
ORACLE®

Doing PL/SQL from SQL: Correctness and Performance

Bryn Llewellyn
Distinguished Product Manager
Database Division
Oracle HQ
twitter: @BrynLite

Fall 2014

Safe Harbor Statement

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.

Abstract

- A SQL statement that calls a PL/SQL function is slower than one that defines the same result by using only SQL expressions. The culprit is the SQL to PL/SQL round-trip.
- Its cost can be reduced by caching: by invoking it in a scalar subquery, by marking the function deterministic, by using the PL/SQL function result cache, or by some other scheme.
- With caching, you can't predict the result set if the function isn't pure, so it would seem that caching should be used with caution.
- This session shows that a function must have certain properties to be safe in SQL, even without caching—in other words that the possible semantic effect of caching is a red herring.
- You can't rely on how often, or in what order, an uncached function is called any more than you can rely on group by to do ordering.

ORACLE

Agenda

1. Two words for the same thing: *pure* and *deterministic*
2. Nonsense when SQL invokes a PL/SQL function that isn't *statement-duration-pure*
3. Three ways to get nonsense: notice "ordinary" side effects; do SQL from PL/SQL; observe the "outside world"
4. We must avoid nonsense. When we do, any caching scheme is safe
5. What caching schemes are available? How do they work? Is caching guaranteed to improve performance?
6. A better way than caching to improve performance of PL/SQL from SQL

ORACLE

Two words for the same thing: *pure* and *deterministic*

pure is generic notion, applicable to all programming languages

- A pure function is boringly predictable:
  the input values uniquely determine the output
- A pure function is environmentally friendly:
  it "leaves no trace"
- This PL/SQL function (exposed in the spec of in some package) is impure:

```
function Impure(i in integer) return integer is
begin
  Global := Global + 1;
  return case
           when i > Global then i
           else                 2*i
         end;
end Impure;
```

Do not try this at home!

deterministic is a PL/SQL keyword

- The author chooses whether or not mark a function deterministic
- The marking asserts that the mapping between input and output can be safely denormalized in an index* – and an index can endure indefinitely
- The denormalized mapping is used to compute restrictions, and may be used to supply selected values
- A deterministic function, therefore, cannot have side effects, even if these don't influence the mapping – because they might not happen
- *deterministic* promises purity until you compile again

** of course, an index is a kind of cache*

When to use *deterministic*? That's a no-brainer

- When you, the author, know that the function you wrote IS *pure* unless/until you re-compile it with a new, impure implementation, mark it deterministic
  - doing so, when it's true, cannot be harmful and might help performance
- When you know that the function you wrote is NOT *pure* on a long time-scale (even if it might be pure for a short time scale) do not mark it deterministic
  - doing so when it isn't true can be harmful by giving you nonsense

What do I mean by *statement-duration-pure*?

- This PL/SQL function (exposed in the spec of in some package) might be statement-duration-pure:

```
function Candidate(i in integer) return integer is
begin
  return case
           when i > Global then i
           else                 2*i
         end;
end Candidate;
```

Caveat emptor

- Here's more of the body:

```
Global integer := -5;

function Candidate(i in integer) return integer is
begin
  return case
           when i > Global then i
           else                 2*i
         end;
end Candidate;

function Silly_Device(i in integer) return integer is
begin
  Global := Global + 2;
  return i;
end Silly_Device;
```

Caveat emptor

- What happens here:

```
select Pkg.Candidate(c1), Pkg.Silly_Device(c2) from t ...
```

  or here:

```
select Pkg.Candidate(c1) from t
where Pkg.Silly_Device(c2) = c3 and ...
```

- On the other hand, following this :

```
begin Pkg.Set_Global(10); end; -- with the obvious meaning
```

