

Freedom, Order, and PL/SQL Optimization

18th December 2003

*Charles Wetherell, Consulting Member of Technical Staff
Oracle Corporation*

*Posted for download on
otn.oracle.com/tech/pl_sql/htdocs/New_In_10gR1.htm*

Please check the site for updated versions

Freedom, Order, and PL/SQL Optimization

Oracle Database 10g introduces a new optimizing PL/SQL compiler. This paper, intended for PL/SQL programmers, explores the use and behavior of the new compiler.

**Charles Wetherell, Consulting Member of
Technical Staff, PL/SQL Development**

Oracle Database 10g introduces a new optimizing PL/SQL compiler. Computationally intensive PL/SQL programs compiled under Oracle Database 10g run, on average, twice as fast as they did under Oracle9i Database and three times as fast as they did under Oracle8 Database. To achieve these speedups, the Oracle Database 10g PL/SQL compiler reorganizes the original source code somewhat (as all optimizing compilers do). The optimizations apply both to interpreted and to natively compiled PL/SQL programs. These rearrangements may cause “inessential” changes in the detailed behavior of a PL/SQL program.

This paper describes exactly what changes may be made and why; it is a careful explanation of parts of the PL/SQL definition which have been only lightly described previously. Those who design or program PL/SQL applications and who generally are familiar with the PL/SQL reference materials will benefit from reading this paper as will those who are responsible for porting existing applications from earlier Oracle Database releases. The material is technical, but every effort is made to keep the presentation clear enough so that all those interested in PL/SQL may learn from it.

Introduction

PL/SQL, like most programming languages, intentionally does not specify execution order for some constructs. There are several reasons for this “imprecision”:

- the underlying mathematics from which a construct was derived does not specify any particular order; for example, in mathematics “ $a+b$ ” can evaluate either a or b first before doing the addition.

- programmers should not be burdened with worrying about ordering in constructs for which no natural order exists.
- the programming language designer thought that specifying an ordering would lead programmers into shaky programming practices and would encourage coding styles which would be buggy and hard to maintain.
- finally, the programming language designer thought that leaving some freedom would allow language compilers an opportunity to optimize program performance.

For example, PL/SQL operators of equal precedence may be executed in any order. If **a**, **b**, and **c** are **PLS_INTEGER** variables, the expression

a + b + c **(Code 1)**

can be evaluated as

(a + b) + c **(Code 2)**

or as

a + (b + c) **(Code 3)**

How can these different orders make any difference in a PL/SQL program? After all, addition is associative — strings of additions can be reparenthesized in any pattern.

There would be no difference if PL/SQL operated on mathematical numbers. But it doesn't: it operates on numbers limited to maximum sizes so that addition can overflow. Assume that variables **a**, **b**, and **c** have the same very large absolute value (say 2,000,000,000), that **a** is negative, and that **b** and **c** are positive. Then if **Code 2** is used, the addition of **a** and **b** will cancel to zero and the result will be the very big positive value from **c**. But if **Code 3** is used, the addition of **b** and **c** will overflow and the result will be an exception.

In the past, the freedom to reorder evaluations has not been much used in PL/SQL. By and large, execution proceeded from top to bottom and from left to right through the source code except where the programmer explicitly specified another order through the use of constructs like **IF-THEN-ELSE** or parenthesization of arithmetic expressions. Oracle Database 10g, however, includes a new optimizing PL/SQL compiler. Optimized computationally intensive PL/SQL code runs about three times faster than the same code did under Oracle8 Database and about twice as fast as it did under Oracle9i Database. To achieve these performance gains, the PL/SQL compiler must reorder the evaluation of the PL/SQL program by taking advantage of the freedoms which are part of the language definition.

Clearly, freedom to reorder can lead to different results. The optimizing PL/SQL compiler must constrain itself to program changes which will not bring surprising effects in their train. The remainder of this paper explores PL/SQL freedoms, the uses the compiler makes of those freedoms, and the limits on those uses.

Informal Description Of PL/SQL Freedoms

Informally, there are several places where PL/SQL has some freedom.

1. In an expression with several operators of equal precedence, the operators may be evaluated in any order.
2. For most operators, operands may be evaluated in any order. There are some operators (**OR**, **AND**, **IN**, **CASE**, and so on) which enforce some order of evaluation on their operands.
3. The actual arguments of a call may be evaluated in any order. The same is true of bind values for SQL statements.
4. When several operands or arguments may be evaluated in any one of several orders, the evaluations may be intermixed. That is, one value may be partially evaluated, a second (or a third or fourth or ...) started, the first continued, the second finished, and so on. One operand evaluation need not be completed before starting another nor must the operand evaluations be completed in the order in which they were started.
5. When an item from a package is used and requires the package to be initialized, the package will be initialized sometime (possibly by an earlier item from the same package) before the item is used. Exactly which items in a package trigger initialization and exactly how long before a use the initialization occurs are not precisely specified.
6. When an operation may raise any one of several exceptions, the choice of the exception to raise is unspecified so long as at least one is raised.
7. The SQL standard offers a freedom which is finding use in the Oracle Database 10g PL/SQL compiler. If there is a way to evaluate an operation and to provide a “correct” result even though some other method of evaluation might raise an exception, then the exception need not be raised.

