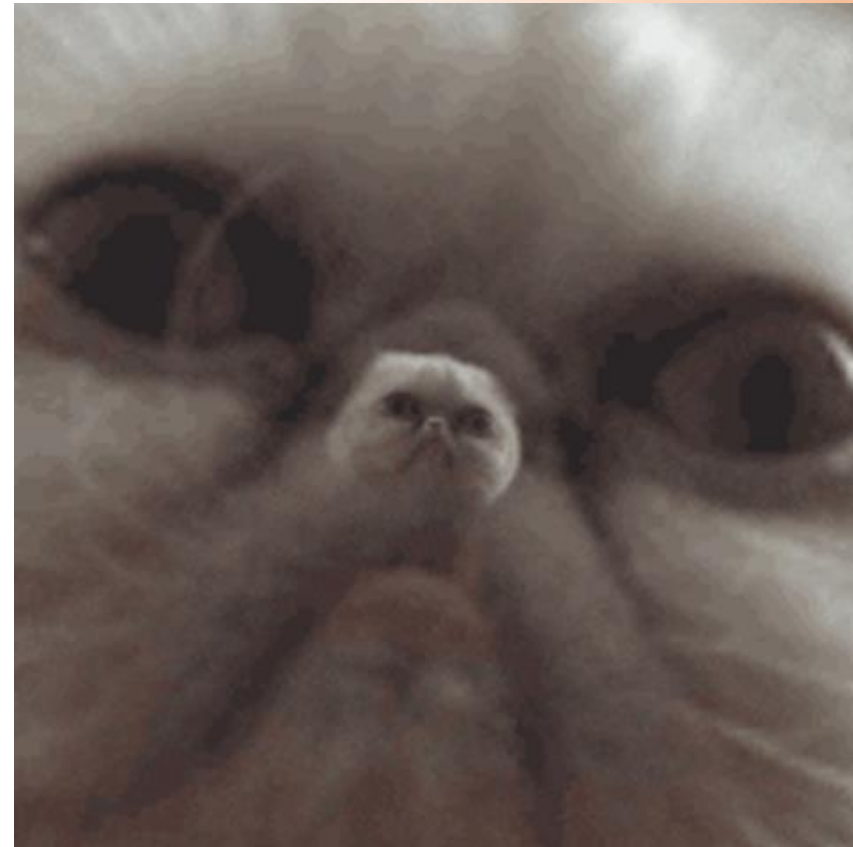


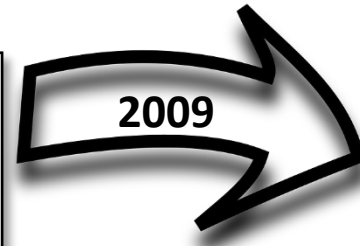
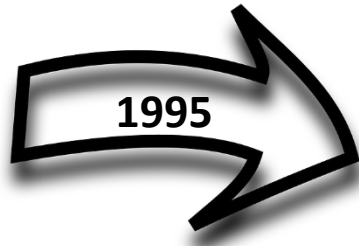
Zooming in on macros

Erik van Roon



Who Am I?

Erik van Roon



EvROCS
COMPLETING THE PUZZLE



Member of Symposium 42

<https://sym42.org/>



MASH Program



Oracle ACE
Pro



: erik.van.roon@evrocs.nl



: www.evrocs.nl



: @evrocs_nl



: @evrocs.bsky.social



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>

SYMPOSIUM 42

Created by the community, to support the community

Sharing of reliable knowledge

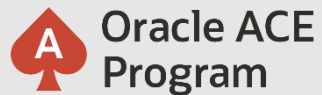
Supporting the various user groups and individuals



@sym_42



<https://sym42.org/>



500+ technical experts helping peers globally

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

3 membership tiers



For more details on Oracle ACE Program:
ace.oracle.com



Nominate
yourself or someone you know:

