PL/SQL Enhancements in Oracle Database 11*g*

An Oracle White Paper
November 2007

ORACLE

NOTE

The following is intended to outline our general product direction. it is intended for information purposes only, and may not be incorporated into any contract. It is not a commitment to deliver any material, code, or functionality, and should not be relied upon in making purchasing decisions. The development, release, and timing of any features or functionality described for Oracle's products remains at the sole discretion of Oracle.

PL/SQL Enhancements in Oracle Database 11*g*

ABSTRACT

Oracle Database 11*g* brings PL/SQL enhancements in these categories:

- transparent and "knob-controlled" performance improvements;
- new language features that you use in your programs to get better performance;
- new language features that bring functionality which you earlier couldn't achieve, or could achieve only with cumbersome workarounds;
- new language features that improve the usability of programming in PL/SQL.

I've made an unashamedly personal selection of some of these features to describe in fair detail in this paper. I used two criteria: my instinct-based ranking according to their relative importance for customers; and my judgment of whether readers might appreciate some background that they're unlikely to find in other accounts. For completeness, I will finish by just listing the other features and giving a skeletal account of each.

MORE ABOUT THE CLASSIFICATION SCHEME

You might notice that this classification scheme works for most previous Oracle Database releases.

Oracle Database 10*g* introduced a brand new optimizing compiler and a freshly tuned execution environment. There could hardly be a better example of a totally transparent performance enhancement; all you have to do is upgrade (which process necessitates recompilation of existing PL/SQL objects) and the time executing your PL/SQL code (not counting the time spend doing SQL) is cut in half.

Oracle9*i* Database introduced native compilation. It did require some DBA set-up — but from the viewpoint of the developer, no code changes were needed. Rather, you had just to recompile chosen units mentioning a new PL/SQL compiler parameter (*Plsql_Code_Type* = *native*). That's what I mean by knob-controlled.

A database version that's no longer supported introduced special syntax for doing SQL select operations and DML operations (insert, delete, and the like) in bulk. These language features boost performance by cutting down the number of context switches between PL/SQL's execution environment and SQL's execution environment. However, developers first had to learn the new syntax and understand the use cases where these language features would deliver benefit; then (for existing units) they had to make a case for code renovation projects. This is what I mean by a performance language feature: its benefit is certainly not brought transparently.

The canonical example of a functionality language feature (again, introduced so long ago that I needn't say when) is the *DBMS_Sql* package. Customers simply have no mechanisms that would allow them to extend PL/SQL (before *DBMS_Sql* was introduced) to allow it to execute SQL statements whose text isn't known until runtime (or SQL statements whose text you know at compile-time but that aren't supported by static SQL). Strictly speaking, an Oracle-supplied package is not a PL/SQL language feature. However, when the package exposes subprograms that wrap implementations that are programmed directly in the code of the Oracle executable (and could not be implemented in PL/SQL alone) the distinction is moot.

Oracle9*i* Database introduced the case expression (in both SQL and PL/SQL) and the *case* statement (in PL/SQL). These language features don't allow developers to program anything that they couldn't write without the new syntax. But the constructs enable developers to think more clearly about the code they're about to write — and, later, to comprehend more easily their own or other developers' code that uses these constructs. This, of course, increases the likelihood of the code being correct. These are examples of usability features.

By the way, new features don't always slot neatly into this scheme. Native dynamic SQL (compared with the pre-existing *DBMS_Sql*) brings a performance benefit, a functionality benefit (it lets you return a *ref cursor* to the database client), and a usability benefit (it's hugely more compact than the equivalent *DBMS_Sql* code).

TRANSPARENT PERFORMANCE IMPROVEMENT: REAL NATIVE COMPILATION

Oracle9*i* Database introduced PL/SQL native compilation; Oracle Database 10*g* brought the storage of the native DLLs in the catalog; and Oracle Database 11*g* brings complete out-of-the-box transparency to the feature. More on that anon. But first, I think that it will help to remind you what PL/SQL native compilation is (and is not), and how the earlier regime (which we informally call C-native) worked.