The point of freedom 7 is that the goal of a program is to compute results, not to nitpick about errors. As an example, consider the (somewhat contrived) expression

`A*null` (Code 4)

Regardless of the value of **A**, the result of this expression is **null**. Under freedom 7, the entire expression could be replaced with the value **null** and there is no need to look at the variable **A** at all. Thus, if there were any exceptions which might be raised while attempting to access or evaluate **A**, under freedom 7, these can be ignored. One might object that this freedom is not very useful because no one would ever write an expression like **Code 4**. However (1) there is no limit to the ingenuity of programmers, (2) quite a lot of PL/SQL code is written by other programs which are not nearly so perceptive as humans, and (3) there is a reason described below why the PL/SQL compiler might want to take advantage of freedom 7. Certainly this freedom is important enough to be part of the SQL standard.

One might also ask if **A** were a function, could the call to the function **A** be eliminated along with any of the call’s side effects? An answer is provided in section “Freedom And Side Effects” on page 10.

Why Does Freedom Exist?

These freedoms seem only to introduce ambiguity and disorder in the execution of PL/SQL programs. Why, then, do they exist?

1. Assembly languages require the specification of every detail of the program; for example, the exact storage location of each value must be specified. Mountains of detail bury programmers and the cost in programming and debugging is enormous. There is almost no freedom in assembly languages and the programmer carries the burden of specifying everything.
2. The flip side is that many higher level languages explicitly obscure the details so that programmers can concentrate on the important structure of problems. SQL, for example, provides almost no control over details of storage layout, management, or memory access. PL/SQL provides almost no access to the bits which compose data. Each of these languages reserves many freedoms to itself so that the programmer cannot and therefore need not fiddle with the details.
3. Many programming languages allow operations to have side effects. It has long been recognized that relying on the exact order of side effects in contexts like expressions and actual argument lists is dangerous. So PL/SQL and its cousins don't ban side effects, but by allowing some freedom during operation evaluation, they discourage programmers from using inherently dangerous mechanisms.
4. If a language has some freedom built in, the translators and execution machines for that language may be able to run programs much more efficiently. In the **PLS_INTEGER** fragment

```
c := a + b + e;  
...  
d := f + a + b;                                     (Code 5)
```

assume that there are no changes to **a** or **b** between the two assignment statements. Then because PL/SQL allows additions to be reassociated, this fragment could be computed more cheaply (one addition will be saved) as

```
T := a + b;  
c := T + e;  
...  
d := f + T;                                         (Code 6)
```

where **T** is a new variable introduced by the PL/SQL compiler precisely for the purpose of saving the subexpression computation. This example also takes advantage of the freedom to choose which exceptions might be raised (a freedom inherent in the freedom to choose which addition to do first).

It is precisely the freedom to reorder computations which makes program optimization possible. If, as in assembly language, every possible detail is set by the programmer, there is little or no scope for the optimization. Assemblers are simple beasts because they have little useful work to do besides translating the characters of source into the bits of machine language. SQL statements, on the other hand, describe only the data set or effect desired and leave it almost entirely up to the SQL engine how to achieve the effect. The SQL engine may even do “intelligent” statistical analysis of data and previous operations to guide its decisions, analysis which the most sophisticated user might find not just daunting, but impossible to do.

PL/SQL, as most imperative languages do, falls somewhere between these two poles. It allows programmers a fair amount of control, but wants to reserve some freedom for itself, either to save the programmer trouble or to make the translation process more effective and to make compiled programs run more efficiently.

A Definition Of Freedom

This discussion is not merely philosophical. Languages without tight descriptions are inelegant. PL/SQL needs a tight definition so that programmers know what to expect and so that new features and capabilities can be added smoothly and with assurance that no old feature is broken. Most important, Oracle Database 10g provides a new optimizing PL/SQL compiler which speeds up computationally intensive programs ported from older versions by two to three times on average.