ace.oracle.com/nominate

Connect: aceprogram_ww@oracle.com

Facebook.com/OracleACEs

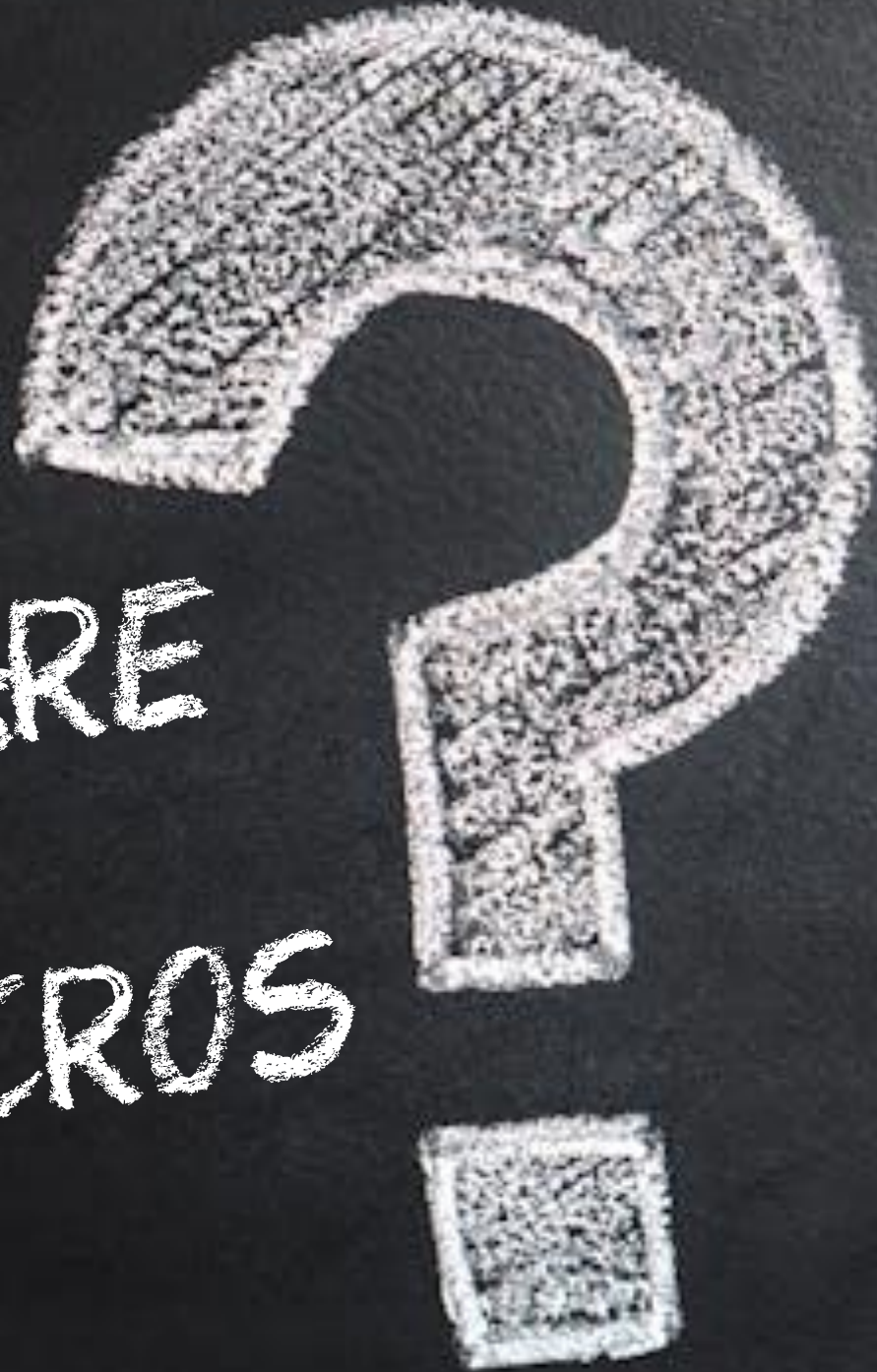
[@oracleace](https://twitter.com/oracleace)



Availability of SQL Macros

- Planned for 20c
=> wasn't released
- Implemented in 21c
=> Innovation release
- Fully available in 23ai
- **Partially** backported to 19c
(Only table macros)

WHAT ARE
SQL MACROS



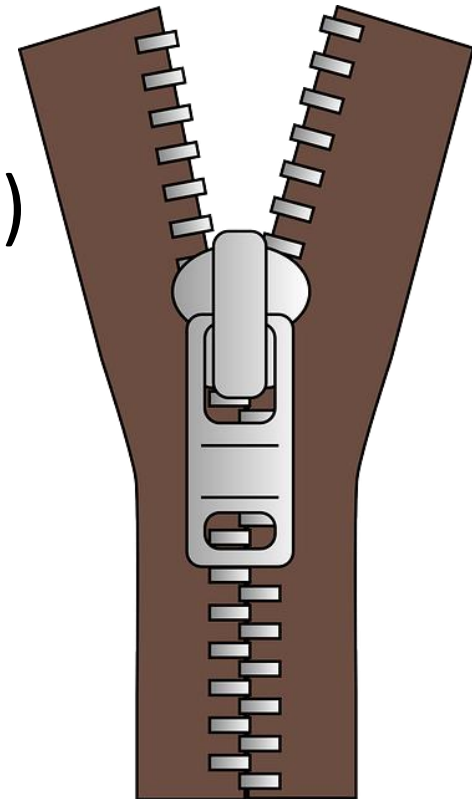
What are SQL Macros

Functions

That return a string

Which is a SQL Fragment
(an expression or a complete select statement)