PL/SQL is famously an interpreted language. That doesn't mean that the source text is processed statement by statement at run time (like in the old days with BASIC); rather, PL/SQL is compiled in the ordinary way into machine code — called M-Code. However, the target machine is virtual: the so-called PVM — for PL/SQL virtual machine. (In this way, it's just like Java with its JVM.) The PVM is implemented as a set of subroutines in the Oracle executable and at run time the M-Code is scanned by another such subroutine. The scanning detects each successive opcode and its operands and then calls the subroutine that implements this opcode with the appropriate actual arguments. Of course, this runtime scanning of the M-Code takes some resources and it is exactly, and only, this that PL/SQL native compilation improves. When you choose *Plsql_Code_Type* = *native*, the compilation of your unit follows exactly the same path as when you choose interpreted right up to the stage when a pre-cursor of the target machine code is available. Then the paths branch: in interpreted mode, the M-Code is produced; and in native mode, a platform-specific DLL is produced which at runtime calls exactly the same PVM subroutines with the same arguments as would have been called by scanning the M-Code. (I'll use DLL as a generic abbreviation for sharable, dynamically linkable library; the file type on Windows systems happens to be *.dll* and on Unix systems it's *.so*.) The difference is that the scanning effort has been moved from runtime to compile time and, of course, this improves runtime performance. Significantly, because exactly the same PVM subroutines are called with exactly the same arguments in both interpreted and native modes, native mode is guaranteed to have exactly the same semantics as interpreted mode. That means that a whole class of question that I periodically hear (for example, *"Is PL/SQL native compilation compatible with shared server mode?"*) has a simple generic answer: *"Yes, because both interpreted and native compilation modes do exactly the same thing at runtime."*

So… where does C come into the picture? Clearly, PL/SQL native compilation has to work on a variety of platforms and this implies a porting effort. In Oracle9*i* Database and in Oracle Database 10*g*, we "outsourced" this effort by getting the Oracle executable to transform the machine code pre-cursor into an equivalent C source text. Then it wrote this out as an operating system file and invoked the C compiler and linker for that platform to generate the required platform-specific DLL. We used this same approach both in Oracle9*i* Database and in Oracle Database 10*g*; the only difference was that in the former, the canonical place of storage was the filesystem (which caused some difficulties with RAC and with backup) but in the latter, the canonical place of storage was the catalog. However, even in the latter, the DLLs had to be materialized on the filesystem both at compilation time and at runtime because the operating system utilities we relied on required this — hence the *Plsql_Native_Library_Dir* and *Plsql_Native_Library_Subdir_Count* initialization parameters. Notice, by the way, that there is no significance in C *per se*; it was just a convenient 3GL in which the

developers at Oracle Corporation were fluent and for which a compiler was readily available on all platforms of interest.

This scheme worked perfectly well from a technical viewpoint but suffered from some serious usability drawbacks: some customers' religion simply forbids the installation of a C compiler on a production machine (and you can't argue with that); other customers were reluctant to pay to license a C compiler (and for some reason mistrusted *gcc*); and other customers found the DBA setup tasks (installing the C compiler, checking the *spnc_commands* file on the *plsql* directory under *Oracle Home*, and setting the *Plsql_Native_Library_Dir* and *Plsql_Native_Library_Subdir_Count* parameters) too daunting. The net result was that PL/SQL native compilation, which delivers an undeniable performance benefit for no developer effort, has been much under-used.

By the way, people often ask *"Will compiling the Oracle-supplied package natively improve their performance?"* It helps to know, first, that most of these are implemented directly in the Oracle executable (which happens to be written in C) and are just exposed via PL/SQL wrappers (as was mentioned in connection with *DBMS_Sql*). In other words, they are already more native than native (if you see what I mean: there's no PVM in the picture) and so it helps only marginally to compile the wrappers natively. (However, it never harms to do this.)

The goal of the "real native" project for Oracle Database 11*g* was therefore to remove the reliance on a C compiler and the operating system linker and loader utilities by programming the Oracle executable to transform the machine-code pre-cursor directly to the required platform-specific machine code. (Of course, this goal implied biting the bullet and doing the task for every platform of interest.) A secondary goal was to speed up compilation time. And a tertiary goal was not to harm runtime performance. The project was a complete success: now the PL/SQL native DLL is never materialized on the filesystem and so the need for managing the *Plsql_Native_Library_Dir* and *Plsql_Native_Library_Subdir_Count* parameters and the contents of the directories they denoted vanishes. Compilation speed has increased by about *2x*. And, far from being harmed, runtime speed has increased noticeably.

PL/SQL native compilation now just has one knob: *Plsql_Code_Type*. The DBA has no configuration duties, and the per-unit decision to compile to a native DLL or to M-Code can be made by the developer secure in the knowledge that the choice will be automatically honored at any installation site.

