HrOUG – Spring 2022

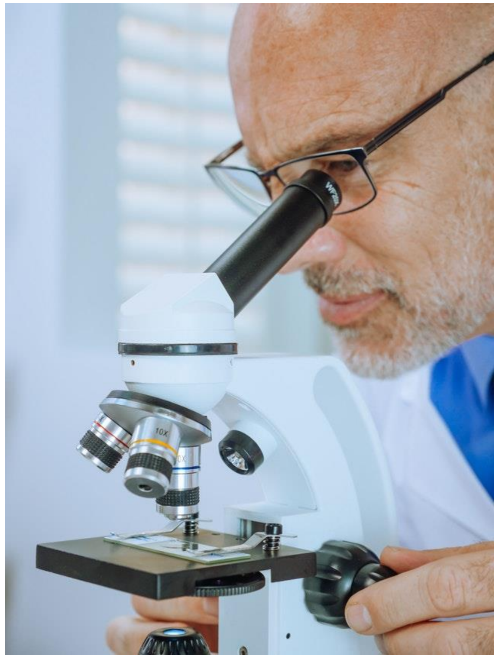
Speeding Up Executions of User-Defined Functions in SQL

Erik van Roon

EvROCS
COMPLETING THE PUZZLE

Who Am I?

Erik van Roon



1995

ORACLE DATABASE

>=Oracle5

2009

EvROCS

COMPLETING THE PUZZLE

_SYM42

https://sym42.org/

MASH

Core team

MASH Program

ORACLE ACE Pro

: erik.van.roon@evrocs.nl

: www.evrocs.nl

: @evrocs_nl

MASH PROGRAM

Mentor and Speaker Hub

Our goal is to *connect* speakers with mentors to assist in *preparing* technical sessions and *improving* presentation skills

Interested? Read more and get in touch

https://mashprogram.wordpress.com

ORACLE
ACE Program

500+ technical experts helping peers globally

The **Oracle ACE Program** recognizes and rewards community members for their technical contributions in the Oracle community

3 membership tiers

ORACLE ACE Director | ORACLE ACE | ORACLE ACE Associate

For more details on Oracle ACE Program:
bit.ly/OracleACEProgram

Oracle Groundbreakers

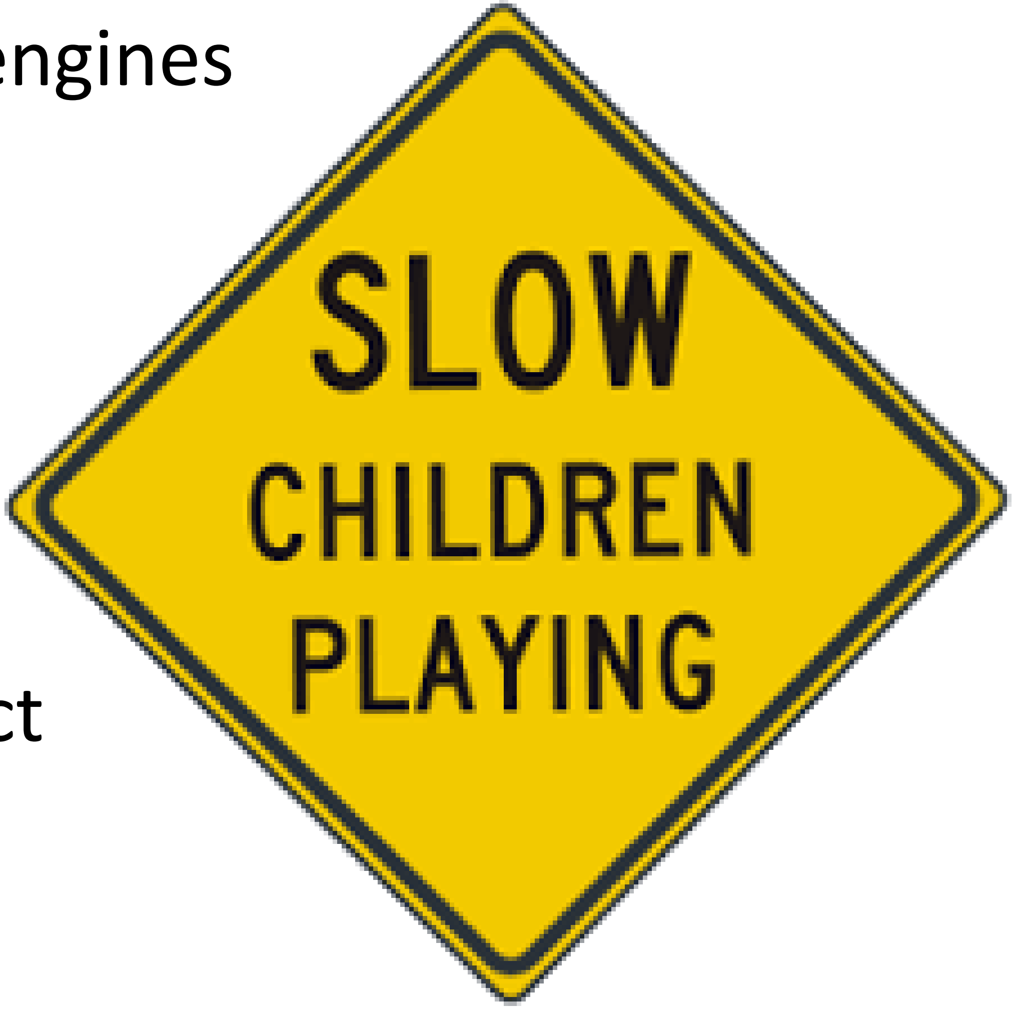
Nominate
yourself or someone you know:
acenomination.oracle.com

Connect: **oracle-ace_ww@oracle.com** Facebook.com/oracleaces @oracleace

What’s the problem?

The problem:

- SQL and PLSQL have their own engines
- Calling one from the other introduces context switches
- Doing it often has
  a noticeable performance impact



For SQL calls from PLSQL we have Bulk Operations

- Bulk Collect
- Forall

For PLSQL calls from SQL we didn’t have such tricks

Our options before 12c (and still today)

Pre 12c we could:

1. Obviously: not do PLSQL if we don’t have to
2. Obviously: Make our PLSQL as efficient as possible
3. Declare the function **Deterministic**
4. Make use of the **Function Result Cache**
5. Make use of **Scalar Subquery Caching**

Option 1 & 2:

- Always a good idea

Option 3-5:

- Only faster when function is called often with the same parameters
- Executions that are not skipped will not be faster

Introduced in 12c:
Functions in the WITH clause

PLSQL in Subquery Factoring (the With-Clause)

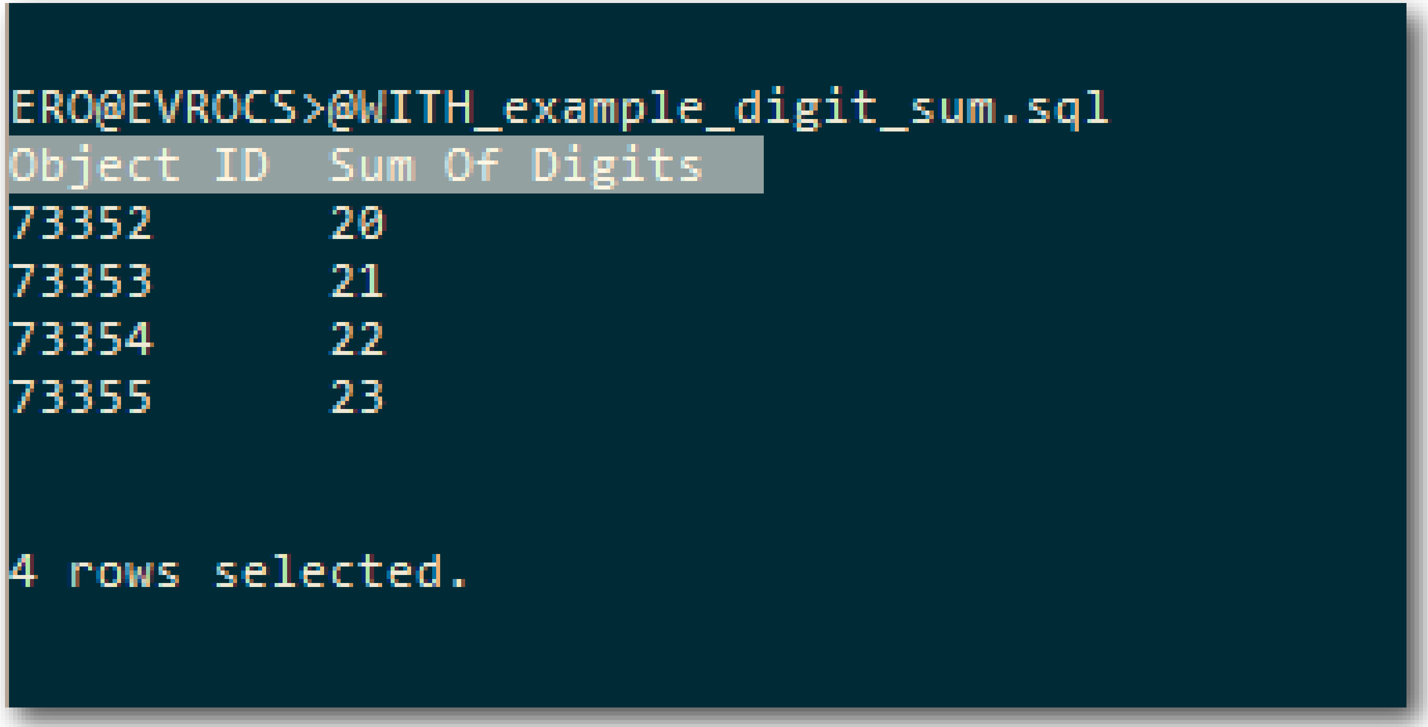
```
with
  function digit_sum
     (pi_integer  in  integer)
     return integer
     is
       c_as_string constant varchar2(15) := to_char(pi_integer,'fm999999999999999');
       l_result             integer   := 0;
     begin
       for i_pos in 1 .. length(c_as_string)
       loop
         l_result := l_result + to_number(substr(c_as_string, i_pos, 1));
       end loop;
       return (l_result);
     end;
--
select object_id                       "Object ID"
,      digit_sum (object_id)           "Sum Of Digits"
from   user_objects
fetch first 4 rows only
/
```

Must be "/", not ";" in SQL*Plus and SQLcl

WITH_example_digit_sum.sql

PLSQL in Subquery Factoring (the With-Clause)

Result:



```
ERO@EVROCS>@WITH_example_digit_sum.sql
Object ID   Sum Of Digits
73352       20
73353       21
73354       22
73355       23


4 rows selected.
```

WARNING

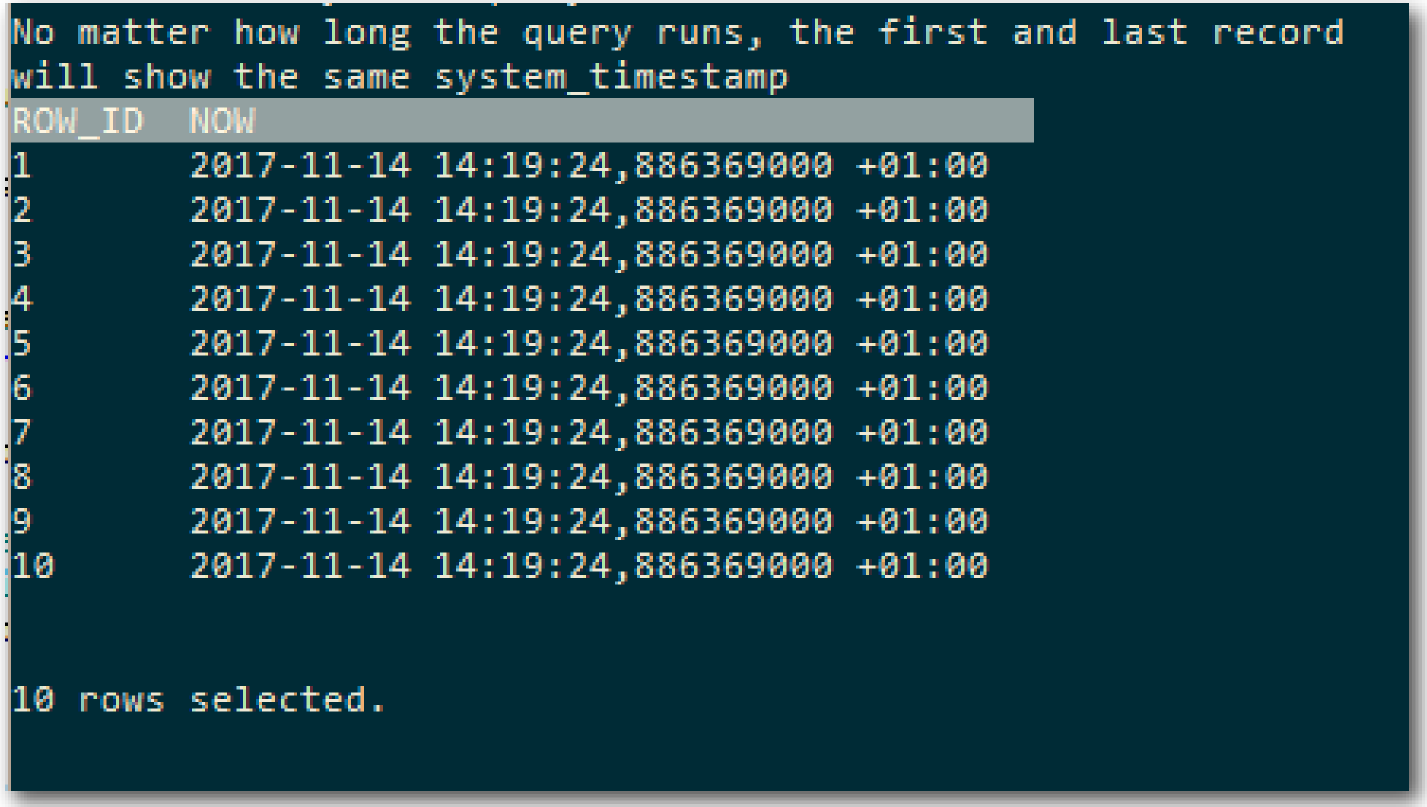
If a function is declared right there in the query
the mistake is easily made to think it’s **part of** the query
However:

This plsql is NOT part of the read consistency

True for ANY PLSQL called from SQL

Get a timestamp from a **query** in the with clause

```
with
  timestamp_query as
    (select systimestamp
     from   dual
    )
select level     row_id
,      (select systimestamp
        from timestamp_query
       )         now
from   dual
connect by level <= 10
;
```

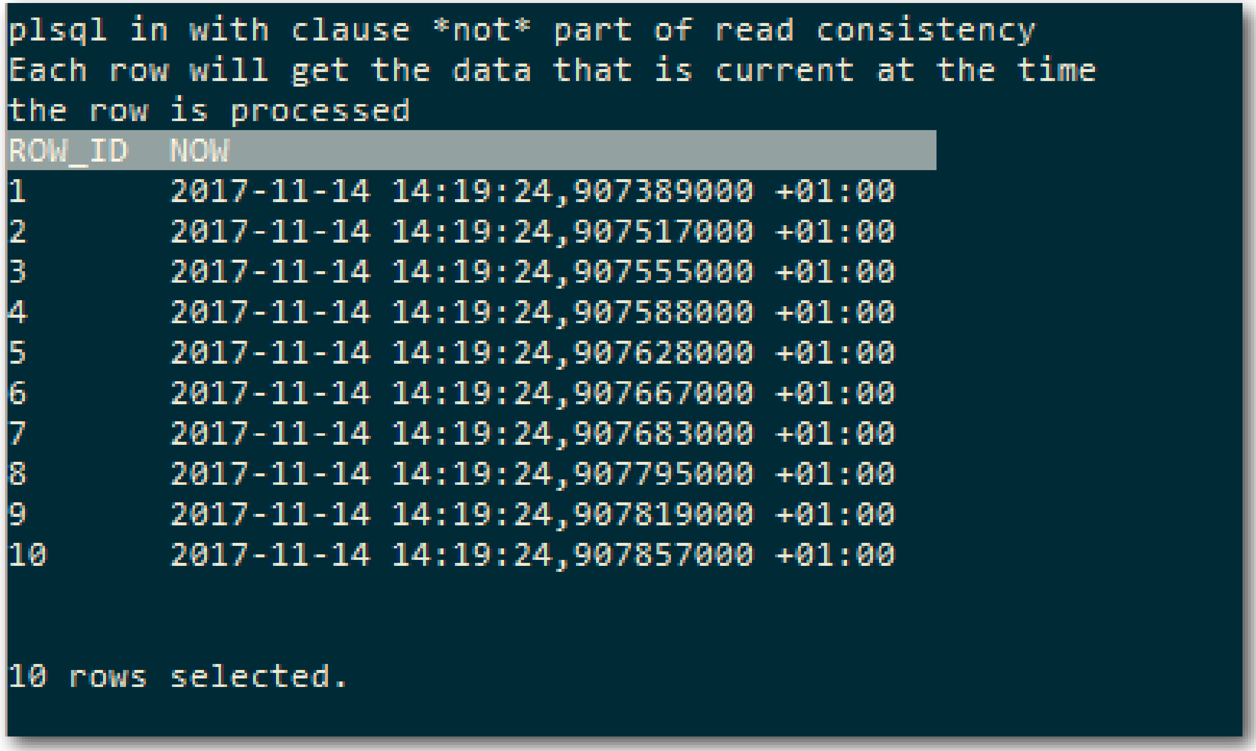


```
No matter how long the query runs, the first and last record
will show the same system_timestamp
ROW_ID  NOW
1       2017-11-14 14:19:24,886369000 +01:00
2       2017-11-14 14:19:24,886369000 +01:00
3       2017-11-14 14:19:24,886369000 +01:00
4       2017-11-14 14:19:24,886369000 +01:00
5       2017-11-14 14:19:24,886369000 +01:00
6       2017-11-14 14:19:24,886369000 +01:00
7       2017-11-14 14:19:24,886369000 +01:00
8       2017-11-14 14:19:24,886369000 +01:00
9       2017-11-14 14:19:24,886369000 +01:00
10      2017-11-14 14:19:24,886369000 +01:00


10 rows selected.
```

WITH_example_read_consistency.sql

Get a timestamp from a **Function** in the with clause

```
with
  function timestamp_data
    return timestamp with time zone
    is
      l_result  timestamp with time zone;
    begin
      select systimestamp
      into   l_result
      from   dual;
      return l_result;
    end;
select level                    row_id
,      timestamp_data           now
from   dual
connect by level <= 10
/
```



```
plsql in with clause *not* part of read consistency
Each row will get the data that is current at the time
the row is processed
ROW_ID  NOW
1       2017-11-14 14:19:24,907389000 +01:00
2       2017-11-14 14:19:24,907517000 +01:00
3       2017-11-14 14:19:24,907555000 +01:00
4       2017-11-14 14:19:24,907588000 +01:00
5       2017-11-14 14:19:24,907628000 +01:00
6       2017-11-14 14:19:24,907667000 +01:00
7       2017-11-14 14:19:24,907683000 +01:00
8       2017-11-14 14:19:24,907795000 +01:00
9       2017-11-14 14:19:24,907819000 +01:00
10      2017-11-14 14:19:24,907857000 +01:00


10 rows selected.
```

WITH_example_read_consistency.sql

The PLSQL can also be defined in for example:

- The with clause of a subquery
- The with clause of the query in “insert into .. select from ..”
- The with clause of "using()" query of a merge statement

If it's NOT in the with clause of the top-level query
We need to warn the compiler that it will run into PLSQL

Use the hint:

```
/*+ WITH_PLSQL */
```

For example PLSQL defined in select of an insert:

```
insert
into   my_table (col1, col2)
with
  function do_something
    return ....
    is
    begin
      return (...);
    end;
select source_data
,      do_something
from   other_table
/
```

will result in
ORA-32034: unsupported use of WITH clause

```
insert --+ with_plsql
into   my_table (col1, col2)
with
  function do_something
    return ....
    is
    begin
      return (...);
    end;
select source_data
,      do_something
from   other_table
/
```

will be successful

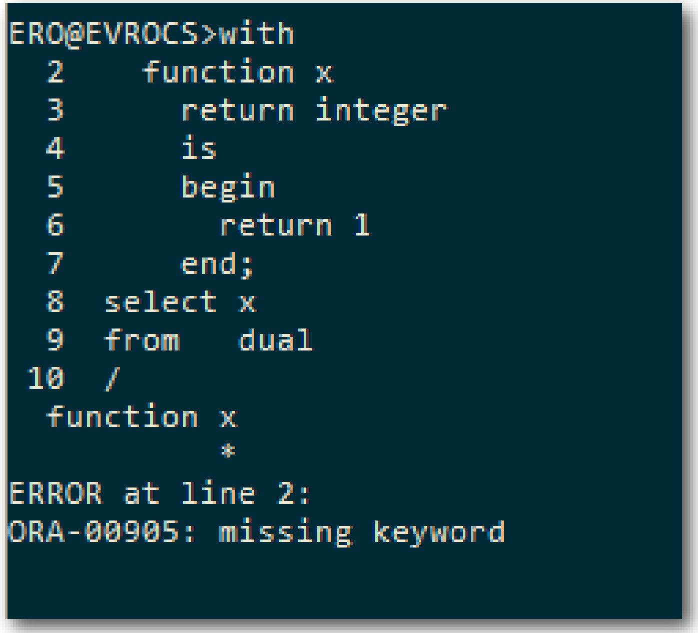
WITH_example_subquery.sql

Syntax errors can lead to pretty useless error messages.

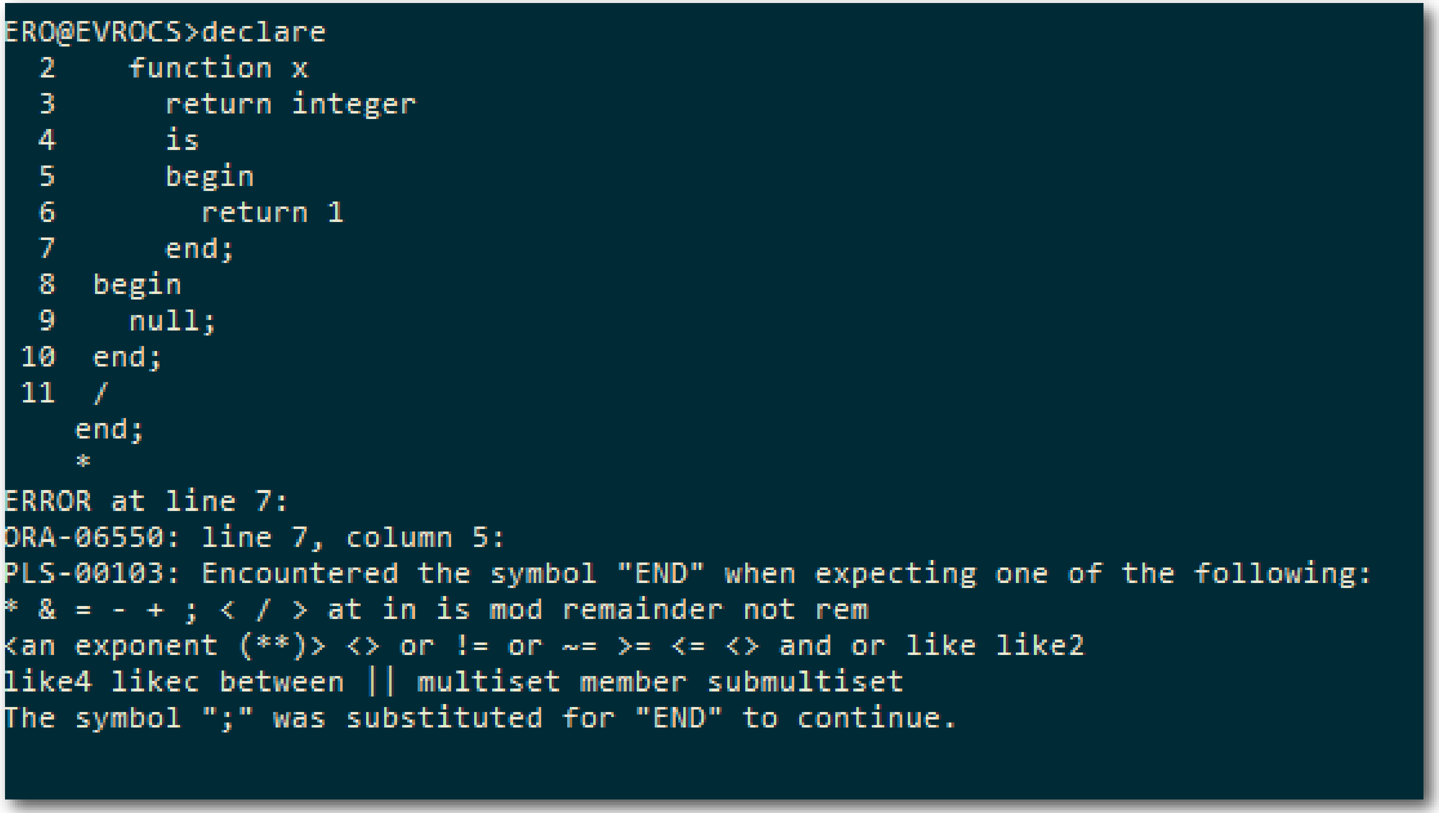
Often the message will just be:

`ERROR at line ###: ORA-00905: missing keyword`

where the line number is the line where the function begins

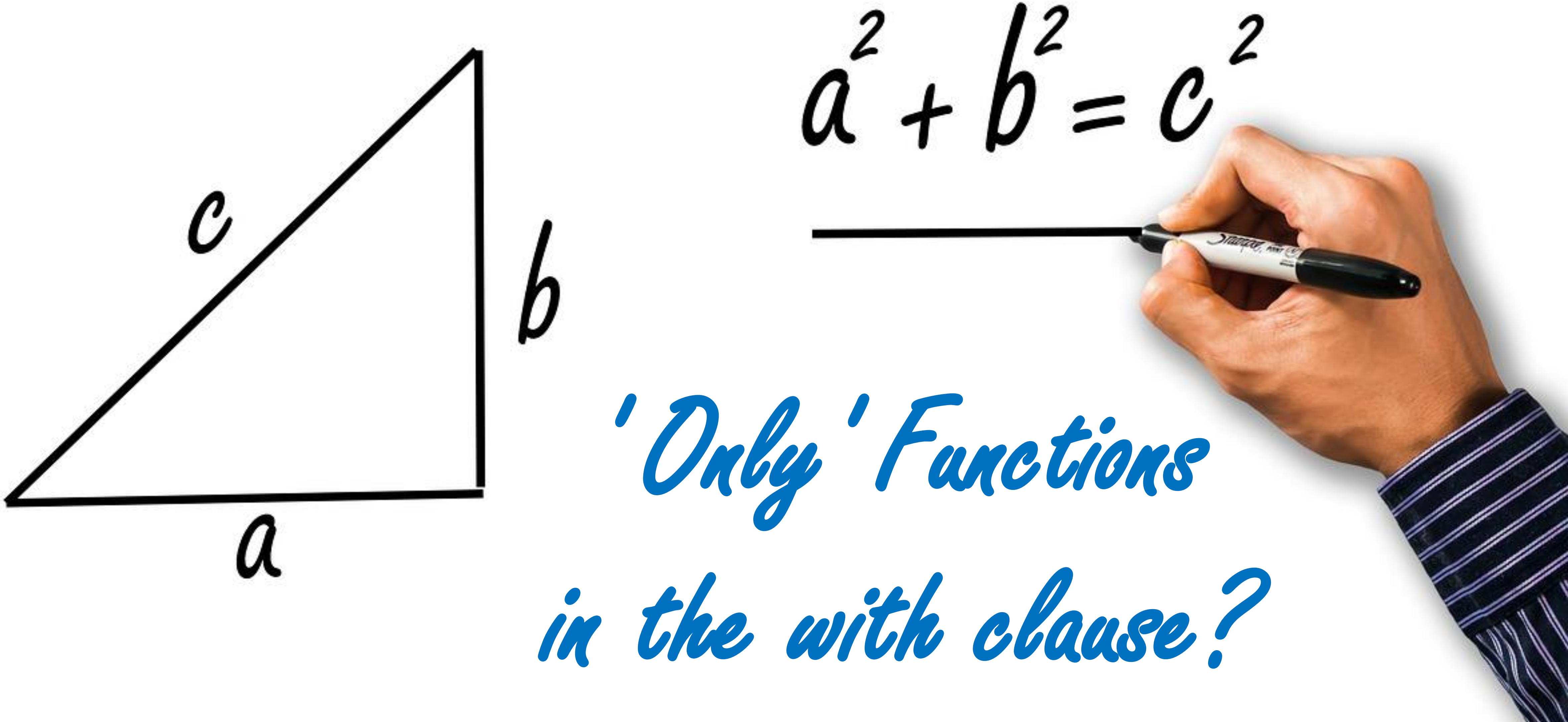


```
ERO@EVROCS>with
  2     function x
  3       return integer
  4       is
  5       begin
  6         return 1
  7       end;
  8   select x
  9   from   dual
 10   /
  function x
           *
ERROR at line 2:
ORA-00905: missing keyword
```



```
ERO@EVROCS>declare
  2     function x
  3       return integer
  4       is
  5       begin
  6         return 1
  7       end;
  8   begin
  9     null;
 10   end;
 11   /
    end;
    *
ERROR at line 7:
ORA-06550: line 7, column 5:
PLS-00103: Encountered the symbol "END" when expecting one of the following:
* & = - + ; < / > at in is mod remainder not rem
<an exponent (**)> <> or != or ~= >= <= <> and or like like2
like4 likec between || multiset member submultiset
The symbol ";" was substituted for "END" to continue.
```

Advise:
develop the PLSQL in an anonymous block until syntactically correct, before putting it into the with-clause

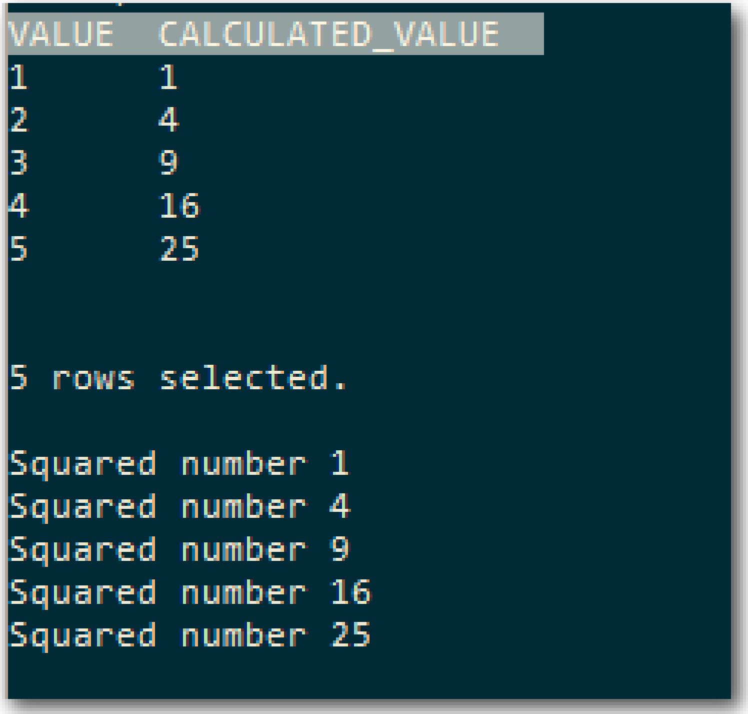


'Only' Functions in the with clause?

Procedures can also be defined in the With-clause!

But can obviously only be executed from other plsql in with clause, not the query.

```
with
  procedure proc1
    (pio_1 in out number)
    is
    begin
      pio_1 := power(pio_1 , 2);
      dbms_output.put_line ('Squared number '||to_char(pio_1));
    end proc1;
  --
  function func1
    (pi_1  in  integer)
    return number
    is
      l_value_1   number   := pi_1;
    begin
      proc1 (pio_1 => l_value_1);
      return (l_value_1);
    end func1;
select level                  value
,      func1 (pi_1 => level)  calculated_value
from   dual
connect by level <= 5
/
```



```
VALUE  CALCULATED_VALUE
1      1
2      4
3      9
4      16
5      25


5 rows selected.

Squared number 1
Squared number 4
Squared number 9
Squared number 16
Squared number 25
```

WITH_example_procedure.sql

How about speeding things up even more…..

Use Function Result Cache

No: Error message

Declare the function to be Deterministic

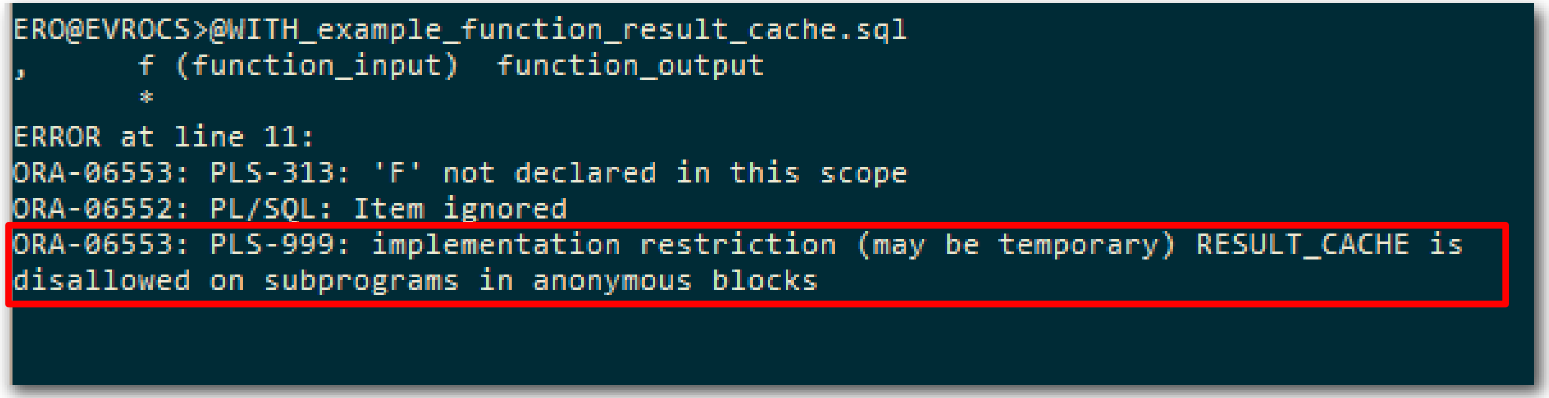
No: Ignored and unreliable

Use Scalar Subquery Caching

No: Ignored

Unfortunately, Function Result Cache can not be used

```
with
  function f (p in integer)
  return varchar2
  result_cache
  as
  begin
    return (p);
  end;
--
select function_input
,      f (function_input)     function_output
from   (select mod(level,2)   function_input
        from   dual
        connect by level <= 4
       )
/
```



```
ERO@EVROCS>@WITH_example_function_result_cache.sql
,      f (function_input)  function_output
       *
ERROR at line 11:
ORA-06553: PLS-313: 'F' not declared in this scope
ORA-06552: PL/SQL: Item ignored
ORA-06553: PLS-999: implementation restriction (may be temporary) RESULT_CACHE is
disallowed on subprograms in anonymous blocks
```

WITH_example_function_result_cache.sql

The function *can* be declared deterministic Without error messages, however .....

In **12.1**

- Deterministic is ignored

In **12.2 - 18**

- Sometimes it works
- Sometimes it's ignored
- Sometimes it gives wrong results due to bug

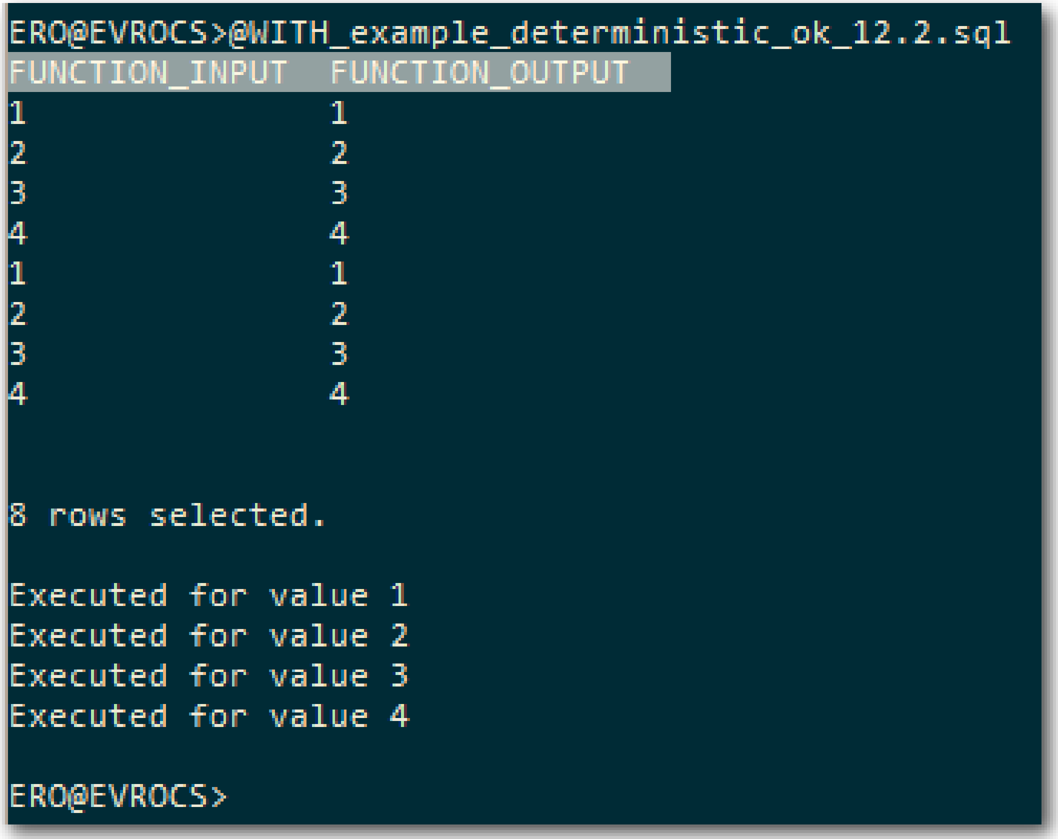
In **19**

- Deterministic is ignored
- Sometimes it gives wrong results due to bug

So, Best bet: Don’t use deterministic with this

Example of Deterministic **that works** in 12.2 – 18

```
with
  function f (p in integer)
  return integer
  deterministic
  as
  begin
    dbms_output.put_line ('Executed for value '||p);
    return (p);
  end;
--
select function_input
,      f (function_input)  function_output
from   (select level        function_input
        from   dual
        connect by level <= 4
        union all
        select level        function_input
        from   dual
        connect by level <= 4
       )
/
```

