

# VIRTUAL PRIVATE DATABASE AND USING ORACLE APPLICATION CONTEXTS

*Robert A. Corfman, The Boeing Company*

|   |    |
|---|----|
| Introduction.....   | 2  |
| Oracle Application Contexts.....                                  | 2  |
| UserEnv Application Context.....                                  | 2  |
| Manipulating userenv application context values.....              | 3  |
| UserEnv Client_Info.....  | 3  |
| UserEnv Client_Identifier.....                                    | 3  |
| UserEnv Current_Schema Info.....                                  | 3  |
| UserEnv NLS_DATE_FORMAT Setting.....                              | 4  |
| UserEnv and the CURRENT_USER Setting.....                         | 4  |
| Custom Application Contexts.....                                  | 7  |
| Application Context Definition and Maintenance.....               | 7  |
| Application Context Package.....                                  | 8  |
| Accessing values in an Application Context.....                   | 10 |
| Simple use of a Global Application Context.....                   | 11 |
| Username and Client Identifier with Global Context.....           | 12 |
| Oracle Virtual Private Database.....                              | 14 |
| Setting the stage.....  | 15 |
| Defining Items.....   | 15 |
| Defining Security Context.....                                    | 15 |
| Creating a Policy Function.....                                   | 17 |
| Introducing the DBMS_RLS Package.....                             | 18 |
| Implementing Row-Level Security Policy.....                       | 18 |
| Implementing Column-Level Security Policy.....                    | 19 |
| Column-Level Security Policy Function.....                        | 20 |
| Column-Level Security by row filtering.....                       | 20 |
| Dropping a Fine-Grained Access Security Policy.....               | 21 |
| Column-Level Security by Value Nulling.....                       | 22 |
| Policy Groups.....  | 22 |
| Overhead Setup for Policy Groups Example.....                     | 23 |
| Creating the data.....  | 23 |
| Creating the Application Context and Security Policy Package..... | 23 |
| SYS_DEFAULT Policy Group.....                                     | 24 |
| Creating New Policy Groups and Associated Policies.....           | 25 |
| Using Driving Context to Selectively Enforce Policies.....        | 26 |
| Further Information in Oracle's Documentation.....                | 28 |

## INTRODUCTION

This paper is intended to provide an overview of Oracle Virtual Private Database (VPD) using fine-grained access control as well as Oracle application contexts and their close relationship to VPD. The paper is broken into two major sections, introduction and use of application contexts and definition of security policies with fine-grained access control.

## ORACLE APPLICATION CONTEXTS

Application contexts in Oracle are name value pair spaces which can be defined to be session specific or used globally across multiple sessions. There is significant infrastructure in place to ensure the secure updating of the values in these spaces and provide access to the stored data in a consistent and high-performance manner.

Unique context name spaces are defined at a global-level along with a pl/sql package in a specific schema which controls updates to the values in these application contexts. All updates to the context must be performed through this package, or sub-procedures called from the package. This ensures the integrity of the data with the package preventing any unauthorized sources from performing updates to the data.

Data in the application contexts is typically accessed using the `sys_context` function, but is alternatively visible in the `session_context` or `global_context` views. `sys_context` usually acts as a bind variable in queries and is used in the format `sys_context('context_name','value_name')` returning the string value defined in the *active* context.

### *USERENV APPLICATION CONTEXT*

Oracle provides a predefined application context, `USERENV`, which contains many session specific values accessible through sql or procedures. This predefined environment allows common access to important session information such as who logged in, what ip address are they at, who is the current user, what sql statement is being executed (could be important implementing fine-grained access control policies), what is the current schema, etc.

The complete list of defined values accessible in the `userenv` application context:

- |                                 |                           |                              |
|---------------------------------|---------------------------|------------------------------|
| • Action*                       | • DB_Name                 | • NLS_Date_Language          |
| • Audited_CursorID              | • DB_Unique_Name*         | • NLS_Sort                   |
| • Authenticated_Identity*       | • EntryID                 | • NLS_Territory              |
| • Authentication_Data           | • Enterprise_Identity*    | • OS_User                    |
| • Authentication_Method*        | • External_Name (9i only) | • Policy_Invoker*            |
| • Authentication_Type (9i only) | • FG_Job_ID               | • Proxy_Enterprise_Identity* |
| • BG_Job_ID                     | • Global_Context_Memory   | • Proxy_Global_UID*          |
| • Client_Identifier             | • Global_UID*             | • Proxy_User                 |
| • Client_Info                   | • Host                    | • Proxy_UserID               |
| • Current_Bind*                 | • Identification_Type*    | • Server_Host*               |
| • Current_Edition_ID**          | • Instance                | • Service_Name*              |
| • Current_Edition_Name**        | • Instance_Name*          | • Session_Edition_ID**       |
| • Current_Schema                | • IP_Address              | • Session_Edition_Name**     |
| • Current_SchemaID              | • IsDBA                   | • Session_User               |
| • Current_SQL                   | • Lang                    | • Session_UserID             |
| • Current_SQL#*                 | • Language                | • SessionID                  |
| • Current_SQL_Length*           | • Module*                 | • SID*                       |
| • Current_User                  | • Network_Protocol        | • StatementID*               |
| • Current_UserID                | • NLS_Calendar            | • Terminal                   |
| • Database_Role**               | • NLS_Currency            |                              |
| • DB_Domain                     | • NLS_Date_Format         |                              |

\* introduced in 10g \*\* introduced in 11g

The definition for each of these `userenv` values is discussed in the Oracle documentation.

### MANIPULATING USERENV APPLICATION CONTEXT VALUES

As with all application contexts in Oracle, the user is not allowed to directly manipulate any values. That said, there are several userenv values which can be modified using functions provided by the database.

#### USERENV CLIENT\_INFO

The `dbms_application_info.set_client_info` procedure can be used to update the `client_info` value. Client Info has been available for several version of Oracle. It can still be used, but it is probably better to use custom application contexts as they provide more flexibility in data validation. That said, this can be a simple place to store values with minimal setup. This is demonstrated in the following sql\*plus session example:

```
SQL> select sys_context('userenv','client_info') client_info from dual;

CLIENT_INFO
-----

SQL> begin
  2   dbms_application_info.set_client_info(
  3     'This is my application Data. X=ABC;Y=123');
  4 end;
  5 /

PL/SQL procedure successfully completed.

SQL> select sys_context('userenv','client_info') client_info from dual;

CLIENT_INFO
-----
This is my application Data. X=ABC;Y=123

SQL>
```

#### USERENV CLIENT\_IDENTIFIER

The client id is a setting that can be used in a shared login environment (for instance a web server proxy environment with shared login) to enable the application to define the current application level user. The client identifier can be heavily involved when global application contexts are used and will be discussed in more detail in a following section. The `dbms_session` package is used to set the current client identifier as shown in the following sql\*plus session:

```
SQL>
SQL> select sys_context('userenv','client_identifier') client from dual;

CLIENT
-----

SQL> begin
  2   dbms_session.set_identifier('End_User_ID');
  3 end;
  4 /

PL/SQL procedure successfully completed.

SQL> select sys_context('userenv','client_identifier') client from dual;

CLIENT
-----
End_User_ID

SQL>
```

#### USERENV CURRENT\_SCHEMA INFO

The current session can be altered to set a new current default schema, effectively updating the `current_schema` value as shown in the following sql\*plus session example:

```
SQL> conn user1/*****@test_db
Connected.
SQL>
SQL> select sys_context('userenv','current_schema') schema from dual;

SCHEMA
-----
USER1

SQL> alter session set current_schema=user2;

Session altered.

SQL> select sys_context('userenv','current_schema') schema from dual;

SCHEMA
-----
USER2

SQL>
```

### *USERENV NLS\_DATE\_FORMAT SETTING*

Altering the current session's setting for `nls_date_format` will effectively modify the value in the `userenv` application context. This is demonstrated with the following sql\*plus session

```
SQL> select sys_context('userenv','nls_date_format') date_fmt from dual;

DATE_FMT
-----
DD-MON-RR

SQL> alter session set nls_date_format = 'mm/dd/yyyy hh24:mi:ss';

Session altered.

SQL> select sys_context('userenv','nls_date_format') date_fmt from dual;

DATE_FMT
-----
mm/dd/yyyy hh24:mi:ss

SQL>
```

### *USERENV AND THE CURRENT\_USER SETTING*

One of the key concepts in Oracle that has a dramatic affect on access and authority is the difference between the `session_user` and the `current_user`. The `session_user` never changes and it is tied to the login credentials used to access the database. The `current_user`, on the other hand, can change regularly based on the objects being accessed and the definition of code object (definer rights vs. invoker rights).