That will be integrated into the query
(at hard-parse time)



Two types of SQL Macros



Table Macro

Returns a **query** or a **table**(name)
Can only be used in **from** clause

Can have parameters of
three 'special' datatypes:

- `dbms_tf.columns_t`
- `dbms_tf.columns_with_type_t`
- `dbms_tf.table_t`

Scalar Macro

Returns an **expression**
Can only be used **everywhere else**

Can have parameters of
two 'special' datatype:

- `dbms_tf.columns_t`
- `dbms_tf.columns_with_type_t`

Using `dbms_tf.table_t` leads to:

PLS-00777: scalar SQL macro cannot have argument of type DBMS_TF.TABLE_T

Very simple example of **scalar** Macro



```
create or replace function now_formatted
return varchar2 sql_macro (scalar)
is
begin
  return (q'[to_char(sysdate
                    , 'yyyy-mm-dd hh24:mi:ss'
                    )
          ]'
);
end;
```

Makes it a macro of type "scalar"

Is what will be merged into the query

```
select now_formatted() as Now
from dual
```

Becomes

```
select to_char(sysdate
               , 'yyyy-mm-dd hh24:mi:ss'
               ) as Now
from dual
```

```
MACRO@FREEPDB1>select now_formatted() as Now from dual;

NOW
-----
2024-04-21 16:09:18

1 row selected.
```

Very simple example of **table** Macro



```
create or replace function give_me_rows
return varchar2 sql_macro (table)
is
begin
    return (q'[select level as row_order
              from dual
              connect by level <= 5
              ]'
           );
end;
```

Makes it a macro of type "table"
(In 19c: just "sql_macro")

Is what will be merged
into the query

```
select row_order
from give_me_rows()
```

Becomes

```
select row_order
from (select level as row_order
      from dual
      connect by level <= 5
      )
```

```
MACRO@FREEPDB1>select row_order from give_me_rows();
```

```
ROW_ORDER
-----
1
2
3
4
5
```

```
5 rows selected.
```

Two things to notice.....

In previous example....

```
select row_order  
from   give_me_rows()
```

Reference to column that doesn't exist

But function is replaced at hard-parse

So column will be known before
select list is evaluated

Parenthesis are mandatory for **table**
macros, even if there are no
parameter values

```
MACRO@FREEPDB1>select row_order from give_me_rows;  
select row_order from give_me_rows  
                        *  
ERROR at line 1:  
ORA-04044: procedure, function, package, or type is not allowed here
```

So.....? Parameters? Of course.



```
create or replace function select_parameters
(p_1 in varchar2
,p_2 in varchar2 default 'X'
,p_3 in varchar2 default 'X'
)
return varchar2 sql_macro (table)
is
begin
  dbms_output.put_line ('p_1: '||p_1);
  dbms_output.put_line ('p_2: '||p_2);
  dbms_output.put_line ('p_3: '||p_3);

  return ('select '''||p_1||''' as p_1
          ,      '''||p_2||''' as p_2
          ,      '''||p_3||''' as p_3
          from dual'
          );
end;
```

Let's try them like we're used to

```
select *
from   select_parameters
      (p_1 => 'X'
      ,p_2 => 'X')
```

```
MACRO@FREEPDB1>select *
 2  from   select_parameters
 3         (p_1 => 'X'
 4*        ,p_2 => 'X');
```

```
P_1    P_2    P_3
```

```
-----
                X
```

```
1 row selected.
```

```
p_1:
p_2:
p_3: X
```

Parameter values are NOT accessible inside the Macro
(Except for numeric ones that receive a literal)