TRANSPARENT PERFORMANCE IMPROVEMENT: FINE-GRAINED DEPENDENCY TRACKING

Since the dawn of time, PL/SQL programmers have enjoyed a benefit which (if they program only in PL/SQL) they might not even have realized they're getting: they don't have to think about the equivalent of a *make* file. When a system of PL/SQL units and other database objects (tables, views, sequences, and the like) make mutual references, it's sufficient simply to let the source code express these relationships and to create the units in any order. Provided that the source code is error free, then all you have to do is create all the objects and then reference any one of interest. It doesn't matter that, during creation, some end up invalid because you chose an unfortunate order. Oracle Database accesses the dependency metadata that was created as a side-effect of compilation and automatically compiles any invalid units upon which the chosen referred unit (recursively) depends. Later, if you change any unit, then all its dependants are (recursively) invalidated and, on next attempting to reference any unit, the same implicit recursive re-compilation happens again. In other words, Oracle Database maintains an invariant: you cannot successfully reference an object unless it and every object it recursively references is valid and every object has been compiled against the most recent versions of all the objects it references. It achieves this by invalidation and implicit recompilation of invalid objects; either all such compilations are error free, and your reference to the chosen object quietly succeeds; or there is at least one compilation error, your chosen object ends up invalid, and you get a clean runtime error. In other words, Oracle Database guarantees that you cannot get mysterious behavior because a given object invokes a stale version of one of the objects on which it depends. (This liberates you from considering a whole class of potential causes during a debugging exercise.)

This "automatic *make*" environment clearly depends critically on the fact that when an object is created or modified, the list of objects on which it depends is kept up to date. Because of this mission-criticality, Oracle Database has, though Release 10.2, used an approach which obviously guarantees correctness — but at the expense of possible optimizations: dependency information was recorded at the granularity of the whole object. Thus, if you made a change to an existing object that you could reason would not cause you to have to change any of its dependent objects to reflect this change, you nevertheless observed that these dependants were invalidated. (The most obvious example of such an "innocent" change is to add a brand new subprogram — which does not add a new overload for an existing subprogram — to an existing package specification.) Not only are the immediate dependants unnecessarily (from a "logical" viewpoint) invalidated — but also the invalidations cascade. The net result, while correctness is guaranteed, is to consume significant computing resources when an installed application is patched doing compilations that could, in principle, be avoided - in other words, to extend the time for the planned outage of the application.

Oracle Database 11*g* solves this by recording dependency information not at the granularity of the whole object but, rather, at the granularity of the element within the object. For example, if procedure *p1* calls procedure *p2* in package *Pkg*, then the metadata for *p1* will include the fact the dependency is upon *p2* (with a specified list of formal parameters with their datatypes and modes) within package *Pkg*. This means that, now, the addition of a new procedure *p3* to package *Pkg* will not invalidate any of the dependants of *Pkg* — which means in turn that the downtime you need for application patching is reduced. Notice that

the installation of an Oracle Database patchset is allowed to change package specifications but only in a strictly compatible way: in other words, all extant code must work correctly against the new versions of the package specifications without making any code changes. Before Oracle Database 11*g*, the installation of such a patchset could cause massive cascading invalidation; now, such an installation will cause no invalidation. That will lead to a dramatic reduction in the downtime needed to install a database patchset.

Notice, finally, that the new fine-grained dependency tracking does not address online application patching and upgrade. It's easy to see that because one of the most obvious problems you meet if you try to update a stateful package body using one session when another session has instantiated it is the notorious so-called *4068* family of errors; but the invalidation (or even deletion) of a package body famously does not invalidate its specification. The introduction of a rich feature set to support online application upgrade must wait until a future version of Oracle Database.

KNOB-CONTROLLED PERFORMANCE IMPROVEMENT: INTRA-UNIT INLINING

Programmers (in any language) are encouraged to limit the size of the executable section of every subprogram. Opinions vary, but the most common preference seems to be no more than a couple of screenfulls. However, programmers know, also, that there's an inevitable cost associated with a subroutine call. Consider, for example, the task of adding two numbers. Probably every programmer knows that *a := b + c;* will be faster at runtime than *a := Sum(a, b);* where the function *Sum()* does what its name says — and understands why this is so. This understanding can lead to an uncomfortable dilemma: whether to code for clarity (and hence increased likelihood of correctness) or for runtime speed. Suppose you have to scan an English text and count the number of "substantial words" (i.e., excluding words like *a*, *the*, *me*, *you*, *and*, *but*, *to*, *where*, *from*, *under*, and so on). A main loop like this is self-evidently correct:

```
for r in (
  select    a.PK, a.v1
  from      t a
  where     a.PK > Some_Value
  order by  a.PK)
loop
  Process_One_Record(r);
end loop;
```

