APEX connect

2020 [online]
#APEXCONN20

Polymorphic Table Functions & Qualified Expressions

Robert Marz

Robert Marz – Independent Consultant

Primary Role

- Senior Technical Architect
- with database centric view of the world

DOAG (German Oracle User Group)

- Active Member of Database Community
- Responsible for Cloud Topics

@RobbieDatabee

robbie.databee.org

robert.marz
@databee.de

ORACLE ACE

500+ Technical Experts Helping Peers Globally

ORACLE ACE PROGRAM

---

bit.ly/OracleACEProgram

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

PL/SQL: Requirements for Web Development and Microservices

Data Formats
- JSON
- YAML

Business Logic
- Encapsulate
- APIs
- Re-Usability

constant improvements

Release 12:
- JSON Support

Release 18:
- Polymorphic Table Functions
- Qualified Expressions

Qualified Expressions

Qualified Expressions

Constructors for PL/SQL Types	Associative Arrays	
	Records	
Initialize Types	Function Call	Same Name as Type
	Provide Values	By Name By Position
Syntactical Sugar	Shorter Code	
	Better readability	

Declaring Types

```
type numbers_ty is table of number
   index by pls_integer;

type user_properties_ty is record(
     is_ops boolean
   , is_dev boolean
   , email varchar2(255)
);
type users_ty is table of user_properties_ty
   index by varchar2(32);
```

Initializing Types **before** 18c

```
    numbers   numbers_ty;
    appUsers  users_ty;
begin
    numbers(2) := 10.5;
    numbers(3) := 65;
    numbers(1) := 3.14;

    appUsers('alice').is_ops := false;
    appUsers('alice').is_dev := true;
    appUsers('alice').email  := 'alice@ioug.org';

    appUsers('bob').is_ops := true;
```

Qualified Expressions: Initializing Types in 18c and up

```
-- Qualified Expressions are Constructors
 -- Initialization by Name
 numbers   numbers_ty := numbers_ty(2 => 10.5, 3 => 65, 1 => 3.14);
 appUsers  users_ty := users_ty(
   'alice'    => user_properties_ty(is_ops => false,
                                    is_dev => true,
                                    email  => 'alice@ioug.org'),
   'robbie'   => user_properties_ty(email  => 'robbie@doag.org',
                                    is_dev => true,
                                    is_ops => true ),
 -- Initialization by Position
   'bob'      => user_properties_ty(true, false,'bob@ioug.org'));
```

Index by pls_integer

Index by varchar2

Qualified Expressions: Demo

```
-- Qualified Expressions in 18c
declare
  type numbers_ty is table of number
    index by pls_integer;

  type user_properties_ty is record(
    is_ops  boolean
  , is_dev  boolean
  , email   varchar2(255)
  );

  type users_ty is table of user_properties_ty
    index by varchar2(32);
  -- Qualified Expressions are Constructors for Types
  numbers   numbers_ty := numbers_ty(2=>10.5, 3=>65, 1=> 3.14);  --
Initialization by Name
  appUsers  users_ty := users_ty(
      'alice'   => user_properties_ty(is_ops => false, is_dev => true,
email=>'alice@rmoug.org')
    , 'bob'     => user_properties_ty(true, false, 'bob@rmoug.org') --
Initialization by Position
    , 'robbie'  => user_properties_ty(email=>'robbie@doag.org', is_dev =>
true, is_ops => true )
    );
  usr varchar2(32);
begin
  for idx in 1..numbers.count loop
    dbms_output.put_line('Index: '||idx||' value: '||numbers(idx));
  end loop;

  usr := appUsers.first;
  while usr is not null loop
    dbms_output.put_line(case appUsers(usr).is_dev when true then 'Dev' else
'' end
                         ||case appUsers(usr).is_ops when true then 'Ops   '
else '   ' end
                         ||' User: '||usr|| ' email: '||appUsers(usr).email
                          );
    usr := appUsers.next(usr);
  end loop;
end;
/
```