Parameters, the correct use



```
create or replace function select_parameters
(p_1 in varchar2
,p_2 in varchar2 default 'X'
,p_3 in varchar2 default 'X'
)
return varchar2 sql_macro (table)
is
begin
  dbms_output.put_line ('p_1: '||p_1);
  dbms_output.put_line ('p_2: '||p_2);
  dbms_output.put_line ('p_3: '||p_3);

  return ('select p_1 as p_1
          ,      p_2 as p_2
          ,      p_3 as p_3
          from dual'
);
end;
```

Parameter name inside the string

```
select *
from   select_parameters
      (p_1 => 'X'
      ,p_2 => 'X')
```

```
MACRO@FREEPDB1>select *
 2  from   select_parameters
 3         (p_1 => 'X'
 4*        ,p_2 => 'X');

P_1      P_2      P_3
-----
X         X         X

1 row selected.

p_1:
p_2:
p_3: X
```

The name is a placeholder, substituted at hard-parse

Local variables



```
create or replace function local_variables
return varchar2 sql_macro (table)
is
  cn_a constant varchar2(15) := 'Constant value';
  l_b   varchar2(15) := 'Variable value';
begin
  return ('select '''||cn_a||''' as const
         ,      '''||l_b ||''' as var
         from dual'
         );
end;
```

Variables and Constants that are local to the macro do get concatenated and are accessible

```
select *
from local_variables()
```

```
MACRO@FREEPDB1>select *
2 from local_variables()
3* ;
```

CONST	VAR
-----	-----
Constant value	Variable value

```
1 row selected.
```

Can we do some SQL injection?



```
create or replace function sql_injection
(p_1 in varchar2
)
return varchar2 sql_macro (table)
is
begin
    return ('select p_1 as p_1
           from dual'
);
end;
```

The parameter values get quoted,
So start and end our malicious string
with a literal quote to create empty
strings in combination with the
quoting of the parameter

```
select *
from sql_injection
('' || default_tablespace from user_users union all select dummy || ''')
```

```
MACRO@FREEPDB1>select *
 2 from sql_injection('' || default_tablespace from user_users union all select dummy || ''')
 3* ;

P_1
-----
' || default_tablespace from user_users union all select dummy || '

1 row selected.
```

So....
NO!!

(For now.... 😊)

How about scalar macro with same name as column?

```
create or replace function min_salary
return varchar2 sql_macro (scalar)
is
  l_stmt varchar2(32767);
begin
  l_stmt := '5000000';

  return l_stmt;
end;
```

```
select job_title
,      min_salary
from   jobs
where  job_id = 'IT_PROG'
```

```
MACRO@FREEPDB1>select job_title
2 ,      min_salary
3 from   jobs
4 where  job_id = 'IT_PROG'
5* ;
```

JOB_TITLE	MIN_SALARY
Programmer	4000

Unfortunately (😊): Nope

What you can do:

- Fully qualify the macro
- Add parenthesis

```
select job_title
,      min_salary           as col_name
,      macro.min_salary     as fully_qual
,      min_salary()         as parenthesis
from   jobs
where  job_id = 'IT_PROG'
;
```

JOB_TITLE	COL_NAME	FULLY_QUAL	PARENTHESIS
Programmer	4000	5000000	5000000

Special datatypes for parameters



Parameters of type

- `dbms_tf.table_t`
to pass in a table to the macro
ONLY to a table macro!!
- `dbms_tf.columns_t`
to pass in a (list of) columns
- `dbms_tf.columns_with_type_t`
to pass in a (list of) columns
including their datatype

dbms_tf.table_t



```
create or replace function couple_of_rows
(p_table in dbms_tf.table_t
,p_rows in integer
)
return varchar2 sql_macro (table)
is
begin
    return ('select *
            from p_table
            fetch first p_rows rows only
            ');
end;
```

```
select *
from couple_of_rows(departments, 3)
```

```
MACRO@FREEPDB1>select *
2 from couple_of_rows(departments, 3)
3* ;
```

DEPARTMENT_ID	DEPARTMENT_NAME	MANAGER_ID	LOCATION_ID
10	Administration	200	1700
20	Marketing	201	1800
30	Purchasing	114	1700

```
select *
from couple_of_rows(locations, 3)
```

```
MACRO@FREEPDB1>select *
2 from couple_of_rows(locations, 3)
3* ;
```

LOCATION_ID	STREET_ADDRESS	POSTAL_CODE	CITY
1000	1297 Via Cola di Rie	00989	Roma
1100	93091 Calle della Testa	10934	Venice
1200	2017 Shinjuku-ku	1689	Tokyo

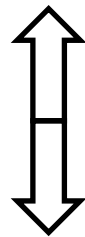
Things to notice...

```
MACRO@FREEPDB1>select *
 2 from couple_of_rows(departments, 3)
 3* ;
```

DEPARTMENT_ID	DEPARTMENT_NAME	MANAGER_ID	LOCATION_ID
10	Administration	200	1700
20	Marketing	201	1800
30	Purchasing	114	1700

3 rows selected.