To achieve these improvements, the compiler must be free to reorder and to eliminate computations so long as the computational intention of the original program is preserved. Thus, PL/SQL must specify precisely what freedoms exist so that the compiler will not unwittingly overstep the bounds of correctness and produce a program which differs from the intention of the original source. Also, PL/SQL programmers can usually force behavior which is required, but PL/SQL (like most programming languages) provides few ways to say “Don’t care” when details of behavior are inessential to a program’s purpose. Precise specification helps programmers know when to be strict and when to relax.

1. Unless otherwise specified, the operands of an expression operator may be evaluated in any order. The evaluations may be intermixed as well. An operator may have its order specified; for example, the short circuit **p OR q** operator of PL/SQL follows the rule that if **p** is **TRUE**, **q** must not be evaluated. However, if **p** itself is an expression, then the operands of **p** (and **q**, if it ends up being evaluated) may be evaluated in any order by this rule, assuming that they, in turn, do not have a required order of evaluation.
2. The actual arguments of a call or a SQL statement may be evaluated in any order (including default actual arguments). The evaluation of the actual arguments may be intermixed. Similarly, the components of an explicitly listed data structure such as a record initializer may be evaluated in any order.
3. In an expression with several operators of equal precedence, the operators may be reassociated in any order assuming that the operators are associative and that the reassociation corresponds to normal mathematical usage.
4. The operands of a commutative operator may be commuted.
5. Computations necessary to establish addresses (indexing and field selection) are treated as operators. This implies that complicated data references may be evaluated as several pieces (look ahead to **Code 18**).
6. The evaluation of the left hand side and the right hand side of an assignment may be done in any order and the evaluations may be intermixed. The left hand side of an assignment often includes embedded expressions such as those used for indexing.
7. Created values may be reused when it is safe to do so (see **Code 5**).

8. Operators may be replaced by simpler ones or eliminated when it is safe to do so (see **Code 4**). Notice that if an operation is eliminated or simplified, the need to evaluate some or all of its operands is usually eliminated as well. However, some operands may resist removal for their own reasons which are explained in section “Freedom And Side Effects” on page 10.
9. Operations may be reordered (which might cause long movements — look ahead to **Code 14**) when it is safe to do so.
10. A revised operation will not raise an exception which the original operation would not have raised — in other words, new exceptions are not created.
11. A revised operation need not raise an exception which the original operation would have raised.
12. A revised operation may modify the order of package initializations.
13. A possible exception may not be moved from one exception handler to another.

Everything the PL/SQL optimizing compiler does to improve PL/SQL execution exploits these freedoms. There are no other liberties it takes with the definition of PL/SQL.

Side Effects Of All Sorts

In programming language parlance, a *side effect* is any change to the state of a program (including communications with the outside world) which is not part of the direct construction of a value. The addition

RANDOM + 1 (Code 7)

(forgetting about possible exceptions) constructs a new value which is one more than whatever the value of **RANDOM** is. This is the direct effect of the expression. If **RANDOM** actually is a function call to a random number generator and if that call changes a seed value buried inside the generator, then the change to the seed value is a side effect. If the example had been

RANDOM + 1 / RAND_SEED (Code 8)

then the value of the expression depends on the order in which the operands of the addition are evaluated. The value of **RAND_SEED** will change when **RANDOM** is called; the side effect of the call to **RANDOM** affects the evaluation.

The definition of side effect depends on the level at which the program is observed. A report generator, viewed from outside, has the direct effect of producing the desired report; if, in the process, it also changes some database tables, those changes are a side effect. From this global perspective, what happens inside the report generator is not visible and is neither a side effect nor a direct effect.

A compiler, however, takes a worm’s eye view of a program. Every distinct action required by the original source is considered. To ensure that changes it wishes to make are safe, the compiler must track every value used or created by the program. Thus, it

regards even an assignment statement as a side effect. For example, consider the sequence

```
... + a(j) + ...  
a(i) := ...  
...  
x := ... + a(j) + ...
```

(Code 9)

Perhaps in the second assignment, the computation of the subexpression involving `a(j)` could be eliminated because it had been saved from a previous computation as in **Code 6**. But the compiler can only do the elimination if it can prove that the value of `i` is not the same as the value of `j`; otherwise, the first assignment to `a(i)` will “kill” the value of the saved subexpression and `a(j)` must be computed again. In other words, the first assignment statement had a possible effect on the second; the programmer may know that there is no problem, but the compiler must treat all such cases as suspicious. Thus, compilers treat all actions which are not direct computations as side effects.

The distinction between the purpose of a program and its effect must be kept clear. The purpose is a human intention, the effect is what the program actually does. For example, the purpose of the sum

```
EXPENSE + PROFITS
```

(Code 10)

might be to present data useful for accounting purposes which may serve, in turn, an even grander purpose of driving some business decision. But the effect is simply to add two numbers together; the program has no sense of the use to which the result will be put. In fact, the sum may overflow and that, presumably, will not serve the purpose at all even though the exception was always one of the possible effects of the sum.