Not on 18c, yet? Try LiveSQL

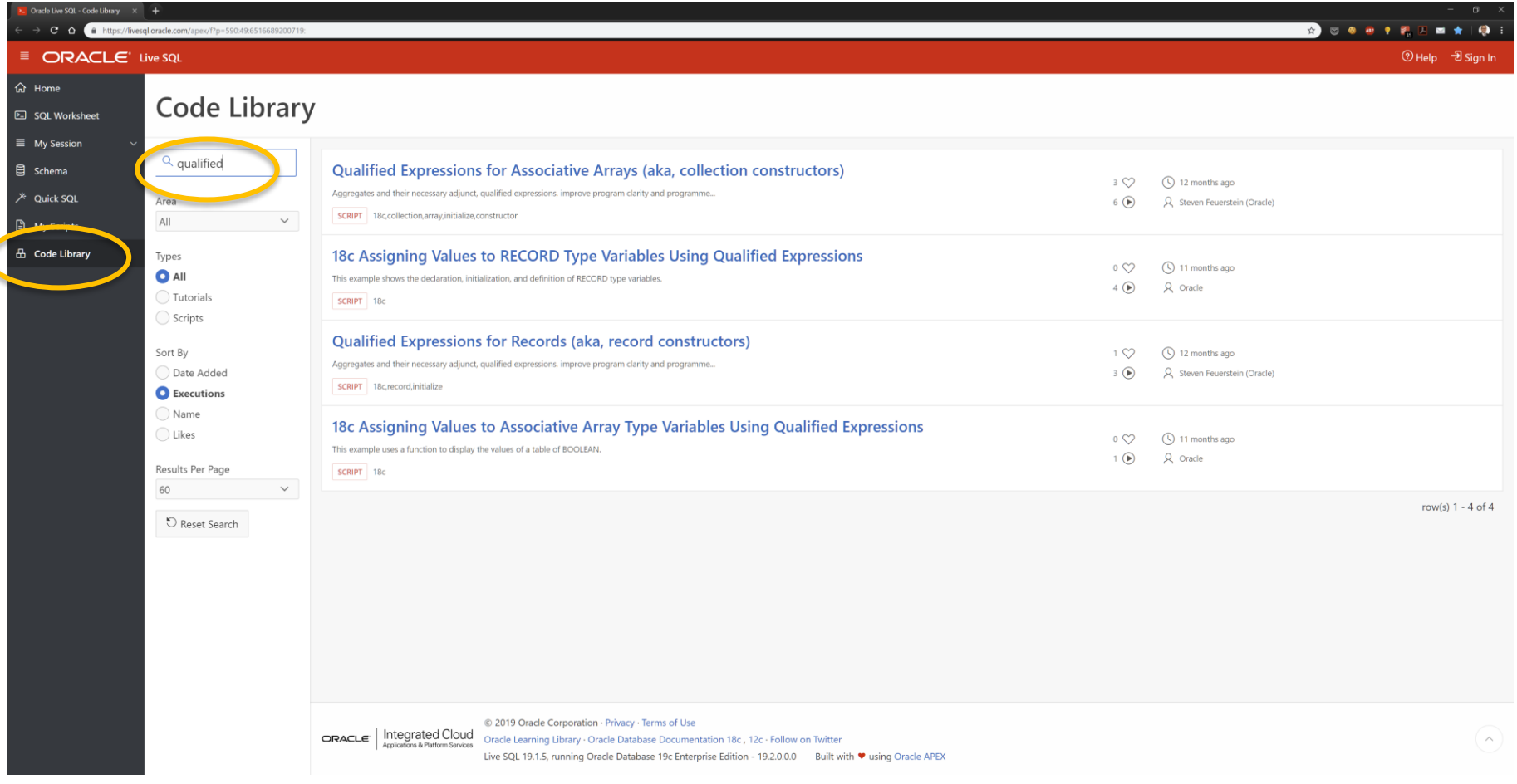


Table Functions (Classic)

Classic Table Functions

Generate Data	arbitrary number of Rows
Row Structure	defined at compile time
	PL/SQL Types
Nesting Possible	Pipelining
PL/SQL Function	Package OK
	arbitrary Parameters

Classic Table Functions - Definition

```
-- Classic Table Functions (9i and up)

create or replace package tf
as
    type fibo_rec is record (fibo number, ind number, tmp number);
    type fibo_tab is table of fibo_rec;

    function fibonacci(fibolimes in number)
        return tf.fibo_tab pipelined;
end tf;
/
```