You can picture how to write *Found_Another_Word()* which scans the input text, extracts the next word, passes this back as an out parameter and returns true until it fails to extract another word. It would use *Instr()* to find the next whitespace and then *Substr()* to extract the characters between the end of the last whitespace and the start of the next. Not rocket science, but definitely the kind of code you'd like to isolate in a dedicated subprogram. A similar argument applies to *Is_Substantial()*. And you can picture, too, how the code would look if you insisted on prioritizing performance and avoided encapsulating the logic of *Found_Another_Word()* and *Is_Substantial()* in subprograms: harder to write and certainly harder for your successor to maintain.

One way to have your cake and eat it is to use a preprocessor before compilation proper that recognizes special mark-up codes in the source text (where otherwise you'd invoke a subprogram) and that substitutes the source text that the mark-up denotes. This approach is called inlining and is familiar to C programmers. We did consider extending PL/SQL's conditional compilation feature to support a similar approach, but we preferred instead to delegate the task to the compiler. Oracle Database 11*g* brings a new pragma inline with allowed values *'yes'* and *'no'*. By writing this:

```
pragma inline(Found_Another_Word, 'yes');
```

immediately before the statement that invokes *Found_Another_Word()*, while *Plsql_Optimize_Level = 2*, the programmer requests that *Found_Another_Word()* should be inlined at this call site. (Inlining never is done when *Plsql_Optimize_Level* is less than *2*.) In certain cases, the request cannot be honored for reasons that are too complex to explain. However, the programmer need never be in any doubt about the outcome of the request. When inlining is done, this is reported using the compiler warnings system like this:

PLW-06005: inlining of call of procedure 'FOUND_ANOTHER_WORD' was done

For the record, the inlining is done not in the regime of the PL/SQL source text but, rather, in the regime of the of the intermediate representation that is

generated from the output of the compiler's syntactic and semantic analysis phase.

Oracle Database 11*g* also brings a new allowed value of *3* for the *Plsql_Optimize_Level* compiler parameter. This instructs the compiler automatically to look for opportunities to inline subprograms at their call sites even when the programmer has not requested this with pragma inline. At this level, *pragma inline(My_Subprogram, 'yes');* requests the compiler to rank this call site high among the candidates for inlining it discovers using is own heuristics. It is at this level that saying *'no'* in the pragma is significant: this is a mandate to the compiler not to inline the subprogram at that call site.

I implemented the algorithm to count the substantial words in a text using the approach that I illustrated above. The helper subprograms were declared locally in the test procedure's declare section. (I won't show you my code; you'll be more convinced if you write your own.) I didn't use the pragma inline statement. I compiled it first using *Plsql_Optimize_Level = 2* and then using level 3. At level 3, the compiler reported *PLW-06005* for the call for each of *Found_Another_Word()* and *Is_Substantial()*. It also reported this:

PLW-06006: uncalled procedure "FOUND_ANOTHER_WORD" is removed.

and the same for *Is_Substantial()*. This is actually an autonomous enhancement to the warnings. You'll get the same warning at any optimization level if you compile a unit that defines a local procedure that you never invoke.

In my test, using a text of about 32k characters randomly selected from the internet, I saw that inlining made the program speed up by getting on for *10x*. Of course, your mileage might vary!

People often ask *"Is inlining a universal panacea? Should I always compile with Plsql_Optimize_Level = 3?"* The answer is that this is not like setting level *2* or like always compiling with *Plsql_Code_Type = native*. These latter choices can never harm runtime performance; the worst you risk is not as much of a gain as you might have hoped for. Of course, both the choices do increase compilation time; but usually it's a simple decision for a production deployment to opt to invest in a cost at compilation time to reap the rewards at runtime. Inlining, on the other hand, does have the potential to harm runtime performance. The explanation is too complex to do justice to in this essay — but, roughly, it has to do with considerations like these: the optimizer is able more easily to deduce the properties of a subprogram — so that it can, for example, safely hoist a call with loop invariant arguments out of a loop — than it is able to deduce corresponding properties of an ordinary stretch of code.