  The outcome of this is boringly predictable:

```
select c1, Pkg.Candidate(c2) a from t order by a
```

statement-duration-purity is a property of the current use

- The *deterministic* quality of a PL/SQL function is an intrinsic and permanent property of the function (until/unless it's recompiled)
- But *statement-duration-purity* is *not* a property of a PL/SQL function
- Rather, it's a property of the *use* of a PL/SQL function in a particular SQL statement
- When *Pkg.Candidate()* is used in the same SQL statement as *Pkg.Silly_Device()*, neither the use of the former, nor the latter is pure:
  - the result of *Candidate()* is not uniquely determined by the inputs
  - When *Silly_Device()* is called, there's a side-effect
- When *Candidate()* is used alone in a SQL statement, it is pure – for the *duration* of that SQL statement

Agenda

1. Two words for the same thing: *pure* and *deterministic*
2. Nonsense when SQL invokes a PL/SQL function that isn't *statement-duration-pure*
3. Three ways to get nonsense: notice "ordinary" side effects; do SQL from PL/SQL; observe the "outside world"
4. We must avoid nonsense. When we do, any caching scheme is safe
5. What caching schemes are available? How do they work? Is caching guaranteed to improve performance?
6. A better way than caching to improve performance of PL/SQL from SQL

ORACLE

Nonsense when SQL invokes a PL/SQL function that isn't *statement-duration-pure*

Don’t fool yourself

- If you knew the order of invocation, in a SQL statement, of an impure PL/SQL function, and the number of times it would be invoked, then you could predict the outcome
- But in general, you’d need a brain the size of a planet to think it through
- However, Oracle makes no promise about those two preconditions for prediction
- You might think that, at least, the number of invocations must be predetermined by the data
- You might think that you can control order by larding up your statement with a gazillion nested in-line views, each with its own “order by”
- But you’d be wrong

Don’t fool yourself

- It means nothing if you do a gazillion experiments that back up your notion that order, and number of invocations, are predictable
- Oracle’s promise is all that matters
- And it makes no promise!
- If you can’t predict the outcome of a SQL statement, when you know its input data, then your SQL is valueless, and you may as well simplify the nonsense to this:

```
select DBMS_Random.Value() from Dual
```

Agenda

1. Two words for the same thing: *pure* and *deterministic*
2. Nonsense when SQL invokes a PL/SQL function that isn't *statement-duration-pure*
3. Three ways to get nonsense: notice "ordinary" side effects; do SQL from PL/SQL; observe the "outside world"
4. We must avoid nonsense. When we do, any caching scheme is safe
5. What caching schemes are available? How do they work? Is caching guaranteed to improve performance?
6. A better way than caching to improve performance of PL/SQL from SQL

ORACLE

Three ways to get nonsense:
notice “ordinary” side effects;
do SQL from PL/SQL;
observe the “outside world”

first: Noticing "ordinary" side effects

- Sensitivity to a package global is "ordinary"
- A package global cannot change over statement duration unless you willfully write code to make this happen
- We've done this point to death
- I seriously cannot imagine why you would even think of writing code like I've shown unless you were doing a presentation like this
- The moral dilemmas provoked by the discussion should never trouble you in the real word
- Any ordinary, self-contained PL/SQL function (does no SQL, doesn't look at the outside world), that you will plausibly invoke from SQL will be safe to cache

ORACLE

second: Doing SQL from the PL/SQL that you call from SQL

- Oracle is famous for the proposition that the results of a query represent the subset of the database that it defines, as this exists at the instant your query starts
- The rows you get, and the values they contain, are guaranteed to be consistent with the enforced data rules
- This is hugely important when other sessions change the data while your query is running – we call it read consistency!
- When you do SQL from the PL/SQL that you call from SQL, each "recursive" SQL runs in its own snapshot
- With this clue, you can easily contrive a testcase that produces a result set that could never have existed!