PL/SQL and other procedural imperative languages live on side effects. A program usually consists of a sequence of intermixed assignments and calls (and, in PL/SQL, SQL statements of course) managed by control flow statements; each changes the program’s state and that is exactly the programmer’s intention. The rules about freedom have taken little account of side effects. Rather, they concentrate on evaluation and on the production of values. For the purpose of analyzing program freedom, side effects must be considered and they fall into four categories.

1. Direct value manipulation side effects: These are explicit changes to data caused by explicit actions. The point of most PL/SQL statements is precisely to cause a side effect; it is explicitly the purpose of an assignment statement and is almost always the purpose of a call or a SQL statement. PL/SQL is designed so that most work is done through such side effects.
2. Intra-statement side effects. Some side effects occur within the execution of a single statement. The programming language community has long felt that relying on the precise ordering of side effects inside single statements — and particularly inside expressions — is risky and prone to error. Most languages discourage such reliance. The example in **Code 8** shows how — in a contrived way — different orderings could lead to different values.
3. Exceptions: Exceptions break the flow of control and may interrupt computations partway to completion. As a general rule, values are in a chaotic state after an exception occurs. Unless handlers are placed around every subcomponent of a

computation, a PL/SQL program cannot diagnose exactly which computations have completed and which have not. Programs usually accommodate this indeterminism either by giving up on a computation entirely or by careful design of safe restart points.

4. Package initialization: Reference to a package element may cause package initialization. Package initialization may execute arbitrary PL/SQL code and may trigger, in turn, initialization of other packages. The cascading effects can change the entire global state of a PL/SQL program. Whether a particular reference to a package element actually triggers initialization depends on whether the package has already been initialized and on the PL/SQL compiler's assessment of whether reference to the package element actually requires initialization for a correct reference.

Direct value manipulation side effects are designed by the programmer and are predictable. Side effects in the other three categories are often not designed by the programmer, are somewhat indeterminate, and in the case of exceptions, commonly signal a failure which is not reparable by the program itself. Even if the original programmer was able to organize all side effects in a coherent way, there is no guarantee that later maintenance or feature changes will maintain the designed organization. It is common enough to encounter bugs in large programs when somebody forgets that calling procedure **DO_IT** requires that setup procedure **START_IT** be called first. But consider a program which depends on the expression

`pkg1.A + pkg2.B` **(Code 11)**

and which assumes that the expression causes package **pkg1** to be initialized before package **pkg2**. Any change in the program, including a distant one in some far off unit, may innocently reference some item in package **pkg2** before the expression is calculated. The order of package initialization is changed and presumably a bug has been introduced, one which experience shows will be difficult to detect and, depending on the need for the distant reference, difficult to rectify.

In client-server applications, many clients access common server packages in an indeterminate order. The combined effect of all the accesses must be correct even though no one accessor can be sure of the initialization status of packages which are accessed. Designers of such applications are aware of the indeterminacy and generally do not rely on specific instantiation order. This is same observation as **Code 11** makes on a larger canvas.

It is for these reasons that programming language designers discourage dependence on the order of intra-statement side effects.

Exceptions And Initializations

Good programs generally minimize intra-statement side effects and avoid implicit dependence on the order of side effects. Of course, PL/SQL and other procedural languages rely on the side effects of assignments and calls, but the cost of debugging unintended side effects is truly staggering. Exceptions are one main source of side effects in PL/SQL. Generally, these fall into one of three categories.

- *Deliberate control flow*: Exceptions defined by the programmer and used to signal specific conditions or built in exceptions which are expected as part of the processing technique. The typical example of a deliberate exception is one which reports the end of input so that the next stage of execution can begin. Obviously, exceptions of this sort should not be reordered or eliminated by compilation or execution.
- *Regrettable*: An exception which might occur but is not part of the normal processing paradigm. The typical example is the loss of a table which an application expects to be present. Commonly, programs deal with such errors at a fairly high level and often use the exception as an opportunity to shut down gracefully or to restart entirely.
- *Unexpected*: An exception for which the program makes no real provision. For example, consider an application which fills a new table using a series of unique keys and then retrieves a row using a **SELECT** statement. If there are two or more rows returned — impossible because the table was just constructed so that the keys are unique — the result is a completely unexpected exception. Such exceptions are commonly the result of malfunctions or bugs. They are usually handled, if at all, only to report cause and location as cleanly as possible.