Inlining also has the possibility to cause code bloat (despite the fact that the optimizer has a bloat governor) or, counter-intuitively for the initiated, to reduce code volume. Reduction can occur when an inlined subprogram is called with actual arguments whose values the optimizer knows at compile time and which determine control flow; in such circumstances, branches of code can be eliminated at compile time.

In conclusion then, we recommend experimentation to determine the efficacy of inlining.

PERFORMANCE LANGUAGE FEATURE: THE PL/SQL FUNCTION RESULT CACHE

Oracle Database 11*g* introduces both a SQL query result cache and a PL/SQL function result cache. Both use the same infrastructure and so it's helpful to describe them in one account. First, to understand the benefit of the SQL query result cache, consider a query that has to process a large amount of data to produce a small result set where the underlying data changes infrequently compared to the frequency with which the query is executed — for example, to find the greatest average value of income grouped by state over the whole population — or some similar metric. Even when the SGA is big enough that all the required data blocks can be retained in the block buffer cache, expensive computation (if not disk reads) must be done, before Oracle Database 11*g*, to get the required result each time the query is executed. The SQL query result cache saves this computation by caching the result. The cache is in the shared pool and so results that one session caches benefit other sessions. Of course, the caching is RAC interoperable. The simplest way to request result caching is to add a hint (spelled /*+ *result_cache* */) to the SQL statement. When any of the tables that are involved in determining the result of the query suffer a change, then the cached results are automatically purged.

Now, to understand the benefit of the PL/SQL function result cache, consider a function that executes several SQL queries (where maybe one is parameterized according to the results of an earlier query), and then performs some calculation based on the results. (Assume that it is cumbersome or impossible to express the calculation in a single SQL query.) While it would help just to add the *result_cache* hint to the SQL queries, even more performance benefit is possible by caching the function result. (This avoids the context switch between the PL/SQL and the SQL execution environments.) This is requested by adding the keyword *result_cache* to the function's declaration in the place where other properties like *authid* and *parallel_enable* are specified. However, it is the programmer's responsibility to mention explicitly the list of tables whose data determine the function's result. This is done with the *relies_on* clause. When the function's declaration and definition are separated, as they are with a function declared in a package specification or with a local function that is forward-declared, then we recommend that just the definition use the *relies_on* clause. The definition would look like this:

```
function f(p1 in Typ_1, p2 in Typ_2,...) return Typ_R
  result_cache relies_on(Tab_1, Tab_2, Tab_3,...)
is
  ...
```

Notice that in the SQL case it's enough just to say *result_cache* - the SQL compiler works out what tables are involved in determining the result; but in the PL/SQL case the programmer must supply this information. This reflects the difference between SQL as a declarative language that describes the required result without specifying how it should be calculated and PL/SQL as an imperative language that specifies how a result should be calculated without expressing the intention explicitly.

When a SQL query uses placeholders, then many results are cached for that query keyed by the combination of bind arguments for which the query was executed. Similarly, when a PL/SQL function has formal parameters, then many results are cached for that function, keyed by the combination of actual arguments with which the function was invoked. There are some restrictions on

the allowed datatypes for the formal parameters and the function's return. For example, you cannot have a non-scalar formal parameter (which rule also excludes the *lob* datatypes) or a *lob* return. These restrictions are explained in the documentation.

An obvious candidate for the SQL query result cache is the query that translates a numeric key to the display text that it represents against a list of values table. This use case extends naturally to PL/SQL. I've seen that programmers often use a PL/SQL function to return a result set from the inner join of a data table with one or several partner list of values tables (for example, to list employees with the name of their department). Here, it's quite common to load the list of values data into a package global *index by varchar2* table when the package is instantiated and then to use this to substitute the display text for the key in PL/SQL code following a query against just the single data table. This caching scheme delivers a noticeable performance benefit (which, of course, is why it's popular). However, it suffers from two crucial drawbacks: correctness can only be asserted, rather than guaranteed, by rehearsing an argument about the stability of the list of values table under normal operation; and when each of very many concurrent sessions caches its own copy of the lookup table, then scalability becomes an issue. By writing a result-cached PL/SQL function that, for example, returns the *index by varchar2* lookup table, both these drawbacks are eliminated. Moreover, the benefit of cache management (an LRU-based aging-out policy, and so on) is provided transparently — whereas in the hand-coded approach I've never seen such a scheme coded manually.