Pass in a table **identifier**
NOT a quoted table name



Even the returned structure may be completely different, depending on the input

```
MACRO@FREEPDB1>select *
 2 from couple_of_rows(locations, 3)
 3* ;
```

LOCATION_ID	STREET_ADDRESS	POSTAL_CODE	CITY	STATE_PROVINCE	COUNTRY_ID
1000	1297 Via Cola di Rie	00989	Roma		IT
1100	93091 Calle della Testa	10934	Venice		IT
1200	2017 Shinjuku-ku	1689	Tokyo	Tokyo Prefecture	JP

3 rows selected.

Input

Input for a `dbms_tf.table_t` parameter can be
table
View
CTE

Input can not be
Pipelined table function

```
MACRO@FREEPDB1>select * from use_my_ptf(my_ptf);
select * from use_my_ptf(my_ptf)
                *
ERROR at line 1:
ORA-64630: unsupported use of SQL macro: use of table function as argument is not supported
```

Structure of table_t



```
TYPE TABLE_T IS RECORD
```

```
(column  
, schema_name  
, package_name  
, ptf_name  
, table_schema_name  
, table_name  
);
```

```
TABLE_COLUMNS_T
```

```
dbms_quoted_id  
dbms_quoted_id  
dbms_quoted_id  
dbms_quoted_id  
dbms_quoted_id
```

```
TYPE TABLE_COLUMNS_T
```

```
IS TABLE OF COLUMN_T;
```

```
TYPE COLUMN_T IS RECORD
```

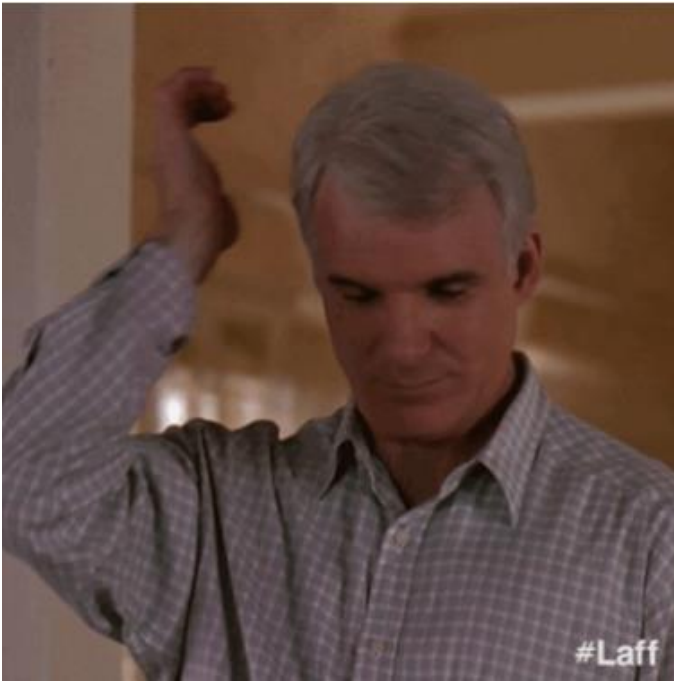
```
(description COLUMN_METADATA_T  
, pass_through BOOLEAN  
, for_read BOOLEAN  
);
```

```
TYPE COLUMN_METADATA_T IS RECORD
```

```
(type PLS_INTEGER DEFAULT TYPE_VARCH  
, max_len PLS_integer DEFAULT -1  
, name VARCHAR2(32767)  
, name_len PLS_INTEGER  
/* used for numerical data */  
, precision PLS_INTEGER  
, scale PLS_INTEGER  
/* used for character data */  
, charsetid PLS_INTEGER  
, charsetform PLS_INTEGER  
, collation PLS_INTEGER  
/* may be used in future */  
, schema_name DBMS_QUOTED_ID  
, schema_name_len PLS_INTEGER  
, type_name DBMS_QUOTED_ID  
, type_name_len PLS_INTEGER  
);
```

But???

But I only supply a table identifier???
Not this structure???



It's magic!