Classic Table Functions - Implementation

```
create or replace package body tf
as
    function fibonacci(fibolimes in number)
        return tf.fibo_tab pipelined
    is
        fibo fibo_rec; -- Type with column definition
    begin
        fibo.ind := 1; fibo.fibo := 1; -- Pre 18c Init
        while fibo.fibo <= fibolimes
        loop
            pipe row (fibo);
            fibo.tmp := fibo.ind + fibo.fibo;
            fibo.fibo := fibo.ind; fibo.ind := fibo.tmp;
        end loop;
    end fibonacci;
end tf;
```

Classic Table Functions - Implementation

```
create or replace package body tf
as
    function fibonacci(fibolimes in number)
        return tf.fibo_tab pipelined
    is
         fibo fibo_rec := fibo_rec(1,1,null); -- Qualified Expression
    begin
        -- fibo.ind := 1; fibo.fibo := 1; -- Pre 18c Init
        while fibo.fibo <= fibolimes
        loop
            pipe row (fibo);
            fibo.tmp := fibo.ind + fibo.fibo;
            fibo.fibo := fibo.ind; fibo.ind := fibo.tmp;
        end loop;
    end fibonacci;
end tf;
```

Classic Table Functions - Implementation

```
select fibo
    from table(tf.fibonacci(15));


 -- Since Oracle 12.2
 -- the table()-Operator
 -- is obsolete

 select fibo
     from tf.fibonacci(15);
```

```
      FIBO
---------
         1
         1
         2
         3
         5
         8
        13

7 rows selected.
```

Polymorphic Table Functions

Polymorphic Table Functions: The Idea

Polymorphic Table Functions (PTF)

Modify Source Table	Add / Remove / Modify
	Rows & Columns
Generic Extension	Like a View but more procedural
	Works for arbitrary input tables
Business Logic	API for Analysts
	Hide complexity
	make dynamic SQL available
SQL 2016 Standard	Not (yet) completely implemented
	Some (minor) discrepancies

Polymorphic Table Function Benefits

PTF Benefits

Minimal data-movement	Only columns of interest are passed to PTF
Predicates, Projections, Partitioning	pushed into underlying table/query (where semantically possible)
Bulk data transfer	Into and out of PTF
Parallelism based on	type of PTF query specified partitioning (if any)


DEMO
delete
©nitroworld - stock.adobe.com

PTF: Basic Patterns

Taking an existing rowset and…

- Column-based **EXPANSION**
  - Calculating/deriving a new column value
- Row-based **EXPANSION**
  - Data pivot operation
- Column-based **REDUCTION**
  - Data unpivot operation
- Row-based **REDUTION**
  - Data aggregation/reduction operation

No existing rowset to process…

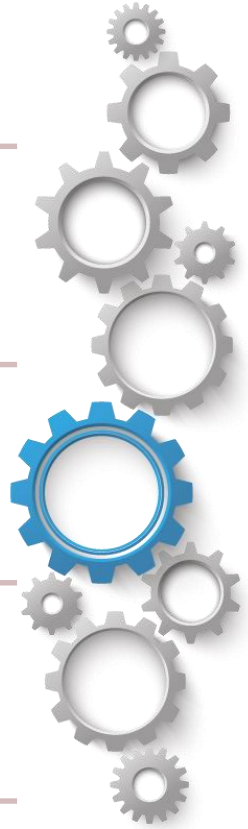
- Rowset **GENERATOR**
  - Creates new rows and columns
  - Importing a CSV file

Thanks to Keith Laker, Oracle @SQLBarista

Polymorphic Table Functions: The Implementation

PTF Implementation

Input	Oracle: exactly one Table
	Column Lists
	arbitrary additional Parameters
Row Structure “Describe”	Fixed at SQL-Execution Time
	PL/SQL Function
Implementation	PL/SQL Package per PTF
	Heavy use of PL/SQL Tables
	DBMS_TF – Helper & Utilites
Performance	New Execution Plan operation
	Not always needed