Grant authority can change through execution of code and is based on the `current_user` value. This is the mechanism that allows a user to access objects through a view or definer rights code which they do not normally have authority to access. As you later define your own fine-grained access rules, you may find `current_user` to be of particular value over `session_user` depending on your requirements.

There are several factors that affect the changing of `current_user` within Oracle. The following sql\*plus session sample demonstrates creation of a definer rights (change current user when executing to the owner) procedure and its execution actually showing this change.

```
SQL> conn user1/*****@test_db
```

```

Connected.
SQL> create or replace package userenv_def_rights is
  2   function get_user_info return varchar2;
  3   end;
  4   /

Package created.

SQL> create or replace package body userenv_def_rights is
  2   function get_user_info return varchar2 is
  3   begin return
  4     'current_user = ' || sys_context('userenv','current_user') || ' ' ||
  5     'session_user = ' || sys_context('userenv','session_user');
  6   end;
  7   end;
  8   /

Package body created.

SQL> grant execute on userenv_def_rights to user2;

Grant succeeded.

SQL> conn user2/*****@test_db
Connected.
SQL>
SQL> set serveroutput on size 30000
SQL>
SQL> begin
  2   dbms_output.put_line('current_user = ' ||
  3     sys_context('userenv','current_user'));
  4   dbms_output.put_line(user1.userenv_def_rights.get_user_info);
  5   end;
  6   /
current_user = USER2
current_user = USER1 session_user = USER2

PL/SQL procedure successfully completed.
SQL>

```

You can clearly see in the above example, that while the package `userenv_def_rights` is executing, the current user changed. Contrast that to the next similar example which is using an invoker rights package instead of the default definer rights. There is very little difference in the code, but a significant difference in the output (the `current_user` does NOT change).

```

SQL> conn user1/*****@test_db
Connected.
SQL> create or replace package userenv_inv_rights AUTHID CURRENT_USER is
  2   function get_user_info return varchar2;
  3   end;
  4   /

Package created.

SQL> create or replace package body userenv_inv_rights is
  2   function get_user_info return varchar2 is
  3   begin return
  4     'current_user = ' || sys_context('userenv','current_user') || ' ' ||
  5     'session_user = ' || sys_context('userenv','session_user');
  6   end;
  7   end;
  8   /

Package body created.

SQL> grant execute on userenv_inv_rights to user2;

```

```

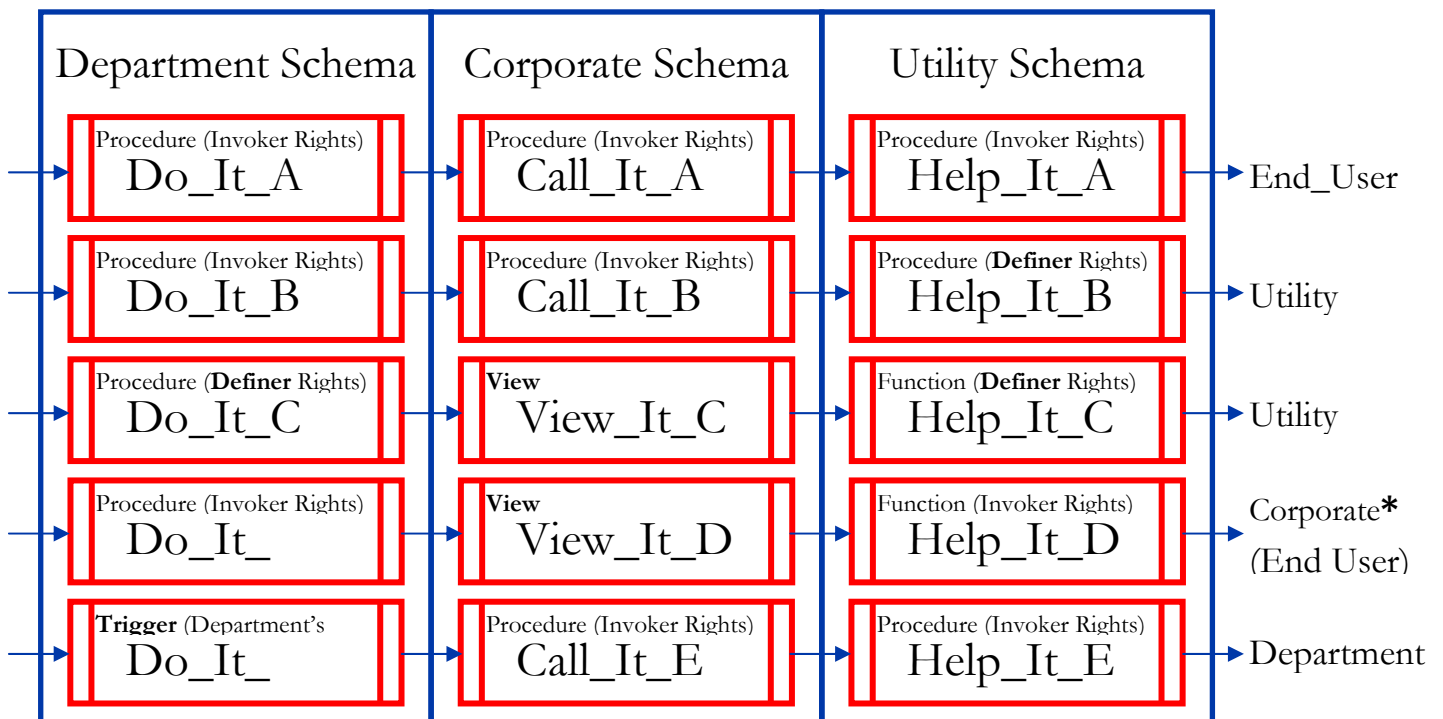
Grant succeeded.

SQL> conn user2/*****@test_db
Connected.
SQL>
SQL> set serveroutput on size 30000
SQL>
SQL> begin
  2  dbms_output.put_line('current_user = ' ||
  3    sys_context('userenv','current_user'));
  4  dbms_output.put_line(user1.userenv_inv_rights.get_user_info);
  5  end;
  6  /
current_user = USER2
current_user = USER2 session_user = USER2

PL/SQL procedure successfully completed.
SQL>

```

The following table shows some of the combinations of how current\_user changes based on the environment. In all Cases, and End\_user session is executing a DO\_IT something located in a department Schema, which calls something in a corporate schema (CALL\_IT) which in turn calls something in a utility schema. When the utility package is executing, this shows what schema/user is the current value of current\_user inside the utility HELP\_IT function:



\*11g documentation indicates current\_user in userenv context does not honor view as definer rights (true in 9i, 10g also)

One point of caution in this, normally userenv.current\_user is very good about showing the actual user in authority between definer and invoker rights. The one exception to this is views, the Oracle documentation on invoker vs. definer rights indicates that grant authority switches in views, making the view owner the current user. The userenv.current\_user value does NOT reflect this when interrogated under a view. This is consistent with the 11g documentation. Additionally, the 10g documentation indicates current\_user is deprecated and session\_user should be used, but the two values clearly have very different meanings. Current\_user was re-instated in 11g, but the documentation was cleared up on how it operates within a

view. In the testing I performed, `userenv.current_user` appears to act consistently in 9i and 10g and matches the behavior as explained in the 11g documentation.