The structure is automatically populated based on the table identifier

And for the query the table name is automatically extracted from this

Do not expect this to work with regular functions/procedures

```
create or replace procedure regular_proc
(p_table in dbms_tf.table_t
)
is
begin
    null;
end;
```

```
begin
    regular_proc (departments);
end;
```

```
regular_proc (departments);
*
ERROR at line 2:
ORA-06550: line 2, column 17:
PLS-00357: Table,View Or Sequence reference 'DEPARTMENTS' not allowed in
```

But you CAN use the entire structure inside the Macro



```
create or replace function no_numeric_cols
(p_table in dbms_tf.table_t)
return varchar2 sql_macro (table)
is
  l_column_rec dbms_tf.column_metadata_t;
  l_columns varchar2(1000);
begin
  for i_col in 1 .. p_table.column.count
  loop
    l_column_rec := p_table.column(i_col).description;
    if dbms_tf.column_type_name (l_column_rec) <> 'NUMBER'
    then
      l_columns := l_columns || case
        when l_columns is not null
        then ','
        end || l_column_rec.name;
    end if;
  end loop;

  return ('select ' || l_columns || ' from p_table');
end;
```

Loop through all the records with column data in the table parameter

Skip if its a number

Make comma separated list of columns

Select only those columns

Using this Macro



```
MACRO@FREEPDB1>desc employees
```

Name	Null?	Type
EMPLOYEE_ID	NOT NULL	NUMBER(6)
FIRST_NAME		VARCHAR2(20)
LAST_NAME	NOT NULL	VARCHAR2(25)
EMAIL	NOT NULL	VARCHAR2(25)
PHONE_NUMBER		VARCHAR2(20)
HIRE_DATE	NOT NULL	DATE
JOB_ID	NOT NULL	VARCHAR2(10)
SALARY		NUMBER(8,2)
COMMISSION_PCT		NUMBER(2,2)
MANAGER_ID		NUMBER(6)
DEPARTMENT_ID		NUMBER(4)

```
select *  
from no_numeric_cols (employees)  
fetch first 3 rows only  
;
```

```
MACRO@FREEPDB1>select *  
2 from no_numeric_cols (employees)  
3 fetch first 3 rows only  
4* ;
```

FIRST_NAME	LAST_NAME	EMAIL	PHONE_NUMBER	HIRE_DATE	JOB_ID
Steven	King	SKING	515.123.4567	2003-06-17 00:00:00	AD_PRES
Neena	Kochhar	NKOCHHAR	515.123.4568	2005-09-21 00:00:00	AD_VP
Lex	De Haan	LDEHAAN	515.123.4569	2001-01-13 00:00:00	AD_VP

```
3 rows selected.
```

dbms_tf.columns_t



It's a collection of strings

```
TYPE COLUMNS_T IS TABLE OF DBMS_QUOTED_ID;
```

Use pseudo-operator columns() to pass it a list of columns

Example:

```
from my_macro (columns(first_name, email, salary, "123_column"))
```

Each collection-row contains one of the column names

Columns do not have to exist anywhere, but have to be valid identifiers

If needed enclosed in double quotes

Parameters of this type **can NOT be included** as a placeholder in the sql string

Need to be processed by the code

Result must be concatenated to the sql string

Example dbms_tf.columns_t



```
create or replace function null_count
(p_columns in dbms_tf.columns_t
)
return varchar2 sql_macro (scalar)
is
  l_sql          varchar2(1000);
  l_columns      varchar2(1000);
begin
  select listagg('nvl2(' || column_value || ',0,1)'
                '+'
                )
  into l_sql
  from table (p_columns)
  ;

  return (l_sql);
end;
```

```
select location_id, postal_code, state_province
from locations
where null_count
      (columns(postal_code, state_province)
) > 0
```

LOCATION_ID	POSTAL_CODE	STATE_PROVINCE
1000	00989	
1100	10934	
1300	6823	
2000	190518	
2300	540198	
2400		

6 rows selected.

dbms_tf.columns_with_type_t

A collection of records of type `COLUMN_METADATA_T`
(See `TABLE_T` type)

Works similar to `COLUMNS_T`, but....

- Need to supply datatypes with the column names

```
from my_macro (columns(first_name varchar2(50), salary number(10,2)))
```

- Find the column names in

```
p_columns_param(i_row).name
```

- Find the datatypes like this

```
dbms_tf.column_type_name (p_columns(i_col))
```

But....?? Hold on....??