How to “tame” this effect

- Who wants a result set that never could have existed? Not me!
- Why are you doing this anyway? I don’t know!
- Maybe you think that by marking the PL/SQL functions *result_cache*, you’ll improve performance
- Notice that the invalidation mechanisms for the result cache guarantee that the semantics of a function are unaffected by the use of the hint
- If you want meaningless results, there are yet faster ways
- So let’s get back to semantics:
  - make all the tables upon which the result-cached functions depend read-only
  - or ensure, more explicitly, that all queries run with the same SCN

Using a read-only table

- A PL/SQL unit *p* that depends on table *t*, presently read-only, is not invalidated by this DDL:

```
alter table t read write
```

  and nor does altering *t* to read-only from read-write invalidate *p*

- Presumably, you call PL/SQL from SQL that, in turn, does recursive SQL, the recursive SQL accesses only "static" configuration data – and you intend to cache it
- Enforce this formally using read-only. If you want to change the configuration data, well… that's an application patch
  - so take downtime to do it
  - or use EBR to do it with zero downtime

(yes, you *can* make an editioning view read-only)

ORACLE

Run the outer SQL and the recursive SQL at the same SCN

- Here's a simple, after-the-fact, approach:

```
begin
  DBMS_Flashback.Enable_At_System_Change_Number(
    DBMS_Flashback.Get_System_Change_Number());
end;
```

then do your query, or queries. Then allow changes again:

```
begin
  DBMS_Flashback.Disable();
end;
```

- This won't allow insert-select, 'cos you can't make changes when you've frozen the SCN. So, if you *really must*, then say:

```
insert... select f1(:SCN,...) ... where f2(:SCN,...) as of scn :SCN
```

where, of course, *f1()* and *f2()'s SQL* use the same *as of scn :SCN* decoration

third: Observing the "outside world"

- The canonical example is stock prices, or the like, got from some URL
- By all means, encapsulate this in a function. The proposition is inherently impure. So feel free to invent whatever *ad hoc* scheme, probably using caching of some kind, that suits you
- For example, the function can guard a table of key-value pairs. If the key you want isn't there, the function goes to the URL, gets the value, and inserts the pair into the table before returning the value. You'll have some or other business rule to tell you when to truncate the table and start again
- This is all about the business rules that make the best they can of a messy world. It's not about "correctness" in the proper sense of the word. And it's not about the mechanism. By all means, choose the result cache, with manual invalidation

Agenda

1. Two words for the same thing: *pure* and *deterministic*
2. Nonsense when SQL invokes a PL/SQL function that isn't *statement-duration-pure*
3. Three ways to get nonsense: notice "ordinary" side effects; do SQL from PL/SQL; observe the "outside world"
4. We must avoid nonsense. When we do, any caching scheme is safe
5. What caching schemes are available? How do they work? Is caching guaranteed to improve performance?
6. A better way than caching to improve performance of PL/SQL from SQL

ORACLE

Do you get it now?

- Use only PL/SQL functions that are statement-duration-pure in your SQL statements
- Then choose to cache, or not, after thinking about likely cache-hit ratios

Agenda

1. Two words for the same thing: *pure* and *deterministic*
2. Nonsense when SQL invokes a PL/SQL function that isn't *statement-duration-pure*
3. Three ways to get nonsense: notice "ordinary" side effects; do SQL from PL/SQL; observe the "outside world"
4. We must avoid nonsense. When we do, any caching scheme is safe
5. What caching schemes are available? How do they work? Is caching guaranteed to improve performance?
6. A better way than caching to improve performance of PL/SQL from SQL

What caching schemes are available?

How do they work?

Is caching guaranteed
to improve performance?

first: Ordinarily in PL/SQL

- I hope that most of you already follow the same religion as me:
  - expose your Oracle Database application backend to client-side code only through a PL/SQL API that models the supported business functions (and only those), hiding all the tables, and the *select*, *insert*, *update*, and *delete* statements that manipulate them, behind your API
- Sometimes, your function return will be set of values for populating a screen's pick list, or a list of items in a shopping card
- This is an ancient, much-loved use case.
  - Performance can be improved by caching the LoV key-value pairs in package globals (a session-duration cache) and using the information to translate the SQL results from a single table to a human-readable version for presentation *in PL/SQL land*
- This is the use-case that motivated the PL/SQL function result cache

(more on this presently)

second: SQL execution's statement duration cache

- Your capacity to enjoy deferred gratification is finally rewarded!
- We will now, at last, look at the caching mechanism behind the two quite different code locutions that ask for (well… allow, anyway) under-the-covers statement duration caching, managed by Oracle
- This (the *scalar subquery*):

```
select (select Plain_f(t.c1) from Dual) a from t ...
```