## CUSTOM APPLICATION CONTEXTS

Custom application contexts provide a common secure infrastructure for defining name/value pair data in a consistent manner for use in the database. The contexts are named globally in the database and, depending on the definition, the values are session specific (stored in the UGA) or shared across sessions (stored in the SGA). Access to the values in an application context is provided via the `sys_context` function which typically acts as a bind variable when used in a query context.

Data can be stored in an application context making it accessible as if it were a data sticky note for your application. Instead of regularly referring back to data tables information can be placed in the application context for ongoing rapid access during and across sessions (depending on the context type).

There are three main concepts involved with custom application contexts, defining the application context, defining the application context package, and accessing the context.

### *APPLICATION CONTEXT DEFINITION AND MAINTENANCE*

There are two Oracle system privileges associated with context maintenance, `CREATE ANY CONTEXT` and `DROP ANY CONTEXT`. Created contexts are visible globally and can be identified in the `dba_context` dictionary view. When a context is created, a controlling package must be specified. The only changes to data within the context allowed are through the use of the specified package. The package does not need to exist at the time of context creation, but obviously it must exist before any data is available within the context.

Contexts can be session specific (the default), globally accessible, or externally initialized through OCI or LDAP. The Oracle documentation provides details of the initialization options. Globally accessible contexts are controlled by `client_identifier` and/or `session_user`.

Examples of creating, looking at dictionary information, and dropping a context are demonstrated in this sql\*plus session:

```
SQL> conn user1/*****@test_db
Connected.
SQL> select privilege from user_sys_privs where privilege like '%CONT%';

PRIVILEGE
-----
DROP ANY CONTEXT
CREATE ANY CONTEXT

SQL> create context sec_context using package_name;

Context created.

SQL> create context sec_context using package_name;
create context sec_context using package_name
*
ERROR at line 1:
ORA-00955: name is already used by an existing object

SQL> select * from all_context;

no rows selected

SQL> select * from session_context;

no rows selected

SQL> select * from dba_context;

NAMESPACE                                SCHEMA
-----
PACKAGE                                    TYPE
-----
```

```
SEC_CONTEXT          USER1
PACKAGE_NAME        ACCESSED LOCALLY
```

```
SQL> drop context sec_context;
```

```
Context dropped.
SQL>
```

### *APPLICATION CONTEXT PACKAGE*

The application context package is responsible for any and all data updates to the values in an application context. The values are set using the procedure `dbms_session.set_context` and cleared using `dbms_session.clear_context`. Calls to these functions raise an error (ORA-01031: insufficient privileges) when attempting to update context values unless the authorized package is currently performing the operation, or the authorized package is currently in the call stack. The only exception to this is with the special predefined application context named `CLIENTCONTEXT`. This special context always allows calls to `set_context` and `clear_context` as there is no authorizing application context security package, this context is completely unsecured.

The following sql\*plus sessions demonstrate creation of a simple session application context and associated package. It demonstrates building the context and package, attempting to set the package without the specified package, setting a value, and viewing the context value.

```
SQL> conn user1/*****@test_db
Connected.
SQL>
SQL> create context sec_context using sec_context_pkg;

Context created.

SQL> create or replace package sec_context_pkg is
  2  procedure set_restricted_context;
  3  end;
  4  /

Package created.

SQL> create or replace package body sec_context_pkg is
  2  procedure set_restricted_context is begin
  3      dbms_session.set_context('sec_context','cost','RESTRICTED');
  4  end; end;
  5  /

Package body created.

SQL> select sys_context('sec_context','cost') from dual;

SYS_CONTEXT('SEC_CONTEXT','COS
-----

SQL>
SQL> exec dbms_session.set_context('sec_context','cost','RESTRICTED');
BEGIN dbms_session.set_context('sec_context','cost','RESTRICTED'); END;

*
ERROR at line 1:
ORA-01031: insufficient privileges
ORA-06512: at "SYS.DBMS_SESSION", line 78
ORA-06512: at line 1

SQL>
SQL> column namespace format a12
SQL> column attribute format a12
SQL> column value      format a12
```

```

SQL>
SQL> exec sec_context_pkg.set_restricted_context;

PL/SQL procedure successfully completed.

SQL> select sys_context('sec_context','cost') cost_sec from dual;

COST_SEC
-----
RESTRICTED

SQL> select * from session_context;

NAMESPACE      ATTRIBUTE      VALUE
-----
SEC_CONTEXT    COST           RESTRICTED

SQL>

```

The following sql\*plus session demonstrates updating the application context with a sub-procedure. The actual set\_context is not authorized when the sub-procedure is called directly, but it is allowed when the actual application context package is in the calling stack. This example builds off the prior context and existing context package specification.

```

SQL> create or replace procedure set_restricted_sub is
  2 begin
  3   dbms_session.set_context('sec_context','cost','SUBCOST');
  4 end;
  5 /

Procedure created.

SQL> create or replace package body sec_context_pkg is
  2   procedure set_restricted_context is
  3   begin
  4     set_restricted_sub;
  5   end;
  6 end;
  7 /

Package body created.

SQL> exec set_restricted_sub;
BEGIN set_restricted_sub; END;

*
ERROR at line 1:
ORA-01031: insufficient privileges
ORA-06512: at "SYS.DBMS_SESSION", line 78
ORA-06512: at "USER1.SET_RESTRICTED_SUB", line 3
ORA-06512: at line 1

SQL> exec sec_context_pkg.set_restricted_context;

PL/SQL procedure successfully completed.

SQL> select sys_context('sec_context','cost') from dual;

SYS_CONTEXT('SEC_CONTEXT','COS
-----
SUBCOST

SQL>

```

The application context package is also responsible for clearing values from the application context. This removes the name/value pair from within the context for the specified session or global context. The following sql\*plus session demonstrates setting and clearing a value from the application context.

```
SQL> create or replace package sec_context_pkg is
  2   procedure set_restricted_context;
  3   procedure clear_restricted_context;
  3 end;
  4 /

Package created.

SQL> create or replace package body sec_context_pkg is
  2   procedure set_restricted_context is begin
  3     dbms_session.set_context('sec_context','cost','RESTRICTED');
  4   end;
  5   procedure clear_restricted_context is begin
  6     dbms_session.clear_context('sec_context',null,'cost');
  7   end;
  8 end;
  9 /

Package body created.

SQL> column namespace format a12
SQL> column attribute format a12
SQL> column value      format a12
SQL>
SQL> exec sec_context_pkg.set_restricted_context;

PL/SQL procedure successfully completed.

SQL> select * from session_context;

NAMESPACE      ATTRIBUTE      VALUE
-----
SEC_CONTEXT    COST           RESTRICTED

SQL> exec sec_context_pkg.clear_restricted_context;

PL/SQL procedure successfully completed.

SQL> select * from session_context;

no rows selected

SQL>
```

Though the examples used are very simple sets and clears, very complicated rules could be applied in the package code. These packages enable the application developer to define data validation and restrictions as appropriate to the business requirements associated with the application context. Once defined, there is no way to bypass the package to set context values, and, since the contexts themselves are defined at the database level, applications can be assured the data in the contexts follow the rules established in the application context package.