- The values of a 'columns' parameter can be accessed inside the macro...
- Whatever you do with the parameter, the result must be concatenated to the sql...
- The input must be valid identifiers...
- But they do not have to be existing columns...
- Anything is a valid identifier if enclosed in double quotes 🤪

First Conclusion:

This can be (ab)used to pass in scalar parameters that are accessible within the macro

But worse.....

Let's be evil



```
create or replace function sql_injection
(p_columns in dbms_tf.columns_t
)
return varchar2 sql_macro (table)
is
  l_stmt varchar2(32767);
  l_cols varchar2(32767);
begin
  l_stmt := 'select first_name
            ,      last_name
            from employees
            where '||replace(p_columns(1), '"')||' = 103
            ';

  return l_stmt;
end;
```

```
select *
from sql_injection(columns(employee_id))
```

```
FIRST_NAME LAST_NAME
-----
Alexander Hunold

1 row selected.
```

```
select *
from sql_injection(columns(manager_id))
```

```
FIRST_NAME LAST_NAME
-----
Bruce Ernst
David Austin
Valli Pataballa
Diana Lorentz

4 rows selected.
```

```
select *
from sql_injection
      (columns("1=1 or null"))
```

```
FIRST_NAME LAST_NAME
-----
Ellen Abel
Sundar Ande
Mozhe Atkinson
Jennifer Whalen
Eleni Zlotkey

107 rows selected.
```

Macros and CTEs – Part 1



```
create or replace function first_3_rows
(p_table in dbms_tf.table_t
)
return varchar2 sql_macro (table)
is
  l_stmnt  varchar2(32767);
begin
  l_stmnt := 'select count(*)
              over() as total
              , t.*
              from p_table t
              fetch first 3 rows only
              ';

  return l_stmnt;
end;
```

Input for a TABLE_T parameter
can also be a CTE

```
with
  commissioned as
  (
    select first_name
    ,      last_name
    ,      commission_pct
    from   employees
    where  commission_pct is not null
  )
select *
from   first_3_rows (commissioned)
;
```

```
TOTAL FIRST_NAME LAST_NAME COMMISSION_PCT
-----
35 John Russell 0,4
35 Karen Partners 0,3
35 Alberto Errazuriz 0,3

3 rows selected.
```

Macros and CTEs – Part 2



```
with
function first_3_rows
(p_table in dbms_tf.table_t
)
return varchar2 sql_macro (table)
is
  l_stmnt    varchar2(32767);
begin
  l_stmnt := 'select count(*)
              over() as total
              ,      t.*
              from  p_table t
              fetch first 3 rows only
              ';

  return l_stmnt;
end;
--
select *
from first_3_rows(countries)
```

Macros can be defined in a with clause

TOTAL	COUNTRY_ID	COUNTRY_NAME	REGION_ID
25	AR	Argentina	2
25	AU	Australia	3
25	BE	Belgium	1

3 rows selected.

Useful?



But you can.

Macros and CTEs – Part 3



```
create or replace function first_3_rows
(p_table in dbms_tf.table_t
)
return varchar2 sql_macro (table)
is
  l_stmnt  varchar2(32767);
begin
  l_stmnt := 'select count(*)
              over() as total
              , t.*
              from p_table t
              fetch first 3 rows only
              ';

  return l_stmnt;
end;
```

Macros can NOT be used in a CTE

```
with
  couple_of_locations as
  (
    select *
    from first_3_rows (locations)
  )
select *
from couple_of_locations
```

```
with
ERROR at line 1:
ORA-64630: unsupported use of SQL macro: use of SQL macro inside WITH clause is not supported
```

Macros and CTEs – Part 4



```
create or replace function give_me_rows
(p_rows in integer
)
return varchar2 sql_macro (table)
is
  l_stmt varchar2(32767);
begin
  l_stmt := 'with
            many_rows as
            (
              select level as id
              from dual
              connect by level <= 1000
            )
            select id
            from many_rows
            where id <= p_rows
            ';

  return l_stmt;
end;
```

The query returned by a table macros can contain a with clause

```
select *
from give_me_rows(4)
```

```
MACRO@FREEPDB1>select *
  2  from give_me_rows(4)
  3* ;

  ID
----
  1
  2
  3
  4

4 rows selected.
```

Macros and CTEs – Part 4..... However.....



```
create or replace function give_me_rows
(p_rows in integer
)
return varchar2 sql_macro (table)
is
  l_stmt varchar2(32767);
begin
  l_stmt := 'with
            many_rows as
            (
              select level as id
              from dual
              connect by level <= p_rows
            )
            select id
            from many_rows
            ' ;

  return l_stmt;
end;
```