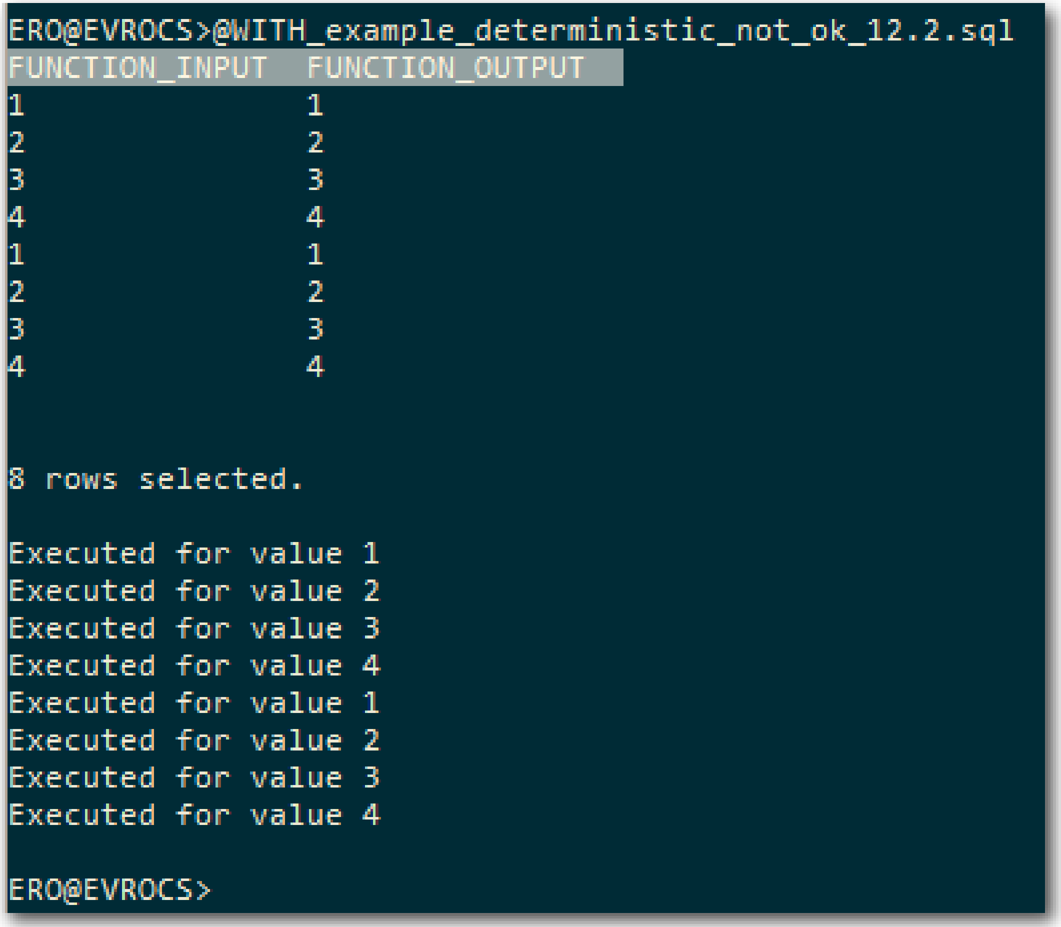


```
ERO@EVROCS>@WITH_example_deterministic_ok_12.2.sql
FUNCTION_INPUT  FUNCTION_OUTPUT
1               1
2               2
3               3
4               4
1               1
2               2
3               3
4               4


8 rows selected.

Executed for value 1
Executed for value 2
Executed for value 3
Executed for value 4

ERO@EVROCS>
```

WITH_example_deterministic_ok_12.2.sql

Example of Deterministic **that is ignored** in 12.2 and above

```
with
  function f (p in integer)
  return varchar2
  deterministic
  as
  begin
    dbms_output.put_line ('Executed for value '||p);
    return (p);
  end;
--
select function_input
,      f (function_input)  function_output
from   (select level       function_input
        from   dual
        connect by level <= 4
        union all
        select level       function_input
        from   dual
        connect by level <= 4
       )
/
```

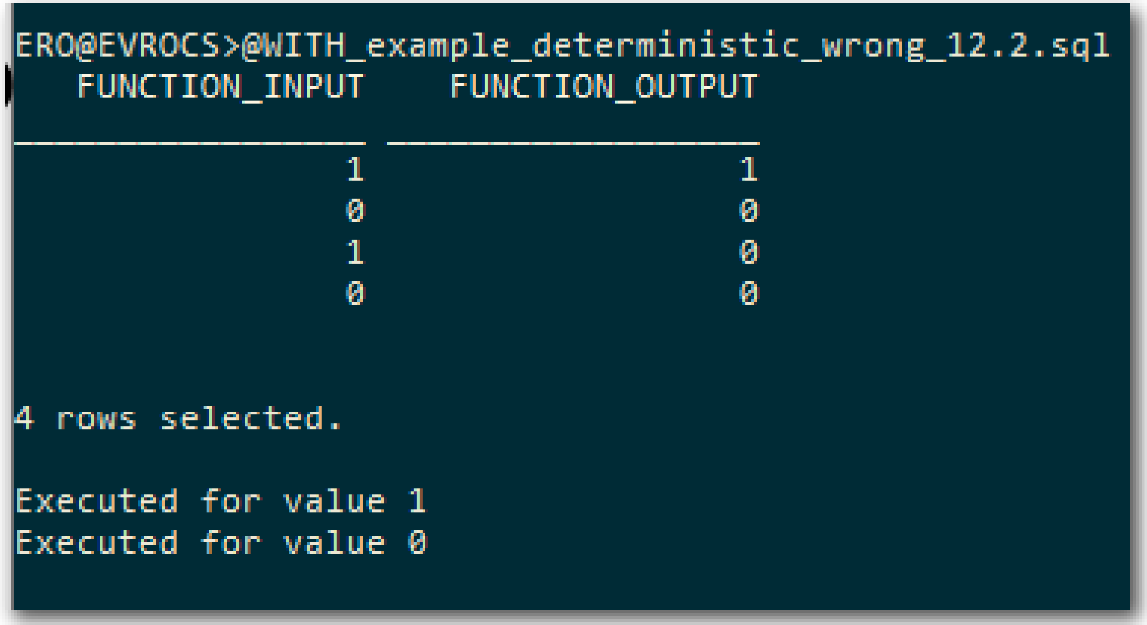


```
ERO@EVROCS>@WITH_example_deterministic_not_ok_12.2.sql
FUNCTION_INPUT  FUNCTION_OUTPUT
1               1
2               2
3               3
4               4
1               1
2               2
3               3
4               4


8 rows selected.

Executed for value 1
Executed for value 2
Executed for value 3
Executed for value 4
Executed for value 1
Executed for value 2
Executed for value 3
Executed for value 4

ERO@EVROCS>
```

WITH_example_deterministic_not_ok_12.2.sql

Example of ***wrong*** deterministic results in 12.2 and above

```
with
  function f (p in integer)
  return integer
  deterministic
  as
  begin
    dbms_output.put_line
      ('Executed for value '||p);
    return (p);
  end;
--
select mod(level,2)         function_input
,      f (mod(level,2))     function_output
from   dual
connect by level <= 4
/
```



```
ERO@EVROCS>@WITH_example_deterministic_wrong_12.2.sql
   FUNCTION_INPUT     FUNCTION_OUTPUT
_________________ __________________
                1                  1
                0                  0
                1                  0
                0                  0


4 rows selected.

Executed for value 1
Executed for value 0
```

Caused by bug: 27329690, WRONG RESULTS FROM INLINE DETERMINISTIC FUNCTION

Workaround: `alter session set "_plsql_cache_enable" = false`

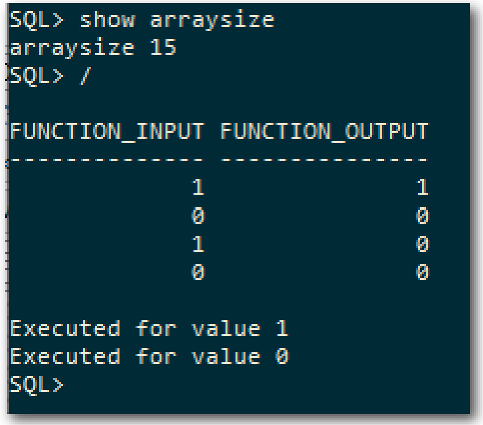
But turns ALL deterministic OFF!

WITH_example_deterministic_wrong_12.2.sql

Those wrong results of previous query…..

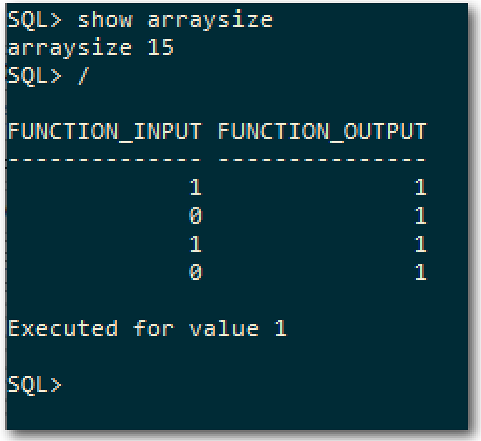
Behavior apparently depends on the client being used:

SQL*Plus
Default settings



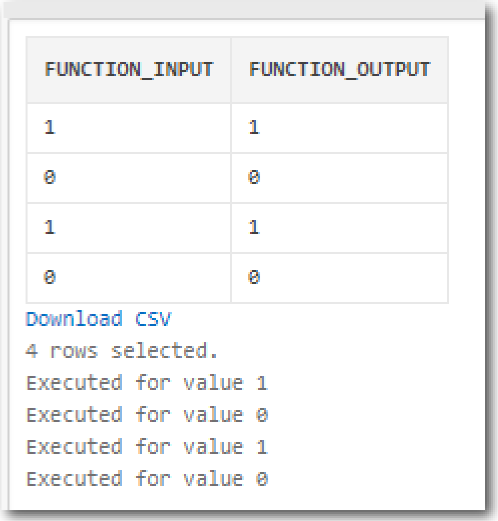
```
SQL> show arraysize
arraysize 15
SQL> /

FUNCTION_INPUT FUNCTION_OUTPUT
-------------- ---------------
             1               1
             0               0
             1               0
             0               0

Executed for value 1
Executed for value 0
SQL>
```

SQLcl
Default settings



```
SQL> show arraysize
arraysize 15
SQL> /

FUNCTION_INPUT FUNCTION_OUTPUT
-------------- ---------------
             1               1
             0               1
             1               1
             0               1

Executed for value 1

SQL>
```

LiveSQL (=From Apex)
https://livesql.oracle.com



FUNCTION_INPUT	FUNCTION_OUTPUT
1	1
0	0
1	1
0	0

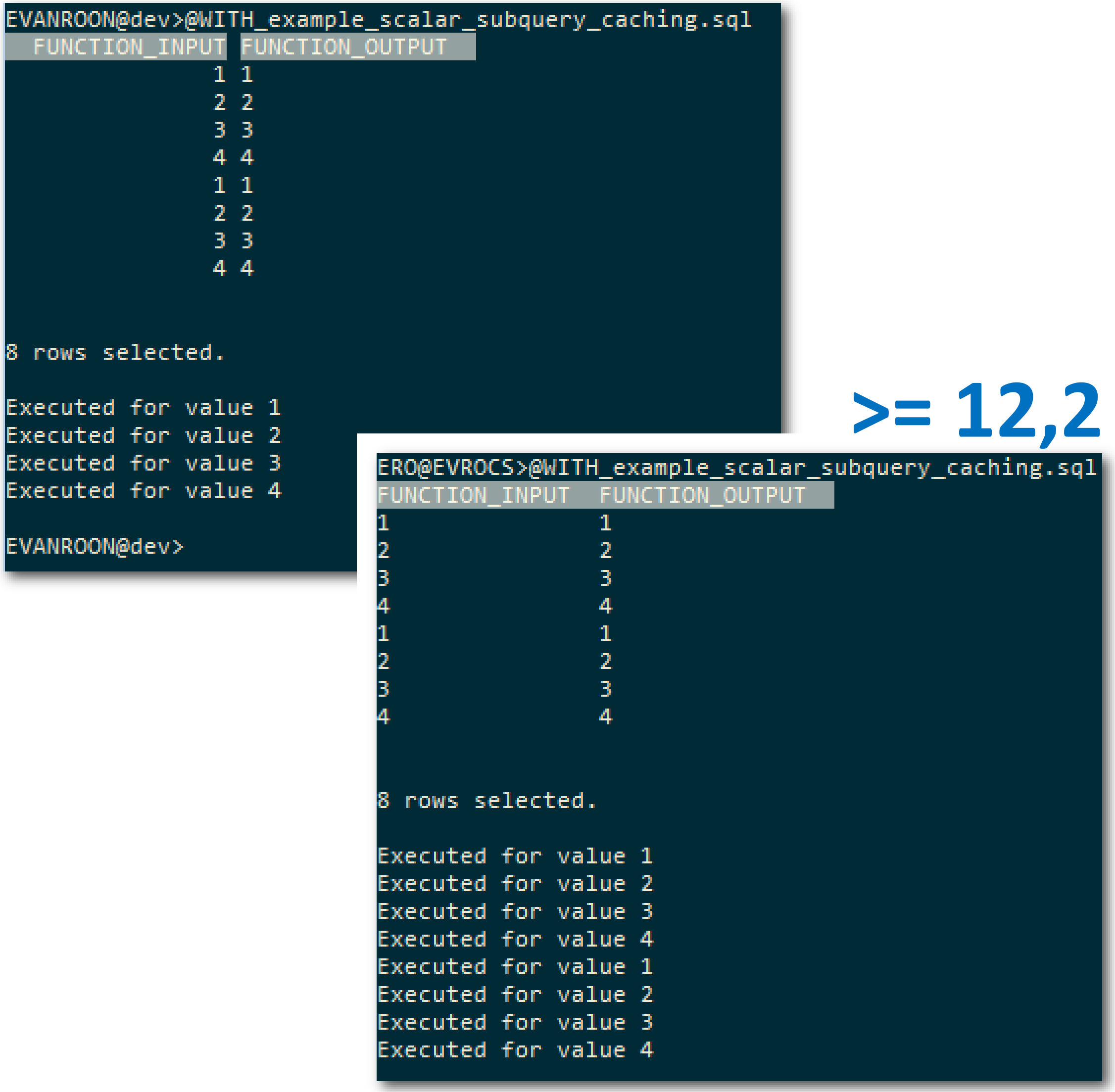
Download CSV

```
4 rows selected.
Executed for value 1
Executed for value 0
Executed for value 1
Executed for value 0
```

Scalar Subquery Caching *does* work in 12.1

Not in any later version ☹

```
with
  function f (p in integer)
  return varchar2
  as
  begin
    dbms_output.put_line ('Executed for value '||p);
    return (p);
  end;
select function_input
,      (select f (function_input)
        from dual
       )  function_output
from   (select level   function_input
        from   dual
        connect by level <= 4
        union all
        select level   function_input
        from   dual
        connect by level <= 4
       )
/
```