- And, using this:

```
function Deterministic_f(i in integer) return integer deterministic
```

  this simpler SQL:

```
select Deterministic_f(t.c1) a from t ...
```

SQL execution's statement duration cache

- This topic has attracted endless empirical investigation, speculation, blogging, and conference presentation
- Guess what: you cannot establish a predictive model of a software system by empirical investigation
- Rather, you have to study the code
- Better yet, interrogate someone who understands the code and can explain the design goals that the code was written to meet
- You can't do that. I can. And I did. These experts all work on the same floor as me

Do you get it now?

The goal was "do no harm"

- don't change semantics
- don't harm performance

SQL execution's statement duration cache

- We have a single (C code) subsystem that can allocate and manage a cache
- To date, there are two clients
  - subqueries
    (scalar subqueries are just an example; but it's these that are of interest in this talk)
  - functions marked *deterministic*
- Because it's common code, we need to describe the caching scheme just once
- But each client is free to add extra nuances – and does

SQL execution's statement duration cache

- The cache is allocated for a particular use by a particular statement
- If you invoke a deterministic function in a scalar subquery, you'll get two caches. This won't harm you. But what were you thinking?
- The cache will accommodate a smallish number of key-value pairs (typically on the order of a couple of hundred entries)
- The number of slots is computed using the worst case estimate for the size of a key-value pair, and the allowed cache size
- Don't ask how to change the cache size. That's asking to harm performance.
- When a key is presented, it's hashed and used to address a slot (instantiated on demand). The hashing scheme never produces more values than the computed number of slots, and there might be collisions
- If a new key hash-collides, or if all slots are full, then the value isn't cached (there's no eviction)

Each client adds its own nuances

- Usage for a subquery
  - statement duration
  - there's a special extra slot for the last-seen key-value pair (eviction in action uniquely here)
  - never gives up
- Usage for a deterministic function
  - call duration (some of you spell this "fetch" duration)
  - no special extra slot
  - gives up after too many cache misses
- Why? We expect you to say *deterministic* when it's true. So the caching that this silently allows must not harm performance. But you *ask* for scalar subquery caching when you've understood the cardinality of your data

An obvious test of the emergent performance effect

- The table has a unique key, *PK*, and a low-cardinality column, *Num*, with 100 distinct values. It has 10 million rows

```
                                                          seconds
                                                  (on my laptop)
select Sum(                PK                  ) from t    0.3
select Sum(     (select PK from Dual)           ) from t    1.6

select Sum(            Sqrt(PK)                ) from t    3.2
select Sum(   (select Sqrt(PK) from Dual)      ) from t    4.4

select Sum(            Sqrt(Num)               ) from t    3.2
select Sum(   (select Sqrt(Num) from Dual)     ) from t    1.6
```

- Is anyone surprised? Let this common sense inform your thinking, rather than obsessing about the internals

third: the PL/SQL function result cache

- It helps to know why this was invented
- E-Business Suite engineers had made widespread use of the scheme that uses package-global PL/SQL index-by tables to translate numeric foreign keys to LoV tables into the human-readable description needed for display, while the data passed from tables, though PL/SQL, to client code
- With very many concurrent sessions, they suffered from excessive session memory consumption
- They needed a better way
- The instance-wide result-cache infrastructure was already under development for SQL subqueries
- No-one asked for caching for PL/SQL functions, called from SQL, that do SQL

The instance-wide result cache

- Like the statement (or call) duration cache, it has two clients: SQL subqueries and PL/SQL functions
- All cached items compete for space, and there's a global LRU
- As with all caches, if your pattern does not re-use cached items, your performance will suffer – so it's an "opt in" model
- Its value is brought when the PL/SQL does something really heavy-weight – i.e. SQL. PL/SQL that does SQL isn't even statement-duration-pure, so not a candidate for marking *deterministic*
- It's hard to imagine how a PL/SQL function that stays in PL/SQL and does a really time-consuming computation would be involved in a SQL statement. But if you have such a function, it's bound to be deterministic (else nonsense). So feel free: mark it both *deterministic* and *result_cache*