Currently parameters can **NOT** be referenced inside that with clause

```
select *
from give_me_rows(4)
```

```
MACRO@FREEPDB1>select *
  2 from give_me_rows(4)
  3* ;
select *
                                     *
ERROR at line 1:
ORA-00904: "P_ROWS": invalid identifier
```

Warning

Macros are implicitly deterministic,
Can't add the deterministic keyword to its definition

Make sure its behavior is indeed deterministic

```
create or replace function current_count
return varchar2 sql_macro (scalar)
is
    l_count integer;
begin
    dbms_output.put_line
        ('### MACRO IS EXECUTED ###');

    select count(*)
    into    l_count
    from    job_history
    ;

    return (l_count);
end;
```

```
select count(*)           as real_count
,      current_count ()  as macro_count
from    job_history
```

```
REAL_COUNT  MACRO_COUNT
-----
              10              10
1 row selected.
### MACRO IS EXECUTED ###
```

```
MACRO@FREEPDB1>delete from job_history;
10 rows deleted.
```

```
REAL_COUNT  MACRO_COUNT
-----
              0              10
1 row selected.
MACRO@FREEPDB1>
```

Macros in packages



A macro can be in a package

However...

Won't compile if used in
the same package

```
create or replace package try_macro
is
  function current_datetime
  return varchar2 sql_macro (table);

  procedure use_macro;
end try_macro;
```

```
create or replace package body try_macro is
  function current_datetime
  return varchar2 sql_macro (table) is
  begin
    return
      ('select sysdate as now from dual');
  end current_datetime;

  procedure use_macro is
    l_now date;
  begin
    select now into l_now
    from try_macro.current_datetime();
  end use_macro;
end try_macro;
```

```
LINE/COL ERROR
-----
16/5      PL/SQL: SQL Statement ignored
18/12     PL/SQL: ORA-62565: The SQL Macro method failed with error(s).
          ORA-04067: not executed, package body "MACRO.TRY_MACRO" does not exist
          ORA-06508: PL/SQL: could not find program unit being called: "MACRO.TRY_MACRO"
          ORA-06512: at line 5
```

Makes sense, because...

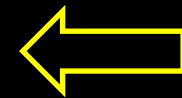
```
create or replace function current_datetime  
return varchar2 sql_macro (table)  
is  
begin  
  dbms_output.put_line  
  ('### MACRO IS EXECUTED ###');  
  
  return ('select sysdate as now  
         from dual'  
        );  
end current_datetime;
```

```
create or replace procedure use_macro  
is  
  l_now date;  
begin  
  select now  
  into   l_now  
  from   current_datetime()  
  ;  
end use_macro;
```

Will macros cause the revival
of standalone functions?

Time will tell.

```
MACRO@FREEPDB1>create or replace procedure use_macro  
2  is  
3    l_now date;  
4  begin  
5    select now  
6    into   l_now  
7    from   current_datetime()  
8    ;  
9  end use_macro;  
10* /  
### MACRO IS EXECUTED ###  
  
Procedure created.  
MACRO@FREEPDB1>
```



Which functions are macros?

New column **SQL_MACRO** in user_ | all_ | dba_procedures views

The Docs:

Indicates whether the procedure is a SQL macro. Possible values:

- **SCALAR**: The procedure is a SQL macro for a scalar expression
- **TABLE**: The procedure is a SQL macro for a table expression
- **NULL**: The procedure is not a SQL macro

```
select nvl(p.sql_macro
         , 'NOT a macro'
        )      as sql_macro
,      count(*)      as macro_count
from    dba_procedures p
where   p.owner      = 'MACRO'
group by p.sql_macro
order by p.sql_macro
```

```
SQL_MACRO      MACRO_COUNT
-----
SCALAR          1
TABLE           6
NOT a macro     1

3 rows selected.
```

```
select nvl(p.sql_macro
         , 'NOT a macro'
        )      as sql_macro
,      count(*)      as macro_count
from    dba_procedures p
join    dba_users      u
on      u.username     = p.owner
and     u.oracle_maintained = 'Y'
group by p.sql_macro
order by p.sql_macro
```

```
SQL_MACRO      MACRO_COUNT
-----
NULL           36720
SCALAR          1
TABLE           2
NOT a macro     2144

4 rows selected.
```

Oracle itself takes "NULL" very literally





“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 !!!