Obviously, the categories shade into one another and one program's unexpected exception may be another's control flow. But as a general rule, both control flow exceptions and regrettable exceptions arise directly from designed and consciously invoked actions: procedure and function calls, system calls, SQL statements, and the like. The points where they may occur are usually well understood and fairly widely spaced. Unexpected exceptions commonly arise from a failure in design, asynchronously from the detailed operation of PL/SQL execution, or from an unanticipated change in the runtime environment.

There are usually several possible unexpected exceptions for every atomic PL/SQL action. For example, an addition may fail because one of its operands is inaccessible or because the addition overflows. An assignment might fail because the target is inaccessible, because the source is inaccessible, or because the source doesn't meet constraints imposed by the target. A collection extension may fail because there simply is not enough memory available. It would be well-nigh impossible to account for all of these and their many cousins; most well-designed programs simply capture such exceptions near the top level and report them.

The side effects which arise from the initialization of packages are similar. Most packages are either independent of all others or are organized in a hierarchy such that the normal initialization process will start all the packages in the proper order and with no need for the programmer to be concerned about cycles among the initializations. Cyclic dependency of package initialization is usually a program design flaw. Indeed, PL/SQL's ancestor Ada says that programs with such cycles have undefined behavior.

In some cases, a specific order of initialization is required. Usually this can be achieved by adding explicit initialization calls at appropriate points in a program. There may be, however, some existing PL/SQL programs which rely on the historical order of package initialization. Such programs could change behavior if initialization side effects are

reordered or modified. PL/SQL does not have an explicit mechanism to force particular package initializations to occur at particular moments during program execution.

Freedom And Side Effects

Direct value manipulation side effects are clearly intentional. Therefore,

1. The PL/SQL compiler will not reorder, add, or remove explicit procedure, function, or method calls which appear in separate statements. Note, though, that the freedom to reorder much of the evaluation of expressions means that function calls which appear within expressions may be reordered as may other instances of function calls appearing within one statement. This also answers the question posed by **Code 4**. If **A** is a function, it will always be called, but the compiler may disregard the value and replace the multiplication with the **null** constant.

There is a **DETERMINISTIC** keyword which is used by SQL on PL/SQL functions to indicate that the function's value depends only on the values of the actual arguments and that the function does not have any side effects which are of consequence. For the moment, the PL/SQL compiler ignores this keyword. It is possible in the future and with explicit permission from the programmer that the compiler will take advantage of the **DETERMINISTIC** keyword to eliminate some function calls. If this change is ever made to the behavior of PL/SQL, there will be plenty of advanced warning to programmers.

Intra-statement and exception side effects infest PL/SQL. Almost every data reference and every operation may raise one or more exceptions or cause one or more package initializations. If all these must be left unmoved or unmodified, then little optimization of source code will be possible. Therefore,

2. The PL/SQL compiler has the freedom not to do an operation whose only effect is to raise an exception or to initialize a package. By **1**, the PL/SQL compiler may not eliminate a call to a user written subprogram under this rule.
3. If a reference to a package element in a PL/SQL subprogram (for this discussion, anonymous blocks also count as subprograms) requires initialization of the package, PL/SQL is free to perform the initialization at any time between the beginning of the call to the subprogram which immediately encloses the package element reference and the moment when the package element is actually used. The PL/SQL compiler constructs implicit procedures for package specifications and package bodies which actually do the package initialization; this rule applies to those implicit procedures as well.
4. If the PL/SQL compiler can determine that a use of a package element does not require the initialization of the package, then PL/SQL is not required to initialize the package when the element is used. For example, a record type defined in a package may include an initial value. If another unit declares a record of that type, the newly declared record is initialized with value defined in the package. Normally, this would require the package to be instantiated so that the value of the record initializer can be known. However, if the record initializer contains only constants, then the compiler simply copies the constant initializer — at compile time — into the unit which is creating the new record and the package is not instantiated.

5. If an operation may raise an exception when executed, but there is an alternate possible method of execution which will not raise that exception, the PL/SQL compiler is free to choose the alternate method. Once again, explicit function or procedure calls may not be eliminated by this freedom. **Code 4** provides an example of an alternate evaluation.
6. The PL/SQL compiler may not introduce a computation as an alternate which may raise an exception which otherwise would not have been raised during the execution of a program.

The effect of these freedoms taken together is to allow the optimizing PL/SQL compiler considerable opportunity to generate more efficient code. For example, assuming that variables **A** and **B** are not changed in the interval, the sequence

```
... A + B ...  
...  
... A + B ...
```

(Code 12)

can be changed to

```
T := A + B;  
... T ...  
...  
... T ...
```

(Code 13)

saving one addition. Notice that in this case, if any exceptions were to be raised, they would be raised in the first evaluation and so the freedom to avoid exceptions is not needed here.