12,1

```
EVANROON@dev>@WITH_example_scalar_subquery_caching.sql
  FUNCTION_INPUT FUNCTION_OUTPUT
               1 1
               2 2
               3 3
               4 4
               1 1
               2 2
               3 3
               4 4


8 rows selected.

Executed for value 1
Executed for value 2
Executed for value 3
Executed for value 4

EVANROON@dev>
```

>= 12,2

```
ERO@EVROCS>@WITH_example_scalar_subquery_caching.sql
FUNCTION_INPUT  FUNCTION_OUTPUT
1               1
2               2
3               3
4               4
1               1
2               2
3               3
4               4


8 rows selected.

Executed for value 1
Executed for value 2
Executed for value 3
Executed for value 4
Executed for value 1
Executed for value 2
Executed for value 3
Executed for value 4
```

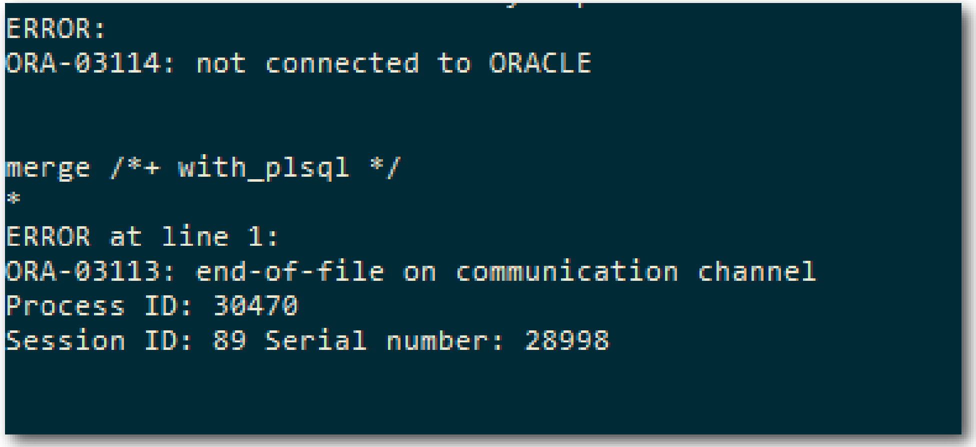
WITH_example_scalar_subquery_caching.sql

Bug 22654079, causing ora-600 in ***a very specific case***

```
merge /*+ with_plsql */
into  ero_test_merge_bug  tgt
using (with
         function f (p in integer)
           return integer
           is
           begin
             return p * 10;
           end;
         --
         select id
         ,      f (id)  value
         from   ero_test_merge_bug
         where  id = :my_id
      ) src
on    (tgt.id = src.id)
when matched
then
  update set tgt.value = src.value
when not matched
then
  insert values (src.id, src.value)
/
```

If:

1. You have a merge statement
2. AND plsql in the with-clause
3. AND the statement uses bind variables



```
ERROR:
ORA-03114: not connected to ORACLE


merge /*+ with_plsql */
*
ERROR at line 1:
ORA-03113: end-of-file on communication channel
Process ID: 30470
Session ID: 89 Serial number: 28998
```

WITH_example_merge_bug.sql

PLSQL in the with clause is not supported by PLSQL (**yet?**)

Declaring a cursor with such a query leads, once again to

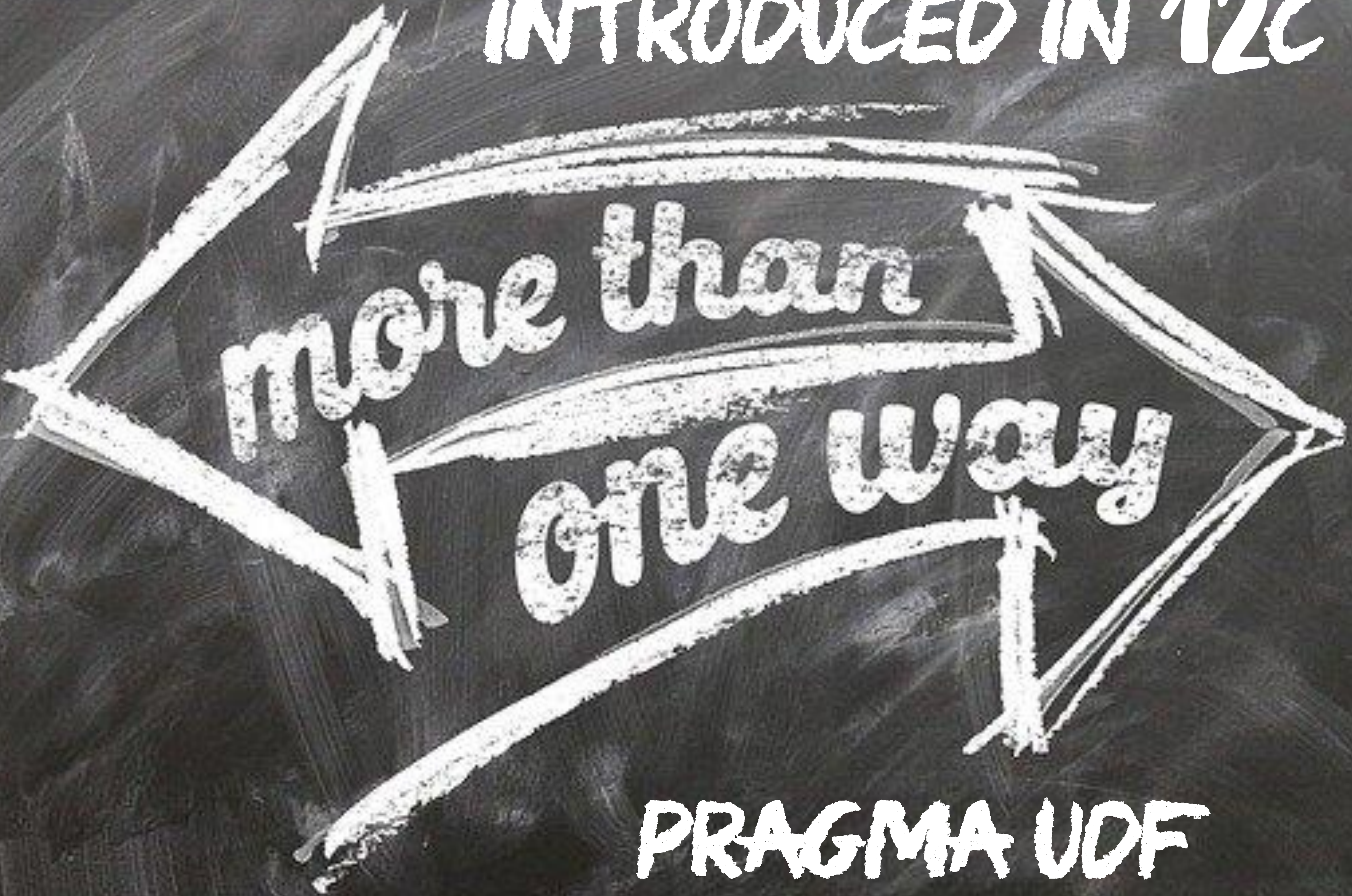
`"ORA-00905: missing keyword"`

Work-around: dynamic sql *does* work

But: ....

If already in plsql anyway,
why not declare the function separately with pragma udf?

INTRODUCED IN 12c



PRAGMA UDF

There's the possibility to add a

Pragma UDF

to any standalone or packaged function

Instructs the **compiler** that the function will **primarily** be used within **SQL**

Documentation says

"...which might improve its performance"

HOW
do we apply this?

```
create or replace function blabla
return ...
is
  pragma udf;
begin
  ....
  return ...
end;
/
```

And we're done!!

Advantages compared to with-clause functions

- ☑ **with_plsql** hint - not needed
- ☑ syntax errors – **'usual' messages**, instead of "missing keyword"
- ☑ **function result cache** UDF_example_result_cache.sql
- ☑ **Deterministic** UDF_example_deterministic.sql
  behavior like in with clause
  except: the wrong results query works correct
- ☑ **Scalar Subquery Caching**
- ☑ Use in **merge** statements with **bind** variables
- ☑ **Reuse of function**

Disadvantages compared to with-clause functions

- Need to create a database object (function/package)

  You may not want to
  (e.g. in a script in your toolkit for daily development/dba work)

  You may not be allowed to
  (e.g. in a script you run on different databases,
  among which Production)

- Performance gain (as we will soon see) is hard to predict

So, how fast is it?

Performance Test Scripts

1. Setup some objects

   Benchmark\initial_setup.sql

Table	ero_test_plsql_input	Contains test-data
Table	ero_test_plsql_tests	Contains test definitions and statistical data for test runs (avg, std. dev.)
Table	ero_test_plsql_results	Contains timing results for individual test executions
Package	ero_test_plsql_performance	Runs the actual tests

2. Run the tests

   Benchmark\run_tests.sql

   - Prompts for a number of rows in the test-data table
   - Runs all tests
   - Writes results to tables

3. Query the ero_test_plsql_tests table for minimum, maximum and average runtime of each test, including standard deviation and 95% confidence interval

4. Query the ero_test_plsql_results table for runtimes of individual executions of each test (which have led to the averages etc.)

5. Drop all objects of this test

   Benchmark\cleanup.sql

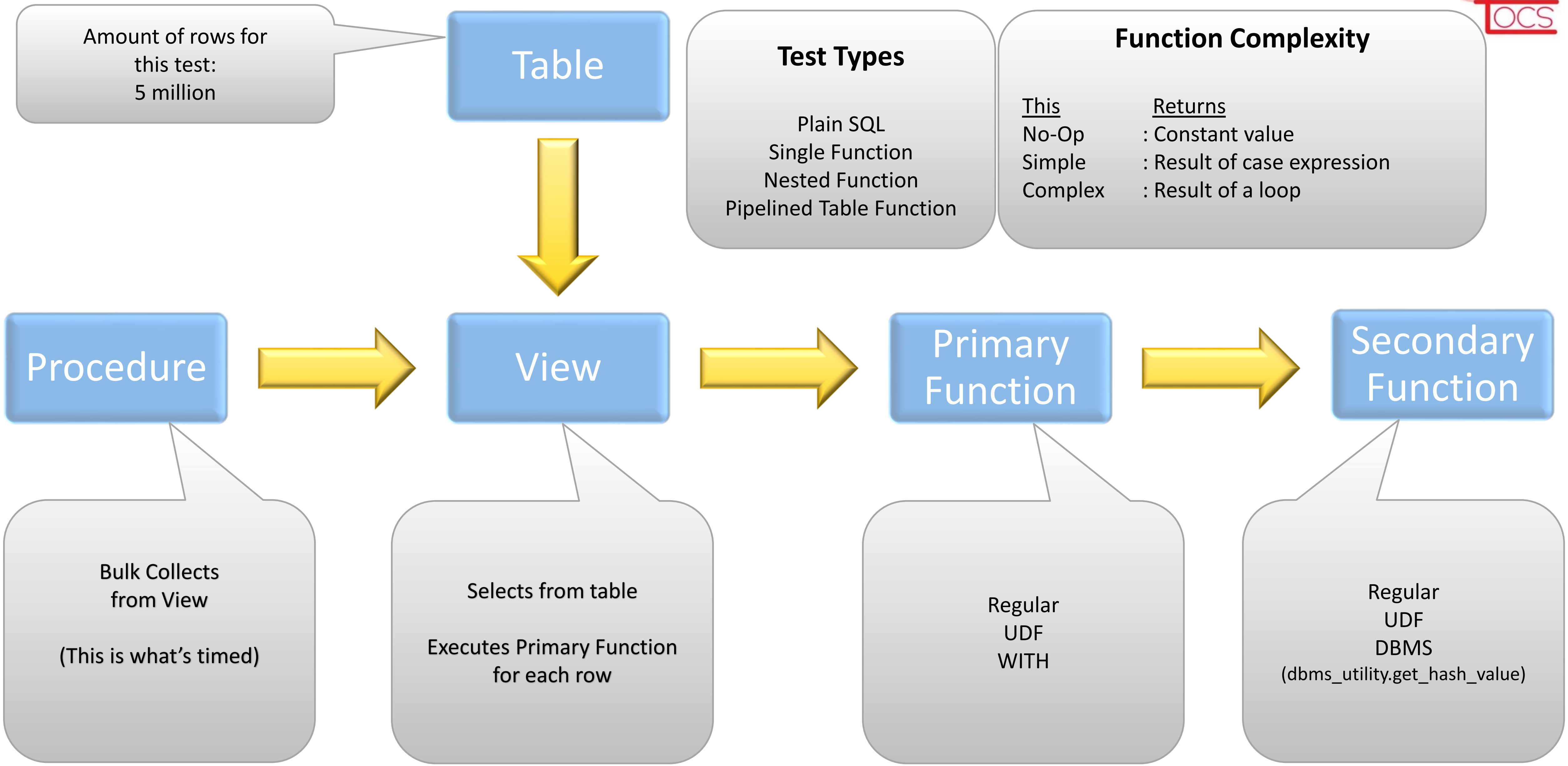
PGA - Warning

Running the scripts for these performance tests you will notice that the tests where the primary function is of type “**function in with clause**” cause a build up of pga used (no matter how simple the function is.
This **pga memory is not released** until the outermost plsql block is finished executing.
Even after the functions, view and procedure for a test are dropped, the pga stays in use. The next test adds to that.
The amount of memory allocated depends on the number of rows in the table (=number of times the function is executed) but it isn’t “a lot”.
In these testst it was about **10-15 MB per execution** of a test.

Running these tests with 5 million rows in the table the procedure crashed because it ran out of pga.
I had to increase max pga until **12 GB** to run the test.
But there are **66 tests** with “function in with clause” each executed **11 times** so **726 test runs** each executing **5 million functions**, totalling **3,6 billion function calls**. Usually that is not an everyday scenario.