### *ACCESSING VALUES IN AN APPLICATION CONTEXT*

Values in application contexts are normally accessed using the `sys_context` function. This function will resolve data in global contexts based on the `session_user` and `client_identifier` as appropriate and provides a common mechanism for access whether the application context is session specific or global. Calls to `sys_context` specify the context name, the name of the desired value, and optionally the maximum return length allowed. The actual values stored and visible to the current session are visible in `session_context` for session contexts and `global_context` for global contexts. If the application context is global, then the `session_user` and `client_identifier` can have an impact on the visible data depending on how the values were set for the context (was `session_user` or `client_identifier` specified in the `set_context` call and what are the values in the session).

*SIMPLE USE OF A GLOBAL APPLICATION CONTEXT*

The following sql\*plus session shows the creation of a global context and viewing the values from two sessions. No session\_user or client\_identifer is used when the context values are set. Once set, the global context values are visible to any session\_user without a client\_identifier.

```
SQL> conn user1/*****@test_db
Connected.
SQL>
SQL> create or replace context sec_context using sec_context_pkg
  2  accessed globally;

Context created.

SQL>
SQL> select * from dba_context;

NAMESPACE      SCHEMA                                PACKAGE
-----
TYPE
-----
SEC_CONTEXT     USER1                                SEC_CONTEXT_PKG
ACCESSED GLOBALLY

SQL>
SQL> conn user2/*****@test_db
Connected.
SQL>
SQL> select * from global_context;

no rows selected

SQL> exec user1.sec_context_pkg.set_restricted_context;
BEGIN user1.sec_context_pkg.set_restricted_context; END;

*
ERROR at line 1:
ORA-06550: line 1, column 7:
PLS-00201: identifier 'USER1.SEC_CONTEXT_PKG' must be declared
ORA-06550: line 1, column 7:
PL/SQL: Statement ignored

SQL>
SQL> conn user1/*****@test_db
Connected.
SQL>
SQL> exec user1.sec_context_pkg.set_restricted_context;

PL/SQL procedure successfully completed.

SQL> select * from global_context;

NAMESPACE      ATTRIBUTE      VALUE      USERNAME
-----
CLIENT_IDENTIFIER
-----
SEC_CONTEXT     COST          RESTRICTED

SQL> select * from session_context;

no rows selected
```

```

SQL> conn user2/*****@test_db
Connected.
SQL>
SQL> select * from global_context;

NAMESPACE      ATTRIBUTE      VALUE          USERNAME
-----
CLIENT_IDENTIFIER
-----
SEC_CONTEXT     COST          RESTRICTED

SQL> conn user1/*****@test_db
Connected.
SQL>
SQL> select count(*) from global_context;

COUNT(*)
-----
1

SQL> exec sec_context_pkg.clear_restricted_context;
PL/SQL procedure successfully completed.

SQL> select * from global_context;
no rows selected

SQL> drop context sec_context;
Context dropped.

SQL>

```

### *USERNAME AND CLIENT IDENTIFIER WITH GLOBAL CONTEXT*

One of the major reasons for using global contexts is in proxied and connection pooled environments. The global context allows the application developer a mechanism for setting up the context in one session and allowing visibility of the context across sessions and connections, where session/local contexts do not provide this functionality. As previously shown, when no username is specified on the `set_context` (and `clear_context`) command, the context values are visible to any session. If username is specified, then the visibility of the context values is restricted to only sessions connected with the same `session_user` (`current_user` does not apply here, it is based on `session_user`). Another mechanism for restricting visibility of global contexts is via use of the `client_identifier`. If the `client_identifier` is specified when `set_context` is used, then it must be set to the identical value to access information in the context. Both these mechanisms allow visibility of specific global context values in a controlled manner across sessions.

For instance, if, in a web environment, there are multiple shared database connections across the web application servers, with hundreds of users, each user is able to have a semi-private global application context. It is the responsibility of the application to ensure the `client_identifier` is set to specify the actual web application-level username. Once the `client_identifier` is set appropriately, any of the shared connections have access to that specific client user's context information. Depending on the application requirements, either or both username and `client_identifier` may be appropriate.

In a nutshell, the rules for global context visibility in relation to `client_identifier` and username are:

- If `client_identifier` is set for the session, it must match `client_identifier` in the global context
- If username is set in the global context it must match the `session_user`

The following sql\*session sets up a global context with a context package that aids in showing the client\_identifier sensitivity of global contexts.

```

SQL> conn user1/*****@test_db
Connected.
SQL>
SQL> create or replace context sec_context using sec_context_pkg
  2  accessed globally;

Context created.

SQL> create or replace package sec_context_pkg is
  2  procedure set_restricted_context;
  3  procedure clear_restricted_context;
  4  end;
  5  /

Package created.

SQL> create or replace package body sec_context_pkg is
  2  procedure set_restricted_context is begin
  3  dbms_session.set_context(
  4  'sec_context', 'cost',
  5  'RESTRICTED-' ||
  6  nvl( sys_context('userenv','client_identifier'),'NULL'),
  7  null,
  8  sys_context('userenv','client_identifier')
  9  );
 10  end;
 11  procedure clear_restricted_context is begin
 12  dbms_session.clear_context(
 13  'sec_context',
 14  sys_context('userenv','client_identifier'),
 15  'cost');
 16  end;
 17  end;
 18  /

Package body created.

SQL> select sys_context('sec_context','cost') cost_sec from dual;

COST_SEC
-----

SQL> exec sec_context_pkg.set_restricted_context;

PL/SQL procedure successfully completed.

SQL> select sys_context('sec_context','cost') cost_sec from dual;

COST_SEC
-----
RESTRICTED-NULL

SQL> exec dbms_session.set_identifier('CLASS_A');

PL/SQL procedure successfully completed.

SQL> select sys_context('sec_context','cost') cost_sec from dual;

COST_SEC
-----

```

```

SQL> exec sec_context_pkg.set_restricted_context;
PL/SQL procedure successfully completed.
SQL> select sys_context('sec_context','cost') cost_sec from dual;
COST_SEC
-----
RESTRICTED-CLASS_A
SQL>
SQL>

```

With the exact same global context already set up and loaded with values, we can access it with a different sql\*plus session as demonstrated in this script.

```

SQL> conn user2/*****@test_db
Connected.
SQL>
SQL> select sys_context('sec_context','cost') cost_sec from dual;
COST_SEC
-----
RESTRICTED-NULL
SQL> exec dbms_session.set_identifier('CLASS_A');
PL/SQL procedure successfully completed.
SQL> select sys_context('sec_context','cost') cost_sec from dual;
COST_SEC
-----
RESTRICTED-CLASS_A
SQL> exec dbms_session.set_identifier('CLASS_B');
PL/SQL procedure successfully completed.
SQL> select sys_context('sec_context','cost') cost_sec from dual;
COST_SEC
-----
SQL>

```

## **ORACLE VIRTUAL PRIVATE DATABASE**

Oracle Virtual Private Database, also known as fine-grained access control, provides a mechanism where Oracle supplies additional query criteria (dynamic predicates) behind the scenes based on rules defined by the application development team. Since these rules are implemented inside the database, they cannot be bypassed. These rules are often based on specific application contexts. Oracle also has provisions for allowing driving application context(s) which allows some control over which fine-grained access rules are currently applicable. In 9i these rules only allowed row-level data restrictions but starting in 10g the fine-grained access control features were extended to allow column-level security.