Polymorphic Table Functions: Oracle's Aproach

What makes a PTF

Function	without	body
	references	Package
Package	required function	`describe`

	optional procedures	`open`
		`fetch_rows`
		`close`

```
create or replace
  function add_lables
     (tabname table,
      colnames columns)
  return
    table pipelined
    row polymorphic
    using lables; -- Package Name

select *
from add_lables(
       emp, columns(empno, mgr));
```

PTF SQL Enhancement: Varadic Pseudo Operators

```
select *
  from add_lables(emp, columns(empno, mgr) );
```

Variadic Pseudo Operators columns()

- operates with a variable number(>= 1) of operands
- introduced in 18c to support PTF
- can only appear in argument lists of PTF
- parsed by SQL Engine
- converted to corresponding DBMS_TF-types
- passed as Input parameter to the DESCRIBE-Function

PTF API Package: Describe Function

Describe Function

invoked by SQL-Engine at parse-time

determines the row_type

new & removed columns
returns dbms_tf.describe_t Table

marks columns for processing

“pass-through” – unchanged, not moved
“for read” - passed to fetch_rows procedure
via dbms_tf.table_t (IN OUT Parameter)

PTF API Package: Open & Close

Open & Close Procedures

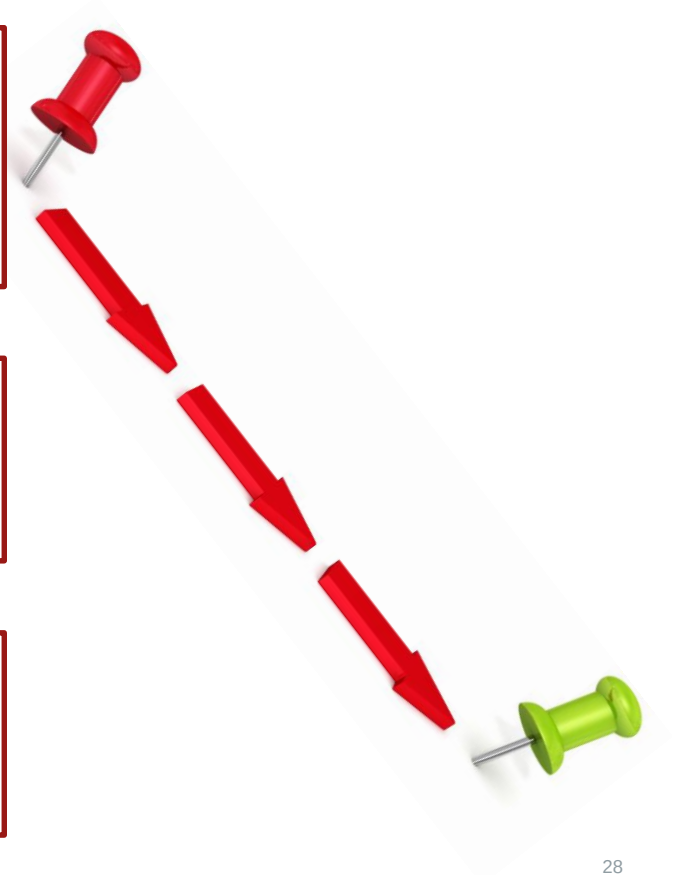
- Setup and Teardown of Environment
- Your instrumentation Code goes here
- Both are optional

Open

- called before first fetch_rows execution
- Initialize Variables

Close

- called at the end, after all fetch_rows
- do cleanup stuff



PTF API Package: Fetch_Rows Procedure

Fetch_rows is the worker

- processes rowsets (chunks of Table Data)
- only columns marked for read
- Needs same scalar parameters as PTF

