of the PreparedStatement or CallableStatement class before each execute.
Execute call	The statement is executed during one of the executeXXX() methods.
Array fetch	Array fetching is implemented by using the setFetchSize() method of the ResultSet class.
Array insert	Array inserting cannot be done.
Best practice	Applications should use the PreparedStatement or CallableStatement class, and should use the prepareStatement() or prepareCall() method of the Connection class once, subsequently repeating one of the executeXXX() methods. As the simple Statement class does not separate parse and execute, and it furthermore does not support bind variables, it is not recommended to use, except for the complex, long-running SQL of decision support applications.
Special considerations	Different ODBC drivers are available, and some have a statement cache mechanism. Please see your ODBC documentation for details.

CONCLUSION

Oracle supports applications that will scale to thousands of users. This paper provides best practices for application developers, and explains how the often-used APIs can be used to ensure scalability and performance.

In modern application development, predictions on sizes of database applications and number of application users are rarely possible. Therefore, designers and developers of applications are required to use all possible methods to plan for growth, and a good understanding of how system resources, such as the database, are working is imperative. The Oracle database has been designed to fully support these needs by allowing thousands of concurrent users. However, certain application design rules needs to be adhered to. This paper explains:

- Generally how applications can be designed to minimize overhead.
- How scalability of repeated execution of the same SQL statements can be achieved.
- How data can be batched to minimize network utilization.
- A number of the often-used application programming interfaces, and how these expose the performance and scalability features.

REFERENCES

[1] Bjørn Engsig et al: *Efficient Use of bind variables, cursor_sharing and related cursor parameters*. August 2001. Available from the Oracle technology network at http://www.oracle.com/technology/deploy/performance

[2] Oracle Corporation: *Oracle Database Performance Tuning Guide, 10g,* chapter 20.

[3] Oracle Support notes 21154.1 and 39817.1. Available for registered Oracle customers from the Oracle support site http://metalink.oracle.com.

REVISIONS

This white paper was first issued in June 2005. This second issue includes the following changes:

- Describing use of SQL statements combining literals and placeholders.
- Array interface usage in PL/SQL
- Minor typing errors and reorder of some sections

Designing applications for performance and scalability
2nd issue
July 2005
Author: Bjørn Engsig
Contributing Authors: Lex de Haan, Graham Wood, John Rees

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