The application developers create policy functions which define the desired criteria to appropriately restrict data access. Once these functions are available, the dbms\_ols package is used to specify when the rules are applied and which policy functions to call when specific objects are accessed. Typically an application user environment is initialized with a login trigger. The login trigger would be responsible for initializing an application context(s) associated to the user. The policy functions would utilize the data in the application environment to apply the specific criteria based on the user's application-level authority.

Since the policies are enforced for all access to the objects, implementation of Virtual Private Database is extremely reliable for restricting access per the defined rules. There is a system privilege, EXEMPT ACCESS POLICY, which prevents enforcement of policies. No policies are enforced for the SYS user or users with this system privilege. This privilege should be used sparingly, primarily to allow database backup processes and other sensitive administrative access.

### SETTING THE STAGE

The remainder of this paper revolves around a hypothetical, very simple, inventory/purchasing environment. In this environment, there are different categories of inventory. Depending on the user authority, they can purchase/view all or some of the inventory. The inventory has cost information, and again, depending on the user authority, they can see some or all of the cost data.

### DEFINING ITEMS

For purposes of keeping this simple, there are only three inventory items in two categories. The following sql\*plus session sets up our inventory data.

```
SQL> conn user1/*****@test_db
Connected.
SQL> create table items(
  2     item      varchar2(10),
  3     category  varchar2(10),
  4     cost      number(6,2) );

Table created.

SQL> insert into items
  2 select 'WRENCH','TOOL',          9.34 from dual union all
  3 select 'TABLE', 'FURNITURE',    138.00 from dual union all
  4 select 'COUCH', 'FURNITURE',    1299.99 from dual;

3 rows created.

SQL> commit;

Commit complete.

SQL> select * from items;

ITEM          CATEGORY          COST
-----
WRENCH        TOOL                9.34
TABLE         FURNITURE           138
COUCH         FURNITURE          1299.99

SQL>
```

### DEFINING SECURITY CONTEXT

In this example, we will have a security table which defines application-level securities based on the session\_user. There are two security areas, item categories and cost categories. These are stored independently for each user authorized to use the application. A custom application context is setup to provide visibility of the settings in the table to the policy functions. It is possible to allow the policy functions to read this data dynamically, but using a context allows some performance benefits both in determining if policy functions need re-execution (Oracle will recognize if an application context is static between policy function calls for example) and in the higher-performance access of context data over table data.

The actual user access authority is stored in a table, but a login trigger would execute the context security package setting up the associated custom application context. The actual policy enforcement functions will access the application context.

The following sql\*plus session sets up the security table as well as the application context and associated package.

```
SQL> conn user1/*****@test_db
Connected.
SQL>
SQL> create table security_users(
```

```

2      username   varchar2(30),
3      cost_class varchar2(10),
4      item_class  varchar2(10) );

```

Table created.

```

SQL> insert into security_users
2  select 'USER1', 'FULL', 'FULL' from dual union all
3  select 'USER2', 'TOOL', 'FULL' from dual;

```

2 rows created.

```
SQL> Commit;
```

Commit complete.

```
SQL> create context sec_context using sec_context_pkg;
```

Context created.

```

SQL> create or replace package sec_context_pkg is
2  procedure set_restricted_context;
3  end;
4  /

```

Package created.

```

SQL> create or replace package body sec_context_pkg is
2  procedure set_restricted_context is
3  sec_dat security_users%rowtype;
4  Begin
5  -- retrieve the session user's security information
6  select * into sec_dat from security_users
7  where username = sys_context('userenv','session_user');
8  -- set their appropriate application context
9  dbms_session.set_context('sec_context','cost',sec_dat.cost_class);
10 dbms_session.set_context('sec_context','item',sec_dat.item_class);
11 exception when no_data_found then
12 -- no user security information found, set no access defaults
13 dbms_session.set_context('sec_context','cost','NONE');
14 dbms_session.set_context('sec_context','item','NONE');
15 end;
16 end;
17 /

```

Package body created.

```
SQL>
```

The following sql\*plus session demonstrates use of this new security context

```
SQL> delete from security_users where username=user;
```

1 row deleted.

```
SQL> exec sec_context_pkg.set_restricted_context;
```

PL/SQL procedure successfully completed.

```
SQL> select sys_context('sec_context','cost') cost_security from dual;
```

```
COST_SECURITY
```

```
-----
NONE
```

```
SQL> insert into security_users values(user,'FULL','FULL');
```

```

1 row created.

SQL> exec sec_context_pkg.set_restricted_context;

PL/SQL procedure successfully completed.

SQL> select sys_context('sec_context','cost') cost_security from dual;

COST_SECURITY
-----
FULL

SQL>

```

As mentioned, the packaged procedure, `sec_context_pkg.set_restricted_context` would typically be executed using a login trigger.

### CREATING A POLICY FUNCTION

All policy functions are basically the same, they take an object owner and object name as parameters and they are responsible for providing a dynamic predicate (where clause criteria) which Oracle then applies to restrict access to the secured object. Policy functions can be either stand-alone or contained in a package. The dynamic predicate can be extraordinarily complicated including subqueries and other query mechanics as requirements dictate. The criteria can include tables and views not normally accessible to the end user as the permissions resolution for the additional criteria is based on the permissions of the policy function owner.

When a given function is created, it should be decided if the returned predicate is static or dynamic, and if it is dynamic, can it return different values even if the application contexts associated to the session have remained static. The static/dynamic nature of the function can affect parameters when the actual policies are defined using `dbms_ols`.

Ultimately, our example will become more complicated and include costing and commodity functions, but it will start out as a fairly simple row-level restriction based on commodity. We will be creating a single policy function contained in a policy package. Two versions will be created, a dynamic predicate and a static predicate version.

The first version of the commodity row-level security policy function is a dynamic policy. Depending on the commodity security of the user, different predicate values will be generated by the policy function. The following sql\*plus session shows creation of the function and demonstrates that it can return different values (making it dynamic by definition).

```

SQL> create or replace package sec_policy_pkg is
  2   function category_security (
  3       o_schema in varchar2, o_name in varchar2 ) return varchar2;
  4   end sec_policy_pkg;
  5   /

Package created.

SQL> create or replace package body sec_policy_pkg is
  2   function category_security (
  3       o_schema in varchar2, o_name varchar2 ) return varchar2 as
  4   begin
  5       if sys_context('SEC_CONTEXT', 'ITEM') <> 'FULL' then
  6           return('category=sys_context(''SEC_CONTEXT'', ''ITEM'')');
  7       else
  8           return(null);
  9       end if;
 10   end category_security;
 11   end sec_policy_pkg;
 12   /

Package body created.

SQL> select sec_policy_pkg.category_security(null,null) predicate from dual;

PREDICATE
-----

```

```

SQL> update security_users set item_class='TOOL' where username=user;
1 row updated.

SQL> exec sec_context_pkg.set_restricted_context;
PL/SQL procedure successfully completed.

SQL> select sec_policy_pkg.category_security(null,null) predicate from dual;
PREDICATE
-----
category=sys_context('SEC_CONTEXT','ITEM')

SQL>

```

Alternatively, the item category row-level security function can be re-written to always return the same predicate no matter what the user's current security status. This static alternative is shown here and is the version we will settle on for ongoing use in our example.

```

SQL> create or replace package body sec_policy_pkg is
  2  function category_security (
  3    o_schema in varchar2, o_name in varchar2 ) return varchar2 as
  4    predicate varchar2(100);
  5  BEGIN
  6    predicate := 'sys_context(''SEC_CONTEXT'', ''ITEM'')='''FULL'' OR '
  7              || 'category=sys_context(''SEC_CONTEXT'', ''ITEM'')';
  8    return(predicate);
  9  end category_security;
 10 end sec_policy_pkg;
 11 /

Package body created.

SQL> select sec_policy_pkg.category_security(null,null) predicate from dual;
PREDICATE
-----
sys_context('SEC_CONTEXT','ITEM')='FULL' OR category=sys_context(
'SEC_CONTEXT','ITEM')

SQL>

```

## INTRODUCING THE DBMS\_RLS PACKAGE

Oracle provides the DBMS\_RLS package as an interface for managing the fine-grained security policies used to implement virtual database. There are procedures to manage independent securities, create security groups for an object, and apply policies to the security groups. Associated with security group management is the ability to create a driving context to enable selective application of policies.