Database can call multiple times

- for each rowset
- in parallel

Produce & Reduce Data

- fill new columns
- generate new rows
- suppress rows

Polymorphic Table Functions: Flavors

PTF Semantic Types

Determine execution Plan
Impact Performance

Row Semantic PTF

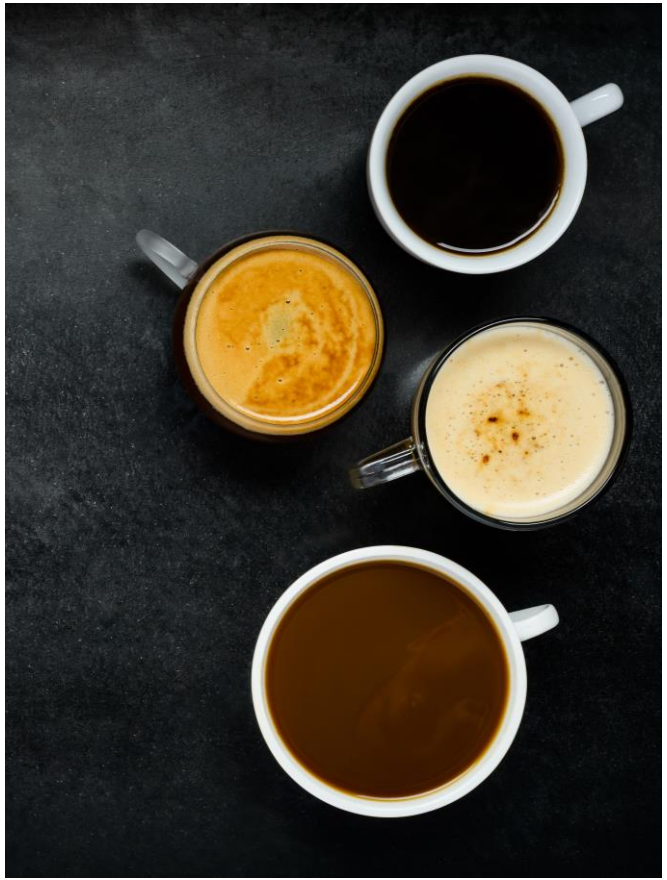
new columns can be derived from current row

```
return table pipelined row polymorphic
```

Table Semantic PTF

works on whole table or partition

```
return table pipelined table
  polymorphic
```



Polymorphic Table Functions: Restrictions

Datatype Restrictions

passthrough is possible with any type
"for read" and new columns must be scalar datatypes
dbms_tf.supported_type function