Similarly,

```
for i in 1 .. 10 loop  
  A := B + C;  
  ...  
end loop;
```

(Code 14)

can be changed to

```
A := B + C;  
for i in 1 .. 10 loop  
  ...  
end loop;
```

(Code 15)

if neither variable **A**, variable **B**, nor variable **C** is changed in the loop. Depending on the number of times the loop executes, this transformation can lower the execution time of the program considerably. However, if the addition raises an exception, the exception order of the program has possibly been changed (if, for example, there were computations in the loop bounds which might also raise exceptions). And if any of the variables is a package element, it is possible that the order in which packages are initialized may change. In this example, the PL/SQL compiler does take advantage of its freedom to reorder computations in order to provide faster execution of the loop.

In PL/SQL, the test `x = null` always has the result `null` regardless of the value of `x`. Real programmers probably never write such a test although it could arise through optimizer transformations. It is useful as an example here:

```
if pkg.A = null then null; end if; (Code 16)
```

The entire `if` statement may be eliminated if `A` is not a procedure or method call because the PL/SQL compiler knows that the result of the comparison is always `null`, that there is no “else” branch on the conditional, and that therefore there is no need to execute the comparison at all. Such an analysis will also eliminate the evaluation of the variable `A` which, in turn, will eliminate any exceptions or package initializations which the reference to `A` might otherwise have caused.

Optimization And Safety

An optimizing compiler makes programs run more quickly. Optimizers employ three basic techniques:

1. Replacement of a slow operation with a faster one.
2. Elimination of an unnecessary operation. This might be regarded as a special case of the first technique where the faster operation does nothing at all!
3. Movement of an operation from a place where it will be executed many times to a place where it will be executed less often.

Clearly, the compiler must be free to make the changes. Further, the compiler must not make changes which violate the intended behavior of the program; it does no good to have an “optimized” program which runs faster but which computes the wrong results. Thus, the rules which determine those actions intended by the programmer and those incidental to program execution must be clear to the optimizer and to the programmer. In other words, the limits of freedom must be well defined for the optimizer to do its job correctly.

As an example of technique 1, consider that `A` is declared in a package `P` and used elsewhere.

```
A NUMBER NOT NULL := 1.0;  
...  
... P.A*0 ... (Code 17)
```

Because the compiler can deduce that the value of `A` is not null, the multiplication might be replaced by the constant `0`, clearly simpler and faster than actually doing the multiplication at run time. But if the program were allowed to rely on the multiplication to initialize the package `P`, then the transformation would not be safe.

Examples of technique 2 are rare in code written by humans, but they often occur in the intermediate code transformations done by the compiler. An example of technique 3 is

```
for j in 1 .. 10 loop
  ... A(3)(j) ...
end loop;
```

(Code 18)

This can be rewritten as

```
T := addr(A(3))
for j in 1 .. 10 loop
  ... T(j) ...
end loop;
```

(Code 19)

where **T** is a temporary variable introduced by the compiler and **addr()** is a special function only available to the compiler — not part of normal PL/SQL. The optimization saves some of the time spent addressing the double subscripted collection variable **A** by moving the unchanging part of the address computation outside of the loop. By the way, the compiler does not copy the value of **A(3)** to the temporary; that could be very expensive. Only the address of the value is saved. To be sure that the transformation is safe, the compiler must determine that there is no change to **A(3)** inside the loop. Much of the research in optimization is on algorithms which allow compilers to make such determinations.

It may seem that the examples are either trivial or unlikely to happen in real well-written programs. However, optimizers discover opportunities because

- not all programs are well-written.
- even in well-written programs, there are often common subexpressions, constant computations included for clarity, and the like which can be improved.
- some code is generated by other programs which are notably less careful than humans.
- addressing and other compiler inserted operations (default initializations, default arguments, and so on) provide opportunities which the PL/SQL programmer cannot avoid.
- each optimization applied tends to open opportunities for others not immediately apparent in the source program.

As an example of the fourth point, **Code 18** shows that addressing might be improved even though a programmer may not be able to make the improvement directly. Here is another example:

```
B := B + 1;
```

(Code 20)

Increments like this are very common in real programs. The PL/SQL virtual machine has a special instruction which does the increment without requiring a reference to a constant **1**; the special instruction is considerably faster than an ordinary addition. There is another version of the instruction for loop variables with a test and a branch back to the beginning of the loop when that is appropriate. The compiler works hard to find places in PL/SQL programs where these instructions may be used.

To summarize, for an optimizer to be useful

- The compiler must know what freedom it has to modify the order of evaluation specified by the program source code and which side effects may be removed or reordered,
- The programmer must know which evaluations, side effects, and exceptions might be modified by the compiler (which is the point of this paper), and
- The two together must agree that the rules of PL/SQL bind them both in a contract.