Reproducable in versions 12.1 - 19. Not tested yet in 21.

Performance Tests



Amount of rows for
this test:
5 million

Table

Test Types

Plain SQL
Single Function
Nested Function
Pipelined Table Function

Function Complexity

This	Returns
No-Op	: Constant value
Simple	: Result of case expression
Complex	: Result of a loop

Procedure

View

Primary Function

Secondary Function

Bulk Collects
from View

(This is what's timed)

Selects from table

Executes Primary Function
for each row

Regular
UDF
WITH

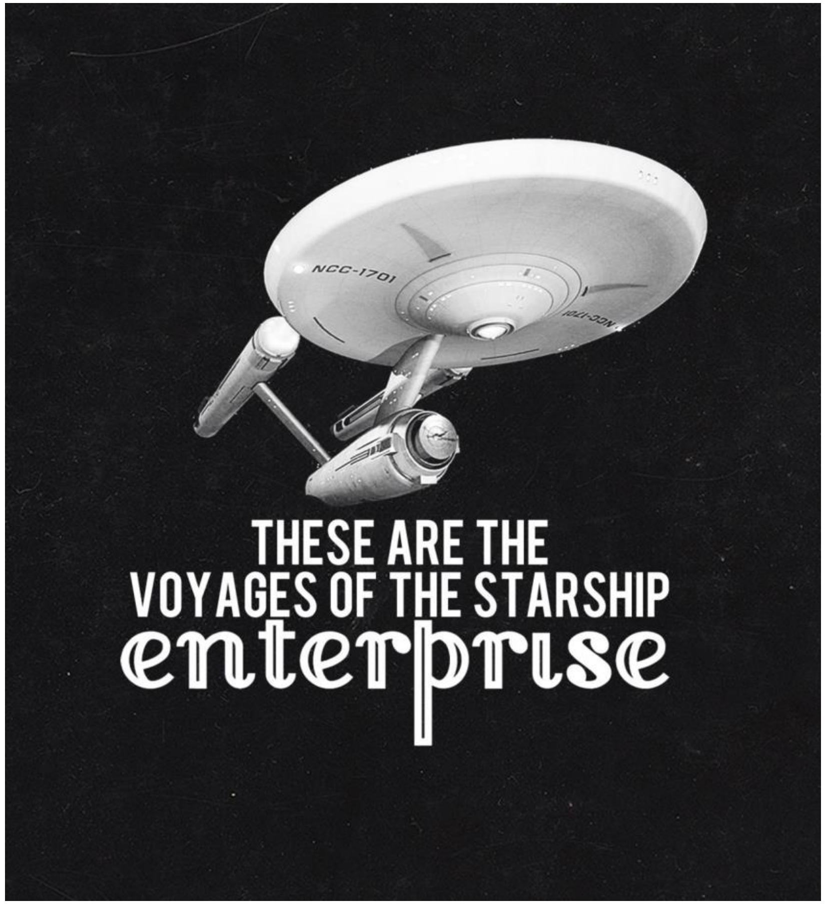
Regular
UDF
DBMS
(dbms_utility.get_hash_value)

Performance Test Execution

- Each test is run 11 times
  First run is ignored: warm-up run
- Average is calculated for the other 10
- Tests are executed for combinations of datatypes for parameter and return value
- All tests have been executed using plsql_optimize_level = 2
- A total of 211 distinct tests
- Baseline test is highlighted
  Is test with "regular function"
- In results runtimes are a percentage of the baseline test

Parameter Datatype	Return Datatype
NUMBER	NUMBER
NUMBER	VARCHAR2
NUMBER	DATE
VARCHAR2	NUMBER
VARCHAR2	VARCHAR2
VARCHAR2	DATE
DATE	NUMBER
DATE	VARCHAR2
DATE	DATE
BINARY_FLOAT	BINARY_FLOAT
BINARY_DOUBLY	BINARY_DOUBLY

Conclusions from **these** performance tests

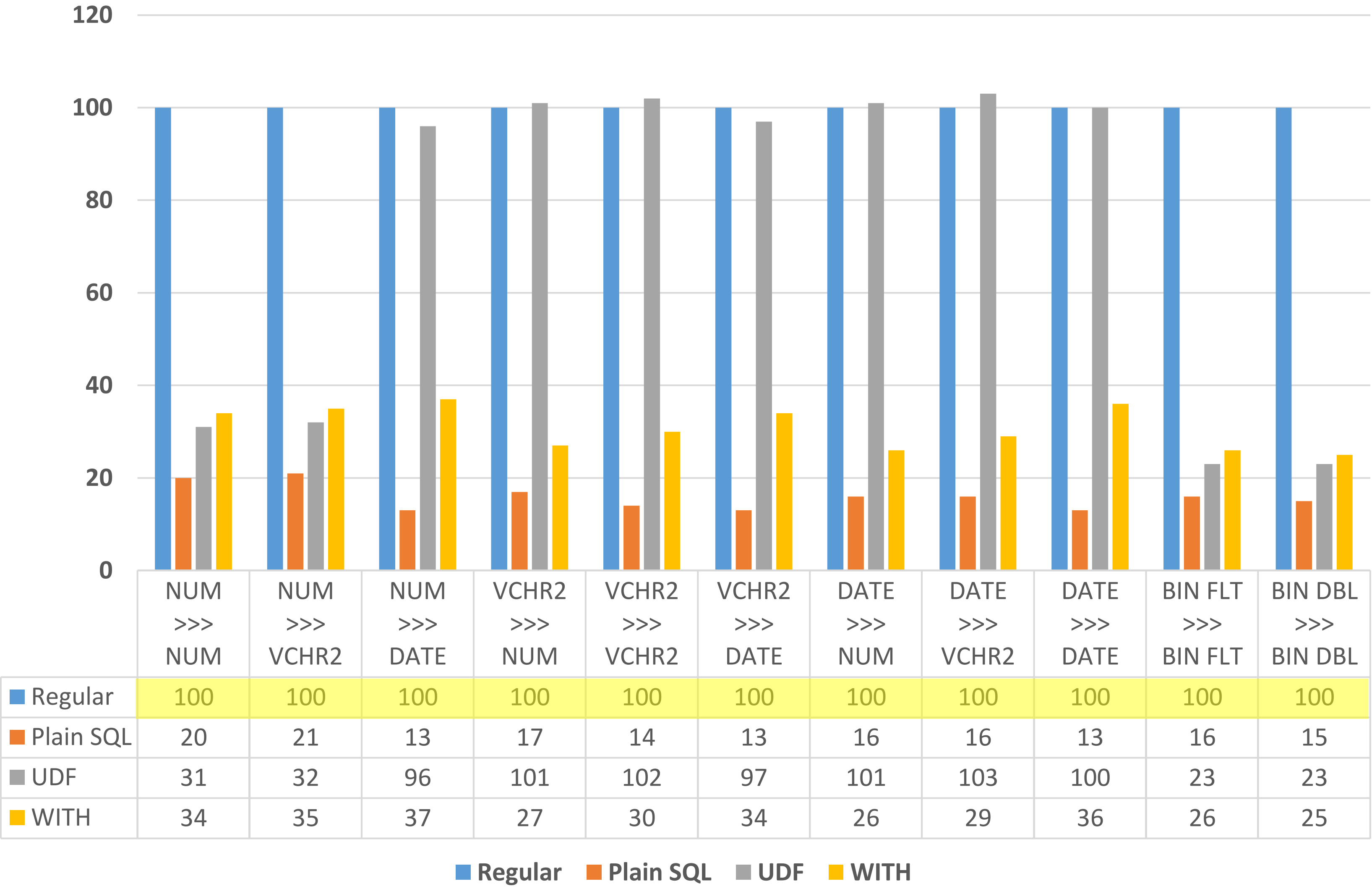


These are **NOT** the voyages of
The Millenium Falcon

These tests have **NOT** been run with **YOUR** real life functions / queries
These tests have **NOT** been run on **YOUR** hardware
These tests have **NOT** been run on **YOUR** version of **YOUR** database with **YOUR** patches

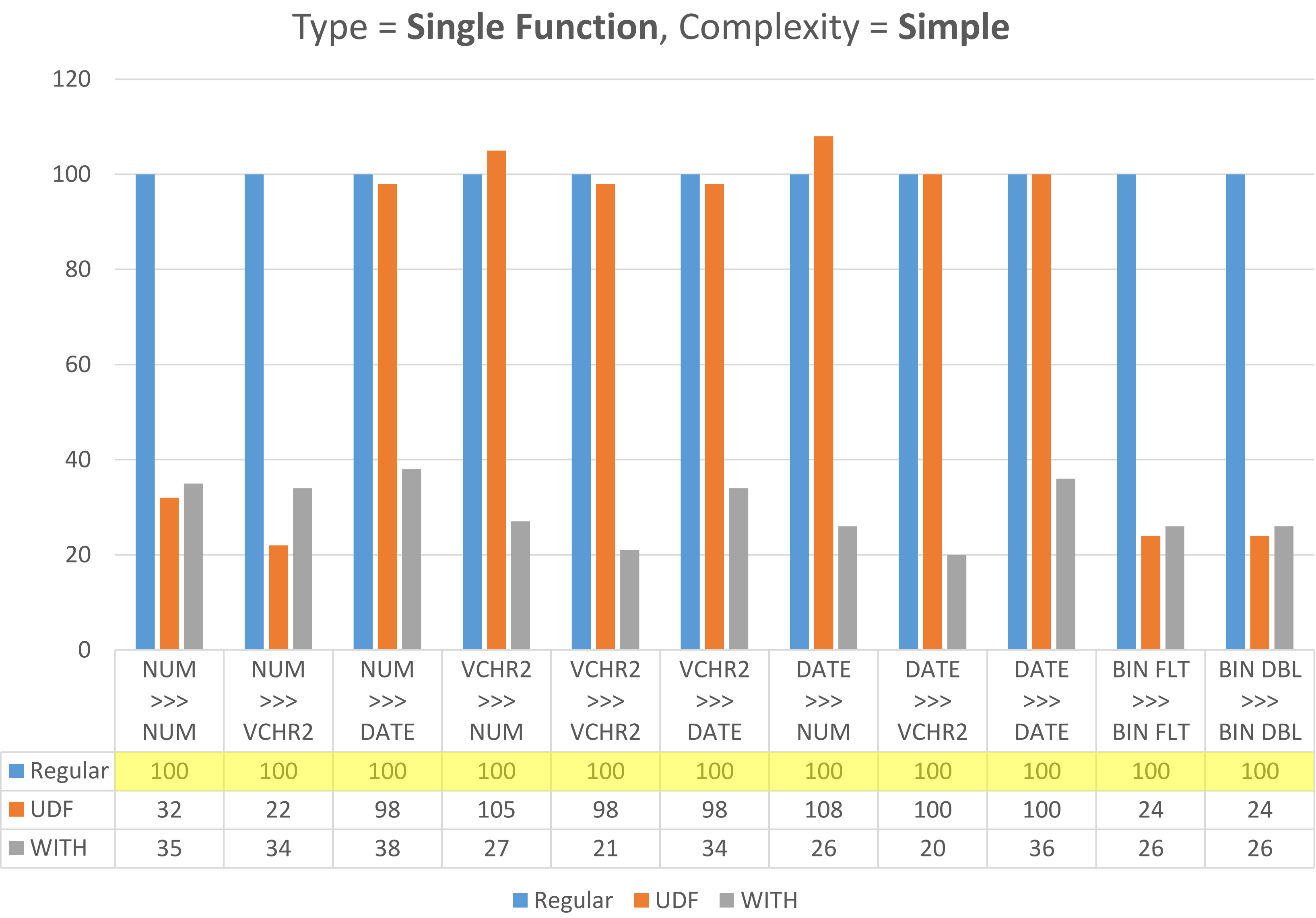
Tests of
Single Function

Type = **Single Function**, Complexity = **No-Op**



	NUM >>> NUM	NUM >>> VCHR2	NUM >>> DATE	VCHR2 >>> NUM	VCHR2 >>> VCHR2	VCHR2 >>> DATE	DATE >>> NUM	DATE >>> VCHR2	DATE >>> DATE	BIN FLT >>> BIN FLT	BIN DBL >>> BIN DBL
Regular	100	100	100	100	100	100	100	100	100	100	100
Plain SQL	20	21	13	17	14	13	16	16	13	16	15
UDF	31	32	96	101	102	97	101	103	100	23	23
WITH	34	35	37	27	30	34	26	29	36	26	25