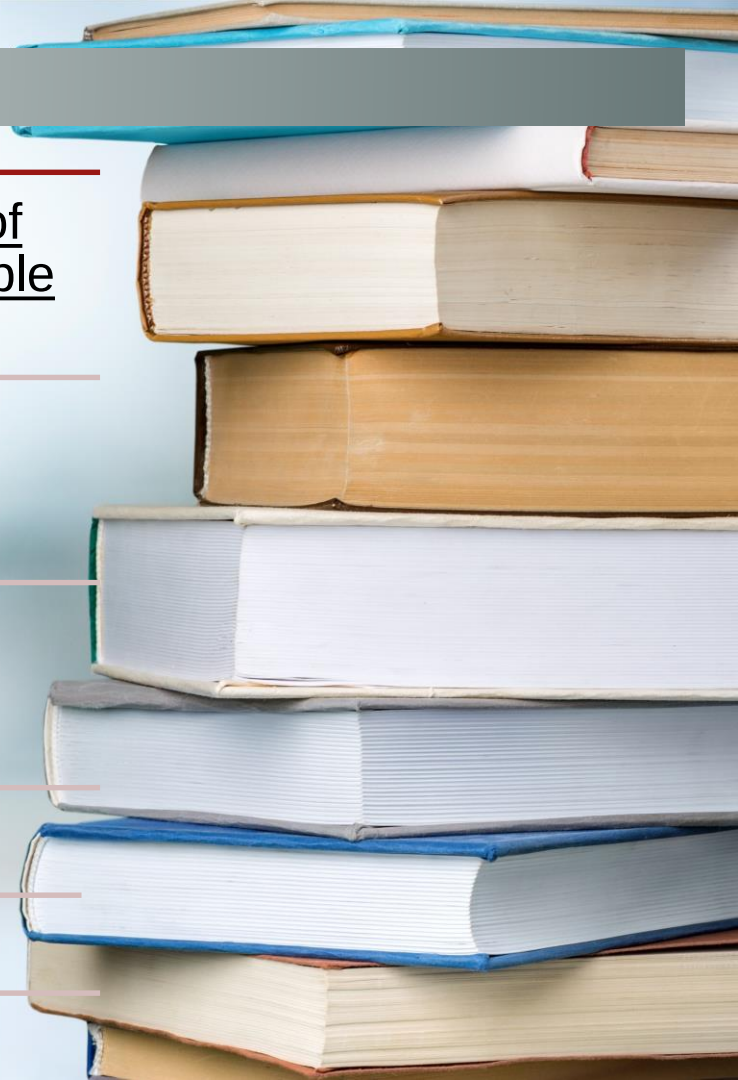
Invocation and Execution Restrictions

PTF cannot be nested in from clause
  workaround: Use with-clause
PTF cannot be an argument to a (classic) table function
PTF yields no rowids
PARTITION BY and ORDER BY only work with Table Semantics PTF
DESCRIBE function cannot be called directly

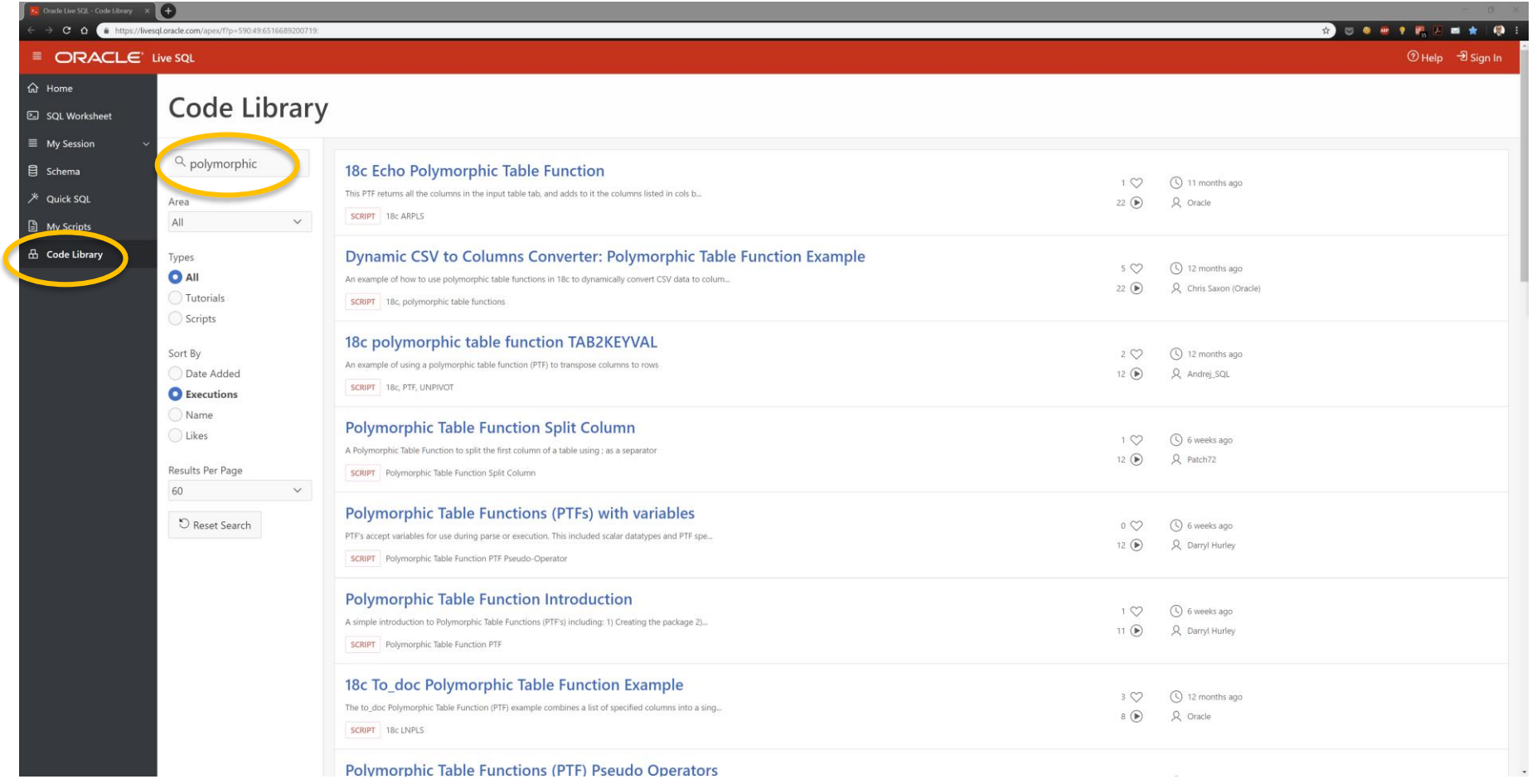
Polymorphic Table Functions: Reading List

Want to know more?