## IMPLEMENTING ROW-LEVEL SECURITY POLICY

The actual concept of Virtual Database applies primarily to row-level securities. These securities allow you to define a selective subset of visible data for individual sessions. For instance, you could define a full set of data, and have a virtual customer database which only allows visibility of each customer's own data... 'Virtually' creating another sub-database.

In our example, the data subset for row-level security will apply to item category. We will implement a security policy which applies our category\_security predicate to any select access on the items table. Since our current function is static (always returns the exact same predicate string), we will create it as a static definition. The policy creation statement allows specification of a table, view, or synonym to a specific schema and the policy function is also defined by schema. The default applies to the current schema, which is what we will be using in the samples.

Based on the scripts so far, our current user should only have access to the TOOL item category, but since we have no rules yet applied, all data is visible. The following sql\*plus session demonstrates this full access, applies a new security policy using our category\_security policy function, then shows the policy is in effect. After this, our current security is modified to FULL, demonstrating access is now available to all items.

```
SQL> select count(*) from items;

COUNT(*)
-----
          3

SQL> BEGIN DBMS_RLS.ADD_POLICY(
2  object_name => 'items',
3  policy_name => 'category_sec',
4  policy_function => 'sec_policy_pkg.category_security',
5  statement_types => 'select,insert,update,delete',
6  policy_type => dbms_rls.SHARED_STATIC);
7  END;
8  /

PL/SQL procedure successfully completed.

SQL> select count(*) from items;

COUNT(*)
-----
          1

SQL> update security_users set item_class='FULL' where username=user;

1 row updated.

SQL> exec sec_context_pkg.set_restricted_context;

PL/SQL procedure successfully completed.

SQL> select items.*,sys_context('SEC_CONTEXT','ITEM')sec_Item from items;

ITEM          CATEGORY          COST SEC_ITEM
-----
WRENCH        TOOL          9.34 FULL
TABLE         FURNITURE          138 FULL
COUCH         FURNITURE    1299.99 FULL

SQL>
```

## IMPLEMENTING COLUMN-LEVEL SECURITY POLICY

Starting in 10g, fine-grained access control was extended to allow column-level security. The definition of policies allows specification of specific columns to be secured. With column security, the policy is only enforced if the secured columns are included in the access request. The default behavior of column-level securities is identical to row-level security, the policy function predicates are applied, but only if access is attempted to the secured columns.

As an example, a column security function could be enabled on an employee table making it available unfiltered as long as no attempt was made to access birth date or salary information. If those columns were accessed, the table would be filtered for normal employees to only their own record, and for managers only to direct reports.

Column-level security has an optional implementation where the policy does not actually limit individual rows, but rather nulls out the secured column values for filtered rows and shows the actual column data for rows meeting the dynamic predicate values. With this implementation, all rows would be available for the hypothetical employee table, but the birth date and salary would be null except for your own employee record or manager's direct reports.

Policy functions are identical for row-level and column-level securities. To implement column-level security instead of row-level security when defining an actual policy with dbms\_rls, include a comma-separated list of the columns passed with the

sec\_relevant\_cols parameter. The default behavior of the policy restricts rows. To implement the optional column-nulling feature, the sec\_relevant\_cols\_opt parameter is set to the value dbms\_ols.ALL\_ROWS.

### *COLUMN-LEVEL SECURITY POLICY FUNCTION*

We will be extending the prior security policy package example to include another policy function associated with the cost data column. This new function will be created as a dynamic predicate that can return different values based on the user's application context values.

```
SQL> create or replace package sec_policy_pkg is
  2   function category_security (
  3     o_schema in varchar2, o_name in varchar2 ) return varchar2;
  4   function cost_security (
  5     o_schema in varchar2, o_name in varchar2 ) return varchar2;
  6 end sec_policy_pkg;
  7 /

Package created.

SQL>
SQL> create or replace package body sec_policy_pkg is
  2   function category_security (
  3     o_schema in varchar2, o_name in varchar2 ) return varchar2 as
  4     predicate varchar2(100);
  5 BEGIN
  6   predicate := 'sys_context(''SEC_CONTEXT'', ''ITEM'')='''FULL'' OR '
  7             || 'category=sys_context(''SEC_CONTEXT'', ''ITEM'')';
  8   return(predicate);
  9 end category_security;
 10 function cost_security (
 11   o_schema in varchar2, o_name varchar2 ) return varchar2 as
 12 begin
 13   if sys_context('SEC_CONTEXT', 'COST') <> 'FULL' then
 14     return('category=sys_context(''SEC_CONTEXT'', ''COST'')');
 15   else
 16     return(null);
 17   end if;
 18 end cost_security;
 19 end sec_policy_pkg;
 20 /

Package body created.

SQL>
```

### *COLUMN-LEVEL SECURITY BY ROW FILTERING*

The following sql\*plus session demonstrates implementation of our cost\_security function in a column-level security on the cost column. This implementation restricts rows based on category if the cost column is accessed. Note that these policies hold true even if the access is attempted through a view.

```
SQL> update security_users set cost_class='FULL', item_class='FULL'
  2   where username=user;

1 row updated.

SQL> exec sec_context_pkg.set_restricted_context;

PL/SQL procedure successfully completed.

SQL> exec DBMS_OLS.ADD_POLICY( object_name => 'items',           -
>                             policy_name => 'cost_sec',       -
>                             policy_function => 'sec_policy_pkg.cost_security', -
>                             statement_types => 'select,insert,update,delete', -
>                             policy_type => dbms_ols.SHARED_CONTEXT_SENSITIVE, -
>                             sec_relevant_cols => 'COST');
```

```

PL/SQL procedure successfully completed.

SQL> select * from items;

ITEM          CATEGORY          COST
-----
WRENCH        TOOL                9.34
TABLE         FURNITURE           138
COUCH         FURNITURE          1299.99

SQL> update security_users set cost_class='TOOL' where username=user;

1 row updated.

SQL> exec sec_context_pkg.set_restricted_context;

PL/SQL procedure successfully completed.

SQL> select * from items;

ITEM          CATEGORY          COST
-----
WRENCH        TOOL              9.34

SQL> select item,category from items;

ITEM          CATEGORY
-----
WRENCH        TOOL
TABLE         FURNITURE
COUCH         FURNITURE

SQL> create or replace view cost_view as select * from items;

View created.

SQL> select * from cost_view;

ITEM          CATEGORY          COST
-----
WRENCH        TOOL              9.34

SQL> drop view cost_view;

View dropped.

SQL>

```

### *DROPPING A FINE-GRAINED ACCESS SECURITY POLICY*

We are going to replace the row-filtering column-level security with a new policy which nulls out the column values which should not be seen rather than filtering rows. Before that can be installed, we need to drop the unwanted policy. The following sql\*plus session demonstrates dropping the unnecessary policy

```

SQL> exec dbms_rls.drop_policy(user,'items','cost_sec');

PL/SQL procedure successfully completed.

SQL>