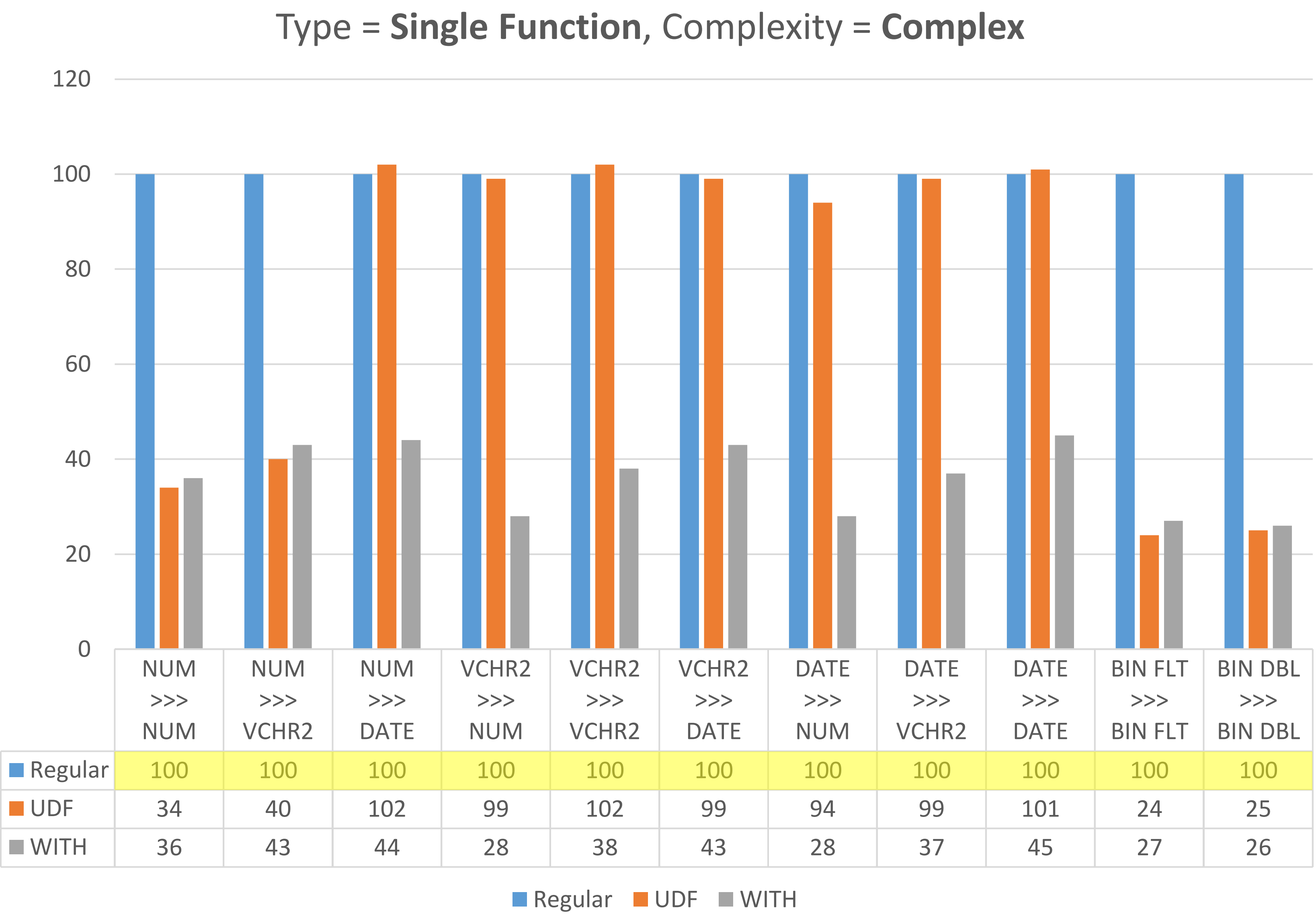
- No surprise:
  Plain SQL is always fastest
- Performance Gain depends on datatype of **parameter** and **returnvalue** especially for UDF functions
- Executing a UDF function is faster than a regular function
  for certain datatype combinations
- When a UDF Function is faster than a regular function it’s also slightly faster than a function in the With Clause
- Executing a function in the With Clause is faster than a regular function
  regardless of the datatypes


Type = **Single Function**, Complexity = **Simple**

	NUM >>> NUM	NUM >>> VCHR2	NUM >>> DATE	VCHR2 >>> NUM	VCHR2 >>> VCHR2	VCHR2 >>> DATE	DATE >>> NUM	DATE >>> VCHR2	DATE >>> DATE	BIN FLT >>> BIN FLT	BIN DBL >>> BIN DBL
Regular	100	100	100	100	100	100	100	100	100	100	100
UDF	32	22	98	105	98	98	108	100	100	24	24
WITH	35	34	38	27	21	34	26	20	36	26	26

- The numbers are **slightly** different than for a No-Op function

  But the conclusions are the same



Type = **Single Function**, Complexity = **Complex**

	NUM >>> NUM	NUM >>> VCHR2	NUM >>> DATE	VCHR2 >>> NUM	VCHR2 >>> VCHR2	VCHR2 >>> DATE	DATE >>> NUM	DATE >>> VCHR2	DATE >>> DATE	BIN FLT >>> BIN FLT	BIN DBL >>> BIN DBL
Regular	100	100	100	100	100	100	100	100	100	100	100
UDF	34	40	102	99	102	99	94	99	101	24	25
WITH	36	43	44	28	38	43	28	37	45	27	26

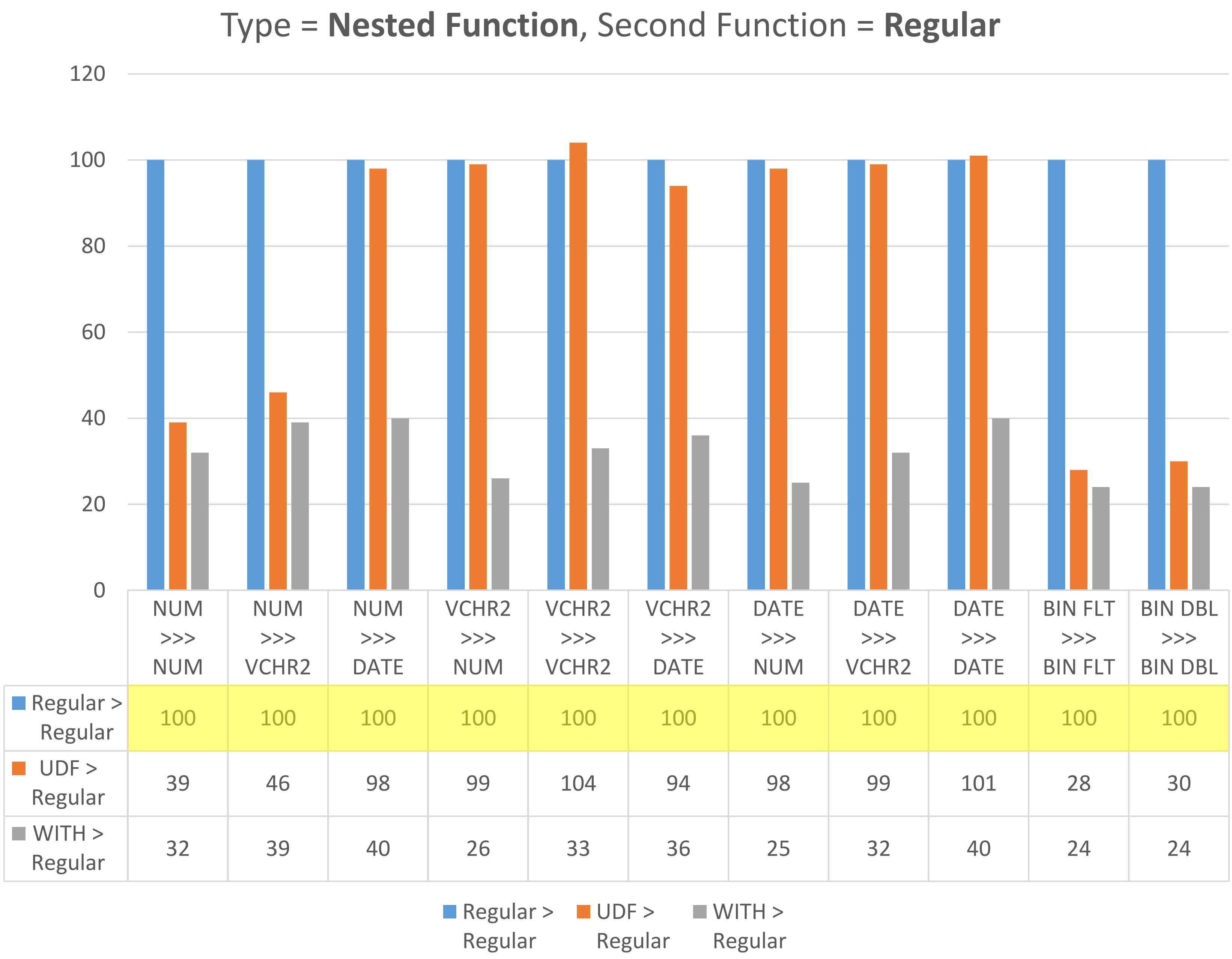
- The numbers are **slightly** different than for a No-Op function

  But the conclusions are the same

Tests of
Nested Functions

Complexity = “Complex”
for all these tests

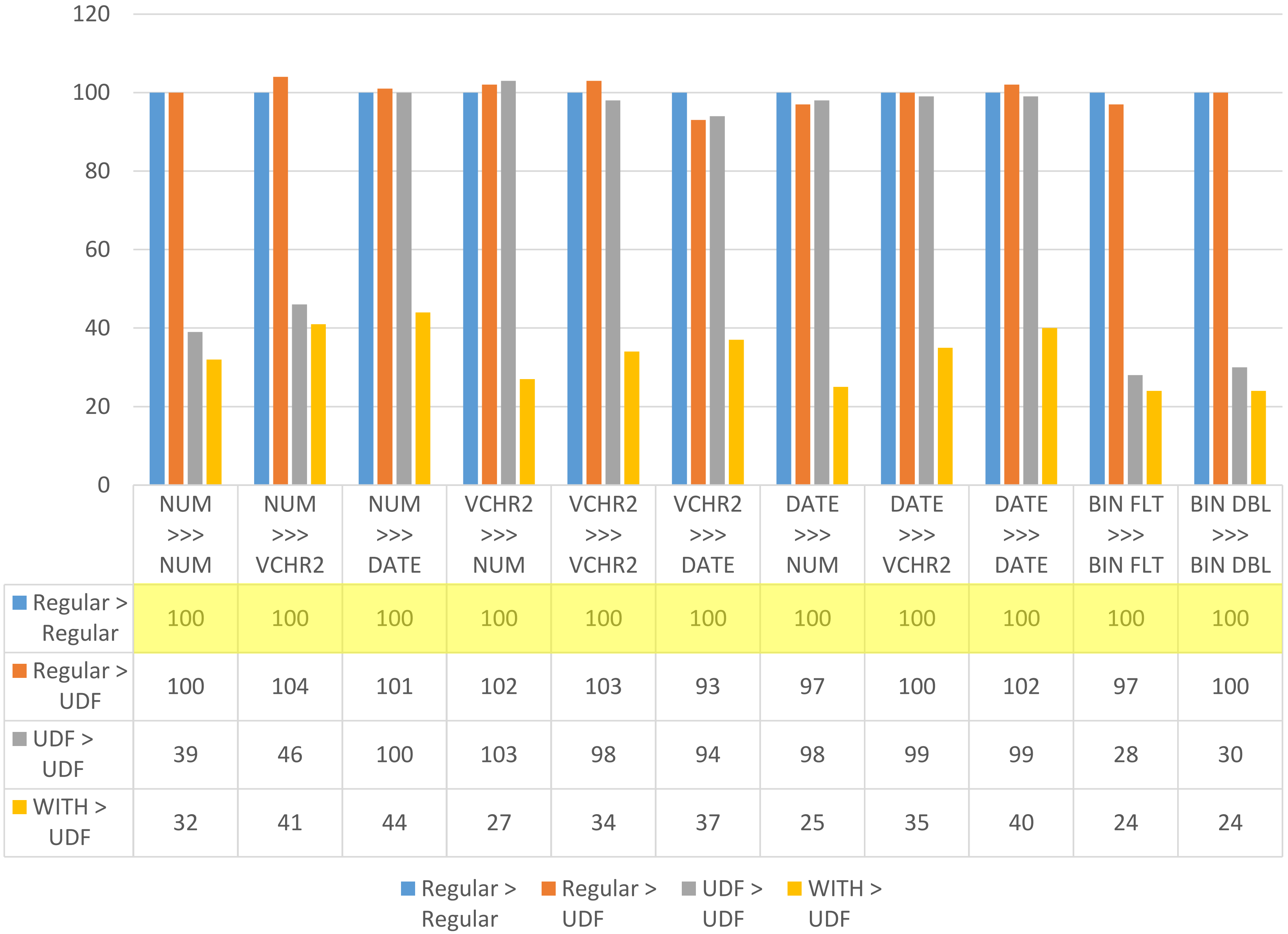
Type = **Nested Function**, Second Function = **Regular**



	NUM >>> NUM	NUM >>> VCHR2	NUM >>> DATE	VCHR2 >>> NUM	VCHR2 >>> VCHR2	VCHR2 >>> DATE	DATE >>> NUM	DATE >>> VCHR2	DATE >>> DATE	BIN FLT >>> BIN FLT	BIN DBL >>> BIN DBL
🟦 Regular > Regular	100	100	100	100	100	100	100	100	100	100	100
🟧 UDF > Regular	39	46	98	99	104	94	98	99	101	28	30
■ WITH > Regular	32	39	40	26	33	36	25	32	40	24	24

- When a regular function is executed by a UDF function or a function in the WITH clause, we see similar results to executing a single UDF or WITH clause function.