Database PL/SQL Language Reference	12.6 Overview of Polymorphic Table Functions
PL/SQL Packages and Types Reference	173 DBMS_TF
LiveSQL	Code Library
Google	Blog Posts
	Presentations



Polymorphic Table Functions: LiveSQL




DEMO
delete
©nitroworld - stock.adobe.com

Demo: PTF addTags – Package Spec

```
create or replace package ptf_tags
as
  function describe(tabname in out dbms_tf.table_t,
                    colnames in dbms_tf.columns_t,
                    tag_string in varchar2)
    return dbms_tf.describe_t;

  procedure fetch_rows(tag_string in varchar2);
end ptf_tags;
/
```

Demo: PTF addTags – PTF Definition

```
create or replace
function add_tags(tabname table,
                  colnames columns,
                  tag_string varchar2)
  return
    table pipelined
    row polymorphic
    using ptf_tags; -- Package Name
/
```

Demo: PTF addTags – Package Body – Function define

```
function describe(tabname in out dbms_tf.table_t, colnames in dbms_tf.columns_t, tag_string in varchar2)
  return dbms_tf.describe_t
as
  new_cols dbms_tf.columns_new_t;
  begin
    for i in 1 .. tabname.column.count -- loop over all table columns
    loop -- skip columns with unsupported data types
      continue when not dbms_tf.supported_type(tabname.column(i).description.type);
      for j in 1 .. colnames.count
      loop
        if (tabname.column(i).description.name = colnames(j)) -- is the column in the colnames table?
        then
          tabname.column(i).for_read := true;
          tabname.column(i).pass_through := true;
          new_cols(i) := tabname.column(i).description; -- copy column in new_cols()
          -- set datatype to varchar2
          new_cols(i).type := dbms_tf.type_varchar2;
          new_cols(i).max_len := 4000;
          new_cols(i).name := '"TAGGED_' -- set new column name
                           || regexp_replace(new_cols(i).name, '^"|"$') -- remove trailing or leading '„‘
                           || '„‘';
          exit;
        end if;
      end loop;
    end loop;
    return dbms_tf.describe_t(new_columns => new_cols);
end describe;
```

Demo

Demo: PTF addTags – Package Body – Procedure fetch_rows

```
procedure fetch_rows(tag_string in varchar2)
as
  rowset dbms_tf.row_set_t;
  rowcount pls_integer;
  colcount pls_integer;
  tag_value varchar2(4000);
begin
  dbms_tf.get_row_set(rowset, rowcount, colcount);
  dbms_output.put_line(rowcount||' - '||colcount);
  for i in 1..rowset.count loop
    dbms_output.put_line(rowset(i).description.name||' - '||rowset(i).tab_varchar2.count||' -
'||rowset(i).tab_number.count);
  end loop;
  for i in 1..rowcount loop
    for j in 1..colcount loop
      -- The new columns are varchar2(4000)
      tag_value:= case rowset(j).description.type
                    when dbms_tf.type_varchar2 then rowset(j).tab_varchar2(i)
                    when dbms_tf.type_number then to_char(rowset(j).tab_number(i))
                    else 'DATATYPE NOT IMPLEMENTED‘
                  end;
      rowset(j).tab_varchar2(i) := replace(tag_string, '%s',tag_value);
    end loop;
  end loop;
  dbms_tf.put_row_set(rowset);
end fetch_rows;
```

Conclusion

Modern Features for a cool Language

Qualified Expressions

- more than syntactical sugar
- make code shorter
- improve readability

Polymorphic Table Functions

- Simplifies SQL for non-technical Users
- Great potential
- Powerful & flexible Enhancement to SQL

* or at LiveSQL.oracle.com