The PL/SQL function result cache

- If you really have a convincing reason to invoke a PL/SQL function from SQL that itself does SQL, then you must already have ensured (for example by caching only read-only tables) that your read consistency is sound
- While you mustn't lie by marking it deterministic, you may then, indicate what you know about the use of your function in the present SQL statement by using the scalar subquery locution
- Normal reasoning about cardinality applies – deluxe, 'cos you have a first-level cache and a second-level cash. Two level caching schemes are not unheard of...

Agenda

1. Two words for the same thing: *pure* and *deterministic*
2. Nonsense when SQL invokes a PL/SQL function that isn't *statement-duration-pure*
3. Three ways to get nonsense: notice "ordinary" side effects; do SQL from PL/SQL; observe the "outside world"
4. We must avoid nonsense. When we do, any caching scheme is safe
5. What caching schemes are available? How do they work? Is caching guaranteed to improve performance?
6. A better way than caching to improve performance of PL/SQL from SQL

ORACLE

A better way than caching to improve performance of PL/SQL from SQL

A better way than caching...

- The overwhelmingly common case, when PL/SQL is called from SQL, is that it's small and self-contained. For example, an encapsulation to some division by 100.0, *To_Char()*, *Lpad()*, and the like – or the famous Tom Kyte *Is_Number()*
- Such things are bound to be statement-duration-pure (but we've done that to death now)
- Oracle Database 12*c* brought a new way to compile PL/SQL that benefits units that don't call to other PL/SQL (and certainly don't do SQL)
- You ask for it explicitly with *pragma UDF* for a schema-level unit, or implicitly by defining the whole function in a *with* clause
- For functions that this benefits, this approach beats statement duration caching

Another obvious test of the emergent performance effect

- The same table as before: a unique key, *PK*, and a low-cardinality column, *Num*, with 100 distinct values. It has 10 million rows

		seconds (on my laptop)
`select Sum(`	**`Pkg.f(PK)`**	`) from t` **`8.9`**
`select Sum(`	**`Pkg.f_Deterministic(PK`**	`) from t` **`8.9`**
`select Sum(`	**`(select Pkg.f(PK) from Dual)`**	`) from t` **`10.8`**
`select Sum(`	**`Pkg.f_UDF(PK)`**	`) from t` **`1.7`**
`select Sum(`	**`Pkg.f(Num)`**	`) from t` **`9.0`**
`select Sum(`	**`Pkg.f_Deterministic (Num)`**	`) from t` **`2.0`**
`select Sum(`	**`(select Pkg.f(Num) from Dual)`**	`) from t` **`1.7`**
`select Sum(`	**`Pkg.f_UDF(Num)`**	`) from t` **`1.7`**

- Do you agree that statement-duration caching does no harm?

Do you get it now?

Do you *really* get it?

I hope so!

Summary

Summary

- A PL/SQL function must be *statement-duration-pure* if it is to be called from SQL
- In most cases, this simply falls out of what you want to write
- Mark it *deterministic* when it is (SSQ when not? Hmm…)
- By its very nature, it's likely to be self-contained – so use *pragma UDF*, or equivalently write its whole definition in the *with* clause
- You might well and up with a function that *both* is marked *deterministic and* has *pragma UDF*. Don't worry. The performance benefit of the SQL-friendly compilation mode is not compromised by computations done for caching
- *pragma UDF* is the hands-down winner 'cos its benefit is *independent of cardinality*

Summary

- If you find a potential use for a PL/SQL function that does SQL, and that is called from SQL, think hard… very hard
- Might you, after all, express the whole thing in SQL?
- Are you confident that you're not getting nonsense by violation read-consistency
- Then, and only then, go ahead
- Result-cache it, or not, in the light of cardinality considerations
- Use the scalar subquery locution as well when you expect to meet only hundreds of distinct values in any one statement execution

Hardware and Software
Engineered to Work Together