```

### COLUMN-LEVEL SECURITY BY VALUE NULLING

Implementing column security by nulling values in unauthorized rows is almost identical to the row restriction method with the exception of the added policy parameter `sec_relevant_cols_opt` being set to `dbms_ols.ALL_ROWS`. The following sql\*plus session example demonstrates this:

```
SQL> exec dbms_ols.add_policy(object_name=>'items',      -
>      policy_name=>'cost_sec',      -
>      policy_function=>'sec_policy_pkg.cost_security', -
>      sec_relevant_cols=>'cost',      -
>      sec_relevant_cols_opt=>dbms_ols.ALL_ROWS);

PL/SQL procedure successfully completed.

SQL>
SQL> select items.*,sys_context('SEC_CONTEXT','COST')sec_Cost from items;

ITEM          CATEGORY          COST SEC_COST
-----
WRENCH        TOOL                9.34 TOOL
TABLE         FURNITURE              TOOL
COUCH         FURNITURE              TOOL

SQL> update security_users set cost_class='FURNITURE' where username=user;

1 row updated.

SQL> exec sec_context_pkg.set_restricted_context;

PL/SQL procedure successfully completed.

SQL> select items.*,sys_context('SEC_CONTEXT','COST')sec_Cost from items;

ITEM          CATEGORY          COST SEC_COST
-----
WRENCH        TOOL                FURNITURE
TABLE         FURNITURE          138 FURNITURE
COUCH         FURNITURE        1299.99 FURNITURE

SQL> update security_users set cost_class='FULL' where username=user;

1 row updated.

SQL> exec sec_context_pkg.set_restricted_context;

PL/SQL procedure successfully completed.

SQL> select items.*,sys_context('SEC_CONTEXT','COST')sec_Cost from items;

ITEM          CATEGORY          COST SEC_COST
-----
WRENCH        TOOL                9.34 FULL
TABLE         FURNITURE           138 FULL
COUCH         FURNITURE          1299.99 FULL

SQL>
```

### POLICY GROUPS

What if we actually have two applications and each requires different policies? For instance, what if we don't want to have the row-level category restrictions in place at the same time or for the same users as the column-level cost restrictions? This can be implemented with policy groups in conjunction with driving contexts. You can also have a combination of policies, where some are always in force and others are optionally in force based on driving context.

### OVERHEAD SETUP FOR POLICY GROUPS EXAMPLE

Our existing example will be extended to include driving context and optional policies with groups. For purposes of this example, we will continue to have items in categories, but now for two completely separate companies. Each company can have an independent set of items and neither is allowed visibility of the other's items. There are two standard applications shared by these companies, a purchasing application, which will utilize the row level category restrictions, and an inventory application, which will use the row-level item category policy restriction.

#### CREATING THE DATA

To start this new example, we need to replace our existing items table with a new items table with an added company column. Note that when we drop the existing items table, all policies against that table are also dropped.

```
SQL> drop table items;

Table dropped.

SQL> create table items(
  2     company  number(2),
  3     item     varchar2(10),
  4     category varchar2(10),
  5*    cost     number(6,2) );

Table created.

SQL> insert into items
  2  select 1, 'WRENCH', 'TOOL',          9.34 from dual union all
  3  select 1, 'TABLE', 'FURNITURE',    138.00 from dual union all
  4  select 1, 'COUCH', 'FURNITURE',    1299.99 from dual union all
  5  select 2, 'STOVE', 'APPLIANCE',    687.99 from dual union all
  6  select 2, 'DRYER', 'APPLIANCE',    399.00 from dual union all
  7* select 2, 'SHIRT', 'CLOTHING',     12.12 from dual;

6 rows created.

SQL> commit;

Commit complete.

SQL> select * from items;

  COMPANY ITEM          CATEGORY          COST
-----
         1 WRENCH        TOOL              9.34
         1 TABLE        FURNITURE         138
         1 COUCH         FURNITURE       1299.99
         2 STOVE         APPLIANCE        687.99
         2 DRYER         APPLIANCE         399
         2 SHIRT         CLOTHING         12.12

6 rows selected.
SQL>
```

#### CREATING THE APPLICATION CONTEXT AND SECURITY POLICY PACKAGE

For the extended example, all the policy functions will be held in one security package. This package has functions to control all aspects of the environment, setting which company is active, controlling the security context, as well as the policy enforcement functions returning dynamic predicates for row and column-level security implementation. For this example, the application context will be a local session context.

The following sql\*plus session defines the application context and the complete security package:

```
SQL> CREATE OR REPLACE CONTEXT apps_ctx USING apps_security_context_pkg;

Context created.
```

```

SQL> CREATE OR REPLACE PACKAGE apps_security_context_pkg IS
  2   PROCEDURE set_current_app (policy_group VARCHAR2);
  3   PROCEDURE set_company (company VARCHAR2);
  4   PROCEDURE set_category(category VARCHAR2);
  5   FUNCTION by_company (sch varchar2, tab varchar2) RETURN VARCHAR2;
  6   FUNCTION by_category(sch varchar2, tab varchar2) RETURN VARCHAR2;
  7   FUNCTION price_sensitive (sch varchar2, tab varchar2) RETURN VARCHAR2;
  8 END apps_security_context_pkg;
  9 /

```

Package created.

```

SQL>
SQL> CREATE OR REPLACE PACKAGE BODY apps_security_context_pkg AS
  2   PROCEDURE set_current_app ( policy_group varchar2 ) IS
  3   BEGIN
  4     DBMS_SESSION.SET_CONTEXT('apps_ctx','ACTIVE_APP', upper(policy_group));
  5   END set_current_app;
  6
  7   PROCEDURE set_company (company varchar2 ) IS
  8   BEGIN
  9     DBMS_SESSION.SET_CONTEXT('apps_ctx','COMPANY', company);
 10   END set_company;
 11
 12   PROCEDURE set_category(category varchar2 ) IS
 13   BEGIN
 14     DBMS_SESSION.SET_CONTEXT('apps_ctx','CATEGORY', category);
 15   END set_category;
 16
 17   FUNCTION by_company (sch varchar2, tab varchar2) RETURN VARCHAR2 IS
 18   BEGIN
 19     RETURN 'COMPANY = SYS_CONTEXT(''apps_ctx'', 'COMPANY' )';
 20   END by_company;
 21
 22   FUNCTION by_category (sch varchar2, tab varchar2) RETURN VARCHAR2 IS
 23   BEGIN
 24     RETURN 'CATEGORY = SYS_CONTEXT(''apps_ctx'', 'CATEGORY' )';
 25   END by_category;
 26
 27   FUNCTION price_sensitive (sch varchar2, tab varchar2) RETURN VARCHAR2 IS
 28   BEGIN
 29     RETURN '0=1'; -- no data will be visible
 30   END price_sensitive;
 31
 32 END apps_security_context_pkg;
 33 /

```

Package body created.

SQL>

### *SYS\_DEFAULT POLICY GROUP*

For this example, there is an overriding policy to restrict access to the items table for everyone by company. This can be implemented with a stand-alone policy, or with a group policy assigned to the SYS\_DEFAULT policy group. Either method is always enforced, even when driving policies are in affect for other policy groups.

To enforce this policy, the apps\_security\_context\_pkg.by\_company policy function will be used to generate the rule as a dynamically assigned static predicate. The apps\_security\_context\_pkg.set\_company function is used to set the session context variable. This must be set to a company with valid data to gain access to any of the data in the items table.

Once this policy is established, we have implemented Virtual Database by Company. One real items table, but it virtually becomes separate sets of data based on the company assigned to a user.