The Oracle Database 10g PL/SQL compiler makes such agreements with the PL/SQL programmer. The payoff is that PL/SQL programs run two to three times faster than they did in Oracle9i Database or Oracle8 Database, respectively. Further, the generated code and the execution space taken by each function call are both typically 20% to 30% smaller than in earlier releases. The smaller sizes mean that less strain is placed on the underlying execution environment by PL/SQL programs.

One final point. Any source program offers myriad opportunities for optimization and the exact order in which the optimizations are applied might change the behavior of the program when it is executed. However, the Oracle Database 10g PL/SQL compiler is careful to avoid gratuitous transformations and random choices. If the same program is compiled again in the same environment, the execution behavior will be the same for each of the compilations when using the Oracle Database 10g PL/SQL compiler. (This was also true of the PL/SQL compilers in previous Oracle releases simply because they did not attempt to improve performance through optimization.) Thus programmers need not fear that recompilation will change the behavior of a program.

The History Of Freedom

In the past, PL/SQL attempted only minor optimizations. Still, it did take advantage of some of the freedoms described here. This list applies to the Oracle9i Database compiler and earlier; it does not necessarily apply to the Oracle Database 10g PL/SQL compiler.

- Operands of operators were generally evaluated left to right, but there was no particular care taken to ensure that this was always so.
- When several packages needed to be initialized because of the operation of a single construct, this was generally done in left to right order, but no particular care was taken to ensure that this was always so.
- In particular, the order of package initializations may have changed in some cases between Oracle8 Database and Oracle9i Database simply as a consequence of an operational change in the PL/SQL interpreter. If there were any changes, they were never analyzed or documented.
- In arithmetic expressions involving any use of **PLS_INTEGER** or **BINARY_INTEGER** values (especially in combination), the PL/SQL compiler sometimes chose **NUMBER** arithmetic operations and sometimes chose the much faster **PLS_INTEGER** operations. The effects may differ because the two operator classes raise different exceptions and produce different ranges of values. Although the choice was deterministic, it was never documented publicly.

- Similarly, the PL/SQL compiler sometimes represented small integer constants as **NUMBER** and sometimes as **PLS_INTEGER**. The effects are the same as the previous point.
- Although generally a reference to a package element requires initialization of the package, the PL/SQL compiler sometimes determined that the initialization is not necessary. This happened when the referenced element was a type name. More surprisingly, it also happened when the type named was a record type with an initializer when all the elements of the initializing value were literal constants.
- The actual arguments of a call were evaluated in order of the formal parameters and not in the order of the actual argument list.

Controlling PL/SQL Optimization

The Oracle Database 10g optimizing PL/SQL compiler has three optimization levels.

- *Level 2*: apply a wide range of modern optimization techniques. This is the default optimization level. Each individual subprogram is analyzed as a whole; optimizations take account of the entire structure of each subprogram. Level 2 also squeezes the space used during execution so that programs require fewer resources and are more scalable.
- *Level 1*: apply only local optimizations. Under this level, analysis is done only from one branch to the next; changes do not propagate beyond branches (although unnecessary branches may be removed). This level should only be used when compilation time is a very serious consideration; experience suggests that only very large, mechanically generated source programs should use level 1.
- *Level 0*: retain the evaluation order choices made by the Oracle9i Database PL/SQL compiler and otherwise apply level 1 transformations. This level should be used very sparingly as it forfeits considerable execution performance. It is intended only to help debug evaluation order problems when porting PL/SQL programs from earlier releases to Oracle Database 10g and beyond.

In addition to these levels, the PL/SQL compiler may be requested to produce code compatible with the PL/SQL symbolic debugger. Such code is optimized slightly less than code compiled under Level 1 so that the debugger can accurately stop and report on every statement in the original source; otherwise, optimization might eliminate or conflate statements so that they are no longer accessible to the debugger. The need for a separate debug version of compiled code is regrettable, but it is common to many programming languages because research simply does not provide any good solutions for the general problem of debugging optimized code.

All changes made by all levels are safe under the freedoms discussed in “A Definition Of Freedom” on page 5 and under the limitations discussed in “Freedom And Side Effects” on page 10. Extensive testing suggests that almost all programs will port to Oracle Database 10g with no difficulty whatsoever; the performance improvement will be free and transparent.

However, it is possible that a program may depend on the (implicit) order of initialization of packages and that, because of such a dependency, behavior does change when the program is executed under Oracle Database 10g. It is strongly recommended that the program be revised by addition of explicit calls to special initialization procedures which initialize packages in the desired order; such a change can usually be done with little violence to the rest of the program if the initializations are inserted near the beginning of program execution. Leaving code which depends on the order of implicit package initializations in place only stores trouble for future maintenance.