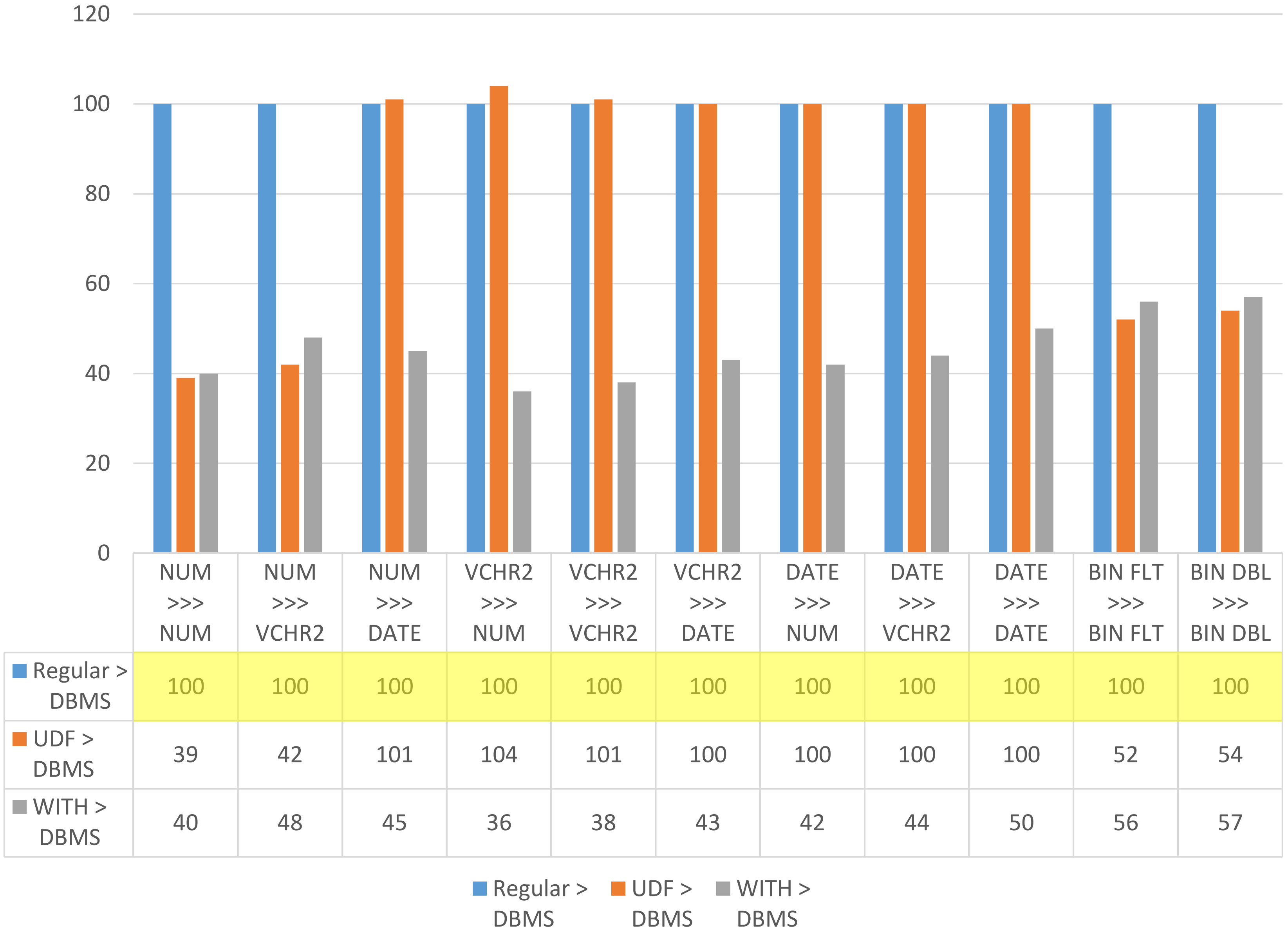
Type = **Nested Function**, Second Function = **UDF**



	NUM >>> NUM	NUM >>> VCHR2	NUM >>> DATE	VCHR2 >>> NUM	VCHR2 >>> VCHR2	VCHR2 >>> DATE	DATE >>> NUM	DATE >>> VCHR2	DATE >>> DATE	BIN FLT >>> BIN FLT	BIN DBL >>> BIN DBL
🟦 Regular > Regular	100	100	100	100	100	100	100	100	100	100	100
🟧 Regular > UDF	100	104	101	102	103	93	97	100	102	97	100
⬜ UDF > UDF	39	46	100	103	98	94	98	99	99	28	30
🟨 WITH > UDF	32	41	44	27	34	37	25	35	40	24	24

- If the primary function is a regular one, having it execute a UDF function may slow it down a bit
- When the primary function is a UDF or WITH clause function we see almost the exact same performance for a secondary UDF function as for a secondary regular function.
- In a chain of functions that call each other the optimization by the compiler depends solely on the type (regular, UDF, with clause) of the function that is executed by the query itself

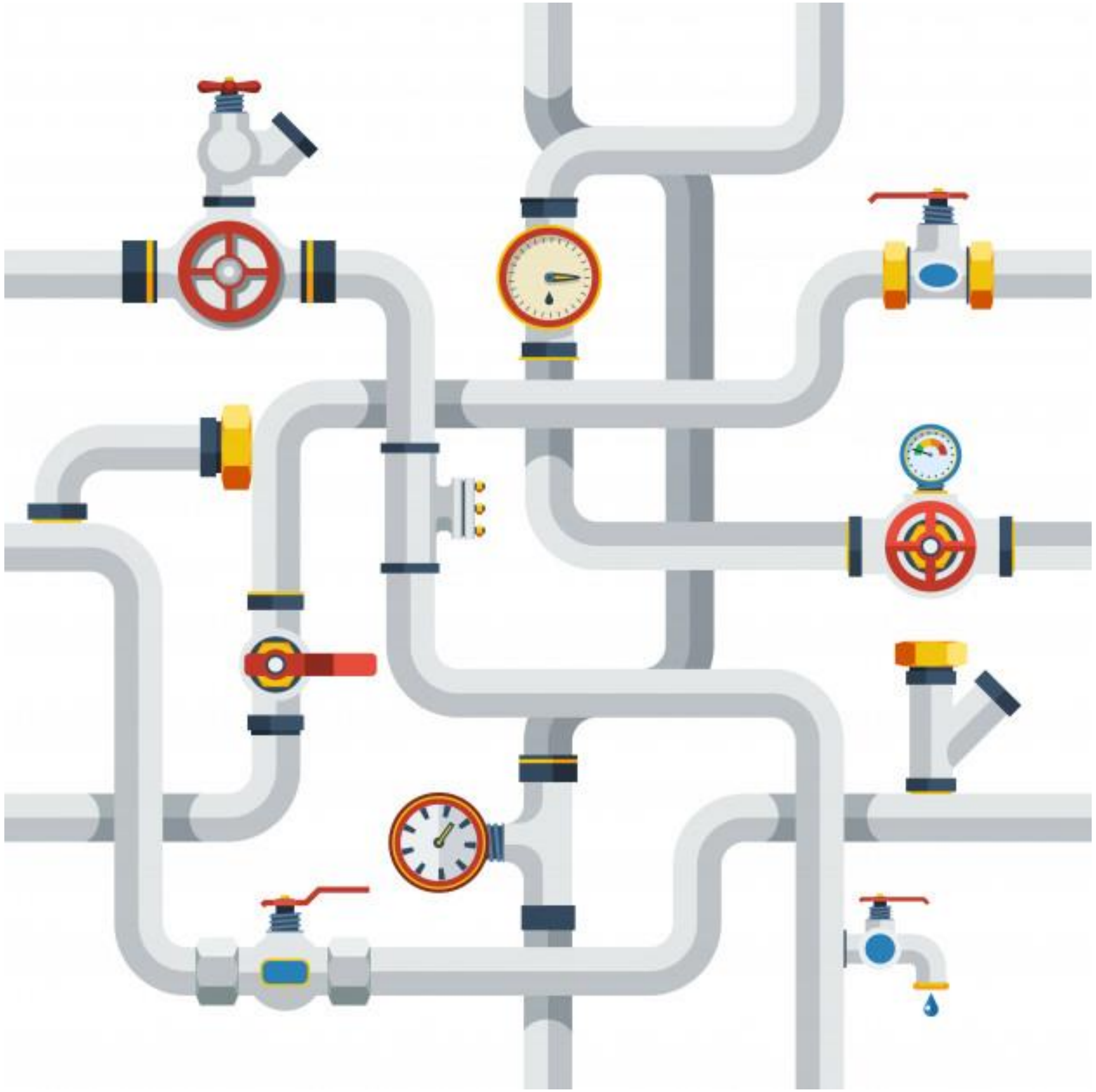
Type = **Nested Function**, Second Function = **DBMS**



	NUM >>> NUM	NUM >>> VCHR2	NUM >>> DATE	VCHR2 >>> NUM	VCHR2 >>> VCHR2	VCHR2 >>> DATE	DATE >>> NUM	DATE >>> VCHR2	DATE >>> DATE	BIN FLT >>> BIN FLT	BIN DBL >>> BIN DBL
🟦 Regular > DBMS	100	100	100	100	100	100	100	100	100	100	100
🟧 UDF > DBMS	39	42	101	104	101	100	100	100	100	52	54
⬜ WITH > DBMS	40	48	45	36	38	43	42	44	50	56	57

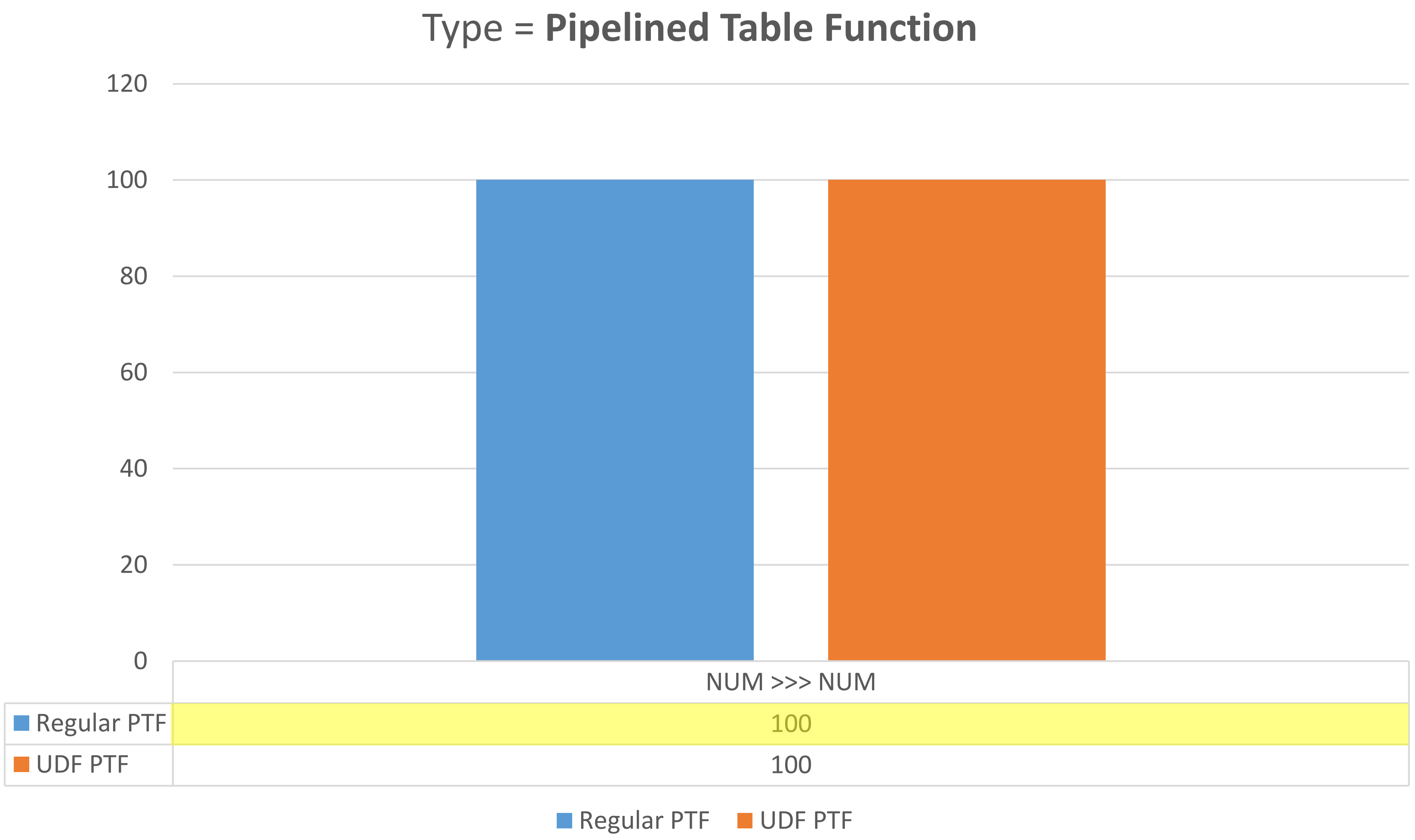
- Even when a function from one of Oracle’s supplied packages is executed from UDF or WITH clause function, we see almost the exact same performance gains as when the executed function is one of our own.

- We seem to be able to get a performance gain for functions we don’t control by having a wrapper function with pragma UDF or in the with clause.



Tests of
Pipelined Table Functions

Complexity = “Complex”
for these tests



- No performance gain at all by using pragma UDF in a pipelined table function

- Comment by Bryn Llewellyn:
  That is correct, because pipelined table functions have always been designed to be called from SQL and have hence always been optimized for use from SQL.

General Conclusions

- **Functions in the WITH clause**
  Not everything is possible, but it always (in these tests) gives a performance benefit.

- **Pragma UDF**
  ‘Everything’ is possible, but it doesn’t always lead to better performance.

- **Functions calling other functions**
  The type of the function that is called by the query determines the performance gain.

- **Wrap regular functions**
  The above seems to justify simply creating regular function in a package.
  So it can be reused everywhere, both in SQL and in PLSQL.
  If it needs to be used within SQL:
  Wrap it (execute it and return its result) in a function in the WITH clause.

  If however the query is in a PLSQL block,
  This can currently only be achieved with dynamic SQL.

“Stupid questions do exist.
But it takes a lot more time and energy to correct a stupid mistake than it takes to answer a stupid question, so please ask your stupid questions.”

a wise teacher who taught me more than just physics

Thanks !!