The following sql\*plus session demonstrates implementation and use of our 'security\_by\_company' restriction policy on the items table:

```
SQL> BEGIN
 2  DBMS_RLS.ADD_GROUPED_POLICY(user,'items','SYS_DEFAULT',
 3  'security_by_company',user,'apps_security_context_pkg.by_company');
 4  END;
 5  /
```

PL/SQL procedure successfully completed.

```
SQL> select * from items;
```

no rows selected

```
SQL> exec apps_security_context_pkg.set_company(1);
```

PL/SQL procedure successfully completed.

```
SQL> select * from items;
```

| COMPANY | ITEM   | CATEGORY  | COST    |
|---------|--------|-----------|---------|
| 1       | WRENCH | TOOL      | 9.34    |
| 1       | TABLE  | FURNITURE | 138     |
| 1       | COUCH  | FURNITURE | 1299.99 |

```
SQL> exec apps_security_context_pkg.set_company(2);
```

PL/SQL procedure successfully completed.

```
SQL> select * from items;
```

| COMPANY | ITEM  | CATEGORY  | COST   |
|---------|-------|-----------|--------|
| 2       | STOVE | APPLIANCE | 687.99 |
| 2       | DRYER | APPLIANCE | 399    |
| 2       | SHIRT | CLOTHING  | 12.12  |

```
SQL>
```

### CREATING NEW POLICY GROUPS AND ASSOCIATED POLICIES

We are assuming there are two distinct applications using our items table. The inventory application, which restricts visibility of all cost data and the purchasing application, which should have visibility of the cost data, but restricts access by item category based on the user's role. They are distinct security policies for each application and distinct policy groups are created for each.

The following sql\*plus session demonstrates creation of the inventory application's policy group and associated policy to restrict all pricing information. Also is demonstrated that the pricing data is no longer visible. Note that the company restriction is also still in place.

```
SQL> BEGIN
 2  DBMS_RLS.CREATE_POLICY_GROUP(user,'items','inventory_app');
 3  END;
 4  /
```

PL/SQL procedure successfully completed.

```
SQL> BEGIN
 2  DBMS_RLS.ADD_GROUPED_POLICY(object_schema=>user, object_name=>'items',
 3  policy_group=>'inventory_app',
 4  policy_name=>'secure_inventory_prices',
 5  function_schema=>user,
 6  policy_function=>'apps_security_context_pkg.price_sensitive',
 7  sec_relevant_cols=>'cost',
 8  sec_relevant_cols_opt=>dbms_rls.ALL_ROWS);
```

```

 9  END;
10  /

PL/SQL procedure successfully completed.

SQL> select * from items;

   COMPANY ITEM          CATEGORY          COST
-----
         2 STOVE          APPLIANCE
         2 DRYER          APPLIANCE
         2 SHIRT          CLOTHING

SQL>

```

Now that we have the inventory application group and policy in place for our items table, we can move on to creating the purchasing application policy group and associated policy to provide row-level security on the items table based on approved item category.

The following sql\*plus session demonstrates creating the purchasing application group and policy. Note the company and inventory application policy are also currently enforced.

```

SQL> BEGIN
 2  DBMS_RLS.CREATE_POLICY_GROUP(user,'items','purchasing_app');
 3  END;
 4  /

PL/SQL procedure successfully completed.

SQL> BEGIN
 2  DBMS_RLS.ADD_GROUPED_POLICY(user,'items','purchasing_app',
 3  'secure_purchasing_category',
 4  user, 'apps_security_context_pkg.by_category');
 5  END;
 6  /

PL/SQL procedure successfully completed.

SQL> select * from items;

no rows selected

SQL> exec apps_security_context_pkg.set_category('CLOTHING');

PL/SQL procedure successfully completed.

SQL> select * from items;

   COMPANY ITEM          CATEGORY          COST
-----
         2 SHIRT          CLOTHING

SQL>

```

### *USING DRIVING CONTEXT TO SELECTIVELY ENFORCE POLICIES*

At this point in the example, there are three policies all in effect, the company restriction policy, the inventory application policy preventing visibility of cost data, and the purchasing application policy restricting rows by category. The final piece of this example requires definition of a driving context. The driving context is an application context value which holds the name of a policy group to be enforced. When no application context is set (as is the case at this point), all policies are enforced. Once a driving context is set, if a valid policy group name is stored in the application context variable, then only the named group policy is enforced. Multiple driving contexts can be established if the application required this enabling enforcement of multiple policy groups (and associated policies).

Note that once a driving context is established, and the value in the driving context application variable is set, if the name does not match an existing policy group for the object, Oracle will raise an exception. The policy group name in the driving application context variable must be in upper case or the name will not match. The function in this example `apps_security_context_pkg.set_company.set_current_app`, performs an `upper()` function on the supplied policy group. The driving context established for this example is `ACTIVE_APP` stored in the `apps_ctx` context.

The following sql\*plus session establishes the driving context. Initially the application context is left undefined so all policies are still enforced. Once the application is set to 'inventory\_app', only the `sys_default` group policy and the `inventory_app` group policy are enforced. The `purchasing_app` group policy is no longer enforced. Enabling the `inventory_app` provides visibility of all the company items, but no pricing is visible.

```
SQL> BEGIN
  2  DBMS_RLS.ADD_POLICY_CONTEXT(user,'items','apps_ctx','ACTIVE_APP');
  3  END;
  4  /
```

PL/SQL procedure successfully completed.

```
SQL> select * from items;
```

| COMPANY | ITEM  | CATEGORY | COST |
|---------|-------|----------|------|
| 2       | SHIRT | CLOTHING |      |

```
SQL> exec apps_security_context_pkg.set_current_app('inventory_app');
```

PL/SQL procedure successfully completed.

```
SQL> select sys_context('apps_ctx','ACTIVE_APP') from dual;
```

```
SYS_CONTEXT('APPS_CTX','ACTIVE
```

```
INVENTORY_APP
```

```
SQL> select * from items;
```

| COMPANY | ITEM  | CATEGORY  | COST |
|---------|-------|-----------|------|
| 2       | STOVE | APPLIANCE |      |
| 2       | DRYER | APPLIANCE |      |
| 2       | SHIRT | CLOTHING  |      |

```
SQL>
```

The next sql\*plus session changes the current application to the purchasing application. We should now have visibility of prices, but only for the currently authorized item category.

```
SQL> exec apps_security_context_pkg.set_current_app('purchasing_app');
```

PL/SQL procedure successfully completed.

```
SQL> select * from items;
```

| COMPANY | ITEM  | CATEGORY        | COST         |
|---------|-------|-----------------|--------------|
| 2       | SHIRT | <b>CLOTHING</b> | <b>12.12</b> |

```
SQL> exec apps_security_context_pkg.set_category('APPLIANCE');
```

PL/SQL procedure successfully completed.

```
SQL> select * from items;
```

| COMPANY | ITEM | CATEGORY | COST |
|---------|------|----------|------|
|---------|------|----------|------|

```

      2 STOVE          APPLIANCE      687.99
      2 DRYER          APPLIANCE       399
SQL>

```

The final sql\*plus session in our example demonstrates the oracle error when an unknown policy group is specified in the driving context variable.

```

SQL> exec apps_security_context_pkg.set_current_app('unknown_app');

PL/SQL procedure successfully completed.

SQL> select * from items;
select * from items
      *
ERROR at line 1:
ORA-28123: Driving context 'APPS_CTX,ACTIVE_APP' contains invalid group
'UNKNOWN_APP'

SQL>

```

## **FURTHER INFORMATION IN ORACLE'S DOCUMENTATION**

For further information and research, there is a significant amount of information in the online Oracle documentation related to application contexts, userenv context, sys\_context function, fine-grained security, and the dbms\_ols package.

This paper is primarily based on the information available in the Oracle 10g online documentation.

Some of the relevant references are:

### **Database Concepts Manual:**