As mentioned, both the SQL query result cache and the PL/SQL function result cache use the same infrastructure: dynamically allocated memory from the shared pool, governed by the *Result_Cache_Max_Size* initialization parameter. The *DBMS_Result_Cache* package exposes cache management subprograms (for example, to bypass the cache, to purge it completely, to purge the results for a specified query or function, and to turn off bypass mode). A family of catalog views with names starting with *v$Result_Cache_* (and corresponding *gv$* views for RAC) expose information that allows the performance engineer to decide whether, for each query and function, the caching is paying off. (Of course, if you cache a result that is purged before you access the cached value, or that never is accessed, then the effect will be to decrease performance.)

Engineers in the PL/SQL Team at Oracle Headquarters used the new PL/SQL Hierarchical Profiler (see below) to observe an automated test of an APEX[1] application. This identified many candidate functions for result-caching. A prototype APEX implementation that recompiled the identified functions with the result_cache property (and appropriate relies_on clauses) showed a performance improvement of about 15%.

1. APEX stands for Oracle Application Express.

OTHER NEW PL/SQL FEATURES IN ORACLE DATABASE 11*G*

Functionality language feature: Dynamic SQL completion

The *execute immediate* can take a *clob*. *DBMS_Sql.Parse()* can take a *clob*. *DBMS_Sql* can handle user-defined types in the select list and for binding. You can convert a *DBMS_Sql* numeric cursor to a *ref cursor* (so that you can pass it back to a database client or, when you know the select list at compile time, you can fetch it into a record or bulk fetch it into a collection of records). You can convert a *ref cursor* to a *DBMS_Sql* numeric cursor (so that you can describe it). You now need *DBMS_Sql* only for "method 4" or for arcane requirements like executing DDL across a database link.

Functionality language feature: *DBMS_Sql* security

An attempt to guess the numeric value of an opened (and parsed) *DBMS_Sql* cursor (so that you can unscrupulously hijack it, rebind to see data you shouldn't, and execute it and fetch from it) now causes an error and disables *DBMS_Sql* for that session. The binding and execution subprograms must be invoked by the same *Current_User*, with the same roles, as invoked the parse. Optionally, via a new *DBMS_Sql.Parse()* overload, the invocation of all *DBMS_Sql* subprograms can be restricted in this way.

Functionality language feature: Fine-grained access control for *Utl_TCP*, etc

It's no longer enough to have the *Execute* privilege on *Utl_TCP* and its cousins (*Utl_HTTP*, *Utl_SMTP*, *Utl_Mail*, *Utl_Inaddr*). You must also have the specific privilege to establish a TCP/IP connection to each *node:port* combination that you intend to access. The fine grained privileges are granted by getting the DBA to create an ACL (of users and roles), and to apply this to a *port* (range) on a *node*, using the *DBMS_Network_Acl_Admin* package.

Functionality language feature: Create a disabled trigger; specify trigger firing order

Use the *disable* keyword in the *create* statement to avoid the risk that, if the trigger fails to compile, DDL to the table will cause the *ORA-04098* error. Use the *follows* keyword in the create statement to control the mutual firing order for triggers at the same timing point (e.g., *before each row*). This helps when triggers are generated programmatically.

Functionality language feature: Invoke a specific overridden ADT method

If you don't know what dynamic polymorphism is and don't write hierarchies of ADTs with overridden member subprograms, then this feature won't interest you. Else, read on. New syntax allows the implementation of an overriding subprogram to invoke the member subprogram that it overrides that is defined in a specified supertype. This was the motivating use case; but the same syntax can also be used ordinarily to extend the dot notation that invokes an object's method. The plain dot notation invokes the method as overridden at this object's type level in the hierarchy; the extended notation allows you to invoke a particular overridden method at any level in the hierarchy.

Usability language feature: The compound trigger

A well known paradigm uses a *before statement* trigger, a *before each row* or *after each row* trigger, an *after statement* trigger, a package specification, and its body in concert. The paradigm is useful when you want to "notice" activity for each row the firing DML affects but act upon this only periodically. This might be to improve efficiency by noticing auditing information, recording this in a package global collection, and bulk inserting it into the audit table; or it might be to avoid the mutating table error. The compound trigger is a single database object that "feels" rather like a package body and lets to you implement the same paradigm in one unit. It lets you implement a "pseudo-procedure" for each of the four table timing points and declare state that each of these pseudo-procedures can see. I called them pseudo-procedures because they look very rather like procedures but they are named using keywords. Like this:

```
before each row is
begin
  ...
end before each row
```

The state is managed automatically to have the lifetime of the firing DML statement rather than that of the session.