Future Directions

Oracle Database 10g brings a new optimizing PL/SQL compiler and substantial performance improvements to PL/SQL programs. However, there are many other optimizations which may profitably be applied to PL/SQL and some of these will likely appear in future Oracle Database releases. Three possible optimizations illustrate some changes the future may bring.

- The PL/SQL compiler may “inline” some routines — this means that a call is replaced by a copy of the body of the routine. The elimination of the call will almost always make execution faster. Further, if the body is small (as many wrapper and utility routines are) or if the arguments are appropriate (commonly, if some of them are constants), then many more optimizations may be applied to the copy of the body and the surrounding code and execution will go even faster. It is not uncommon to see appropriate use of inlined routines speed up applications by factors of two to four.

If an inlined routine comes from a package and if the copy of the body of the routine — particularly after it has been optimized — does not reference any other element of the package, then there is no longer any need to initialize the package at the call site where the routine was inlined. Programmer assent to the initialization removal will probably be part of this optimization.

- The PL/SQL language might be modified so that the programmer may specify that particular functions are deterministic in the sense of SQL. Then a sequence like this

```
... F(x, y) ...  
...  
... F(x, y) ...
```

(Code 21)

(assume **x** and **y** do not change between the two calls) might be changed to

```
T := F(x, y);  
... T ...  
...  
... T ...
```

(Code 22)

where **T** is a new variable introduced by the compiler. This change saves the cost of a call to the function **F**. By specifying that **F** has no appreciable side effects, the programmer has also given permission to the PL/SQL compiler to break the rule about functions and to eliminate one of the calls. If this change comes about, it will probably be controlled by richer interpretation of the existing **DETERMINISTIC** keyword by the PL/SQL compiler.

- Consider a loop to sum the squares of the first 100 integers.

```
Sum := 0;
for i in 1 .. 100 loop
    Sum := Sum + i*i;
end loop;
```

(Code 23)

This could be changed into

```
Sum := 0;
Sum := Sum + 1*1;
for i in 2 .. 100 loop
    Sum := Sum + i*i;
end loop;
```

(Code 24)

by “unrolling” the first iteration of the loop. Now other optimizations can transform the loop into

```
Sum := 1;
for i in 2 .. 100 loop
    Sum := Sum + i*i;
end loop;
```

(Code 25)

One cycle through the loop has been saved. The same transformation can be applied again to generate

```
Sum := 1;
Sum := Sum + 2*2;
for i in 3 .. 100 loop
    Sum := Sum + i*i;
end loop;
```

(Code 26)

which turns into

```
Sum := 5;
for i in 3 .. 100 loop
    Sum := Sum + i*i;
end loop;
```

(Code 27)

which saves another loop iteration. An aggressive compiler might actually be able to replace the entire construct with the statement

```
Sum := 5050;
```

(Code 28)

In any case, because loops consume most execution time, any improvement in loop performance is worth gaining. Although this particular loop is unlikely to appear in a real program, loop transforms like this simple unrolling can improve execution speed and can provide many opportunities for other optimizations to take effect.

If and when the PL/SQL compiler adds new optimizations, it will continue to use the freedoms and to honor the limitations described in this paper. If, as in the examples, amendments are needed, PL/SQL programmers will be given the control they need to ensure that existing PL/SQL programs will work now and into the future.

Conclusion

Oracle Database 10g includes a new optimizing PL/SQL compiler. It is clear that the compiler must be allowed to manipulate and rearrange the original source to achieve improved execution performance. The PL/SQL rules for allowed transformations — which had not been previously documented in one place — and the ways in which the compiler takes advantage of these rules are now evident. The optimization level of the compiler may be set to meet programmer's needs. PL/SQL programmers may use the ideas presented here to write faster, clearer, and more maintainable PL/SQL programs.

Acknowledgments

Comments of reviewers — from inside and outside Oracle — have improved this paper dramatically. Criticisms, even down to the exact placement of punctuation, were uniformly thoughtful and helpful. Their hard work and valuable insights are deeply appreciated.



Freedom, Order, and PL/SQL Optimization
December 2003

Author:
Charles Wetherell, Consulting Member of Technical Staff, Oracle Corporation

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

Oracle Corporation provides the software
that powers the internet.

Oracle is a registered trademark of Oracle Corporation. Various
product and service names referenced herein may be trademarks
of Oracle Corporation. All other product and service names
mentioned may be trademarks of their respective owners.

Copyright © 2003 Oracle Corporation
All rights reserved.