Usability language feature: Use *My_Sequence.Nextval* in a PL/SQL expression

You no longer need to say select *My_Sequence.Nextval into v from Dual;* The same goes for *Currval*.

Usability language feature: The *continue* statement

PL/SQL catches up with other languages. The *continue* has a familial relationship to *exit*. Both can be written unconditionally or together with a *boolean* expression as the argument of *when*; and both can specify the name of an enclosing loop. The difference is that *exit* abandons the nominated loop but *continue* starts the next iteration of that loop.

Usability language feature: Named and mixed notation from SQL

PL/SQL programmers know the value of invoking a subprogram that has a very large number of defaulted formal parameters by writing

Interesting_Formal=>*Interesting_Actual*

only for those formal parameters whose default they don't want to accept. This value has been unavailable when a PL/SQL function is invoked in a SQL statement. Now this restriction is removed.

Usability language feature: *"when others then null"* and other compile-time warnings

Naïve programmers have a tendency to provide a *when others then null* exception handler for every subprogram because they believe that it's wrong to cause an error (yes, really - read Tom Kyte on this topic). Of course, when a subprogram quietly returns after suppressing an unknown error, and the caller blunders on in blissful ignorance, then anything might happen — and debugging is a nightmare.

It would be very labor intensive, and error-prone, to attempt an analysis of extant code to look for this dangerous locution. Now, the compiler spots if a when others handler does not unconditionally re-raise the exception or raise a new one; in this case it issues *PLW-06009*.

Other new warnings are implemented, for example *PLW-06006*, as has been discussed above.

Usability language feature: *Regexp_Count()* in SQL and PL/SQL

This is just what its name implies. It would be tedious to write code to produce the same answer without this new function.

TOOLS

The PL/SQL Hierarchical Profiler

Oracle Database 11*g* brings a new hierarchical performance profiler for database PL/SQL. In very broad terms, you use it like you use SQL Trace: turn on data collection, run the code, turn off data collection, and format the raw data for human analysis. However, because of the richness of the information gathered, the formatter stores its output in database tables or generates a set of hyperlinked HTML pages. You can then answer questions like these: show me the list of subprograms and SQL statements that were executed during the run, ordered by the elapsed time; for a particular subprogram, show me the time spent in itself and the time spent in each of the subprograms it calls; for a particular subprogram, show me the list of subprograms that call it ordered by the total time for those calls. This information guides you efficiently to the code whose optimization will have the greatest effect.

PL/Scope

Oracle Database 11*g* introduces a new compiler parameter to control the generation of metadata about the identifiers in a unit. If you compile a unit with *PLScope_Settings = 'identifiers:all'* then you can query one of the *DBA_Identifiers* catalog view family to get the line and column location for the reference to each identifier and to find out where each identifier is defined (which might be in a different unit, of course). Because PL/SQL is a block structured language, a particular name often corresponds to a different phenomenon at different locations in a unit. Therefore, ordinary text search is a feeble tool to discover where a given identifier is defined or where it is referenced; rather, you need the full intelligence of the PL/SQL compiler. You can use the *DBA_Identifiers* views to support ad hoc impact analysis queries if, for example, you discover that two subprograms have been given the same name and you want to rename one to lesson lessen the confusion this causes to the programmer (but, of course, not to the compiler).

However, this feature was motivated by a different goal: to enable SQL Developer to present source code where the identifier usages hyperlink to their definitions, where identifier definitions hyperlink to lists of hyperlinks to their usages, and so on. This facility is analogous to *Cscope* (see *cscope.sourceforge.net*) which C programmers use. For this reason, we call the feature PL/Scope . Such a facility is especially useful to the programmer who inherits a large corpus of unfamiliar code.

CONCLUSION

Oracle Database 11*g* brings many enhancements to PL/SQL. These will improve the performance, functionality, and security of your applications and will increase your productivity as a developer.

Bryn Llewellyn,
PL/SQL Product Manager, Oracle Headquarters
bryn.llewellyn@oracle.com
12-November-2007

PL/SQL Enhancements in Oracle Database 11*g*
November 2007
Bryn Llewellyn, PL/SQL Product Manager, Oracle Headquarters

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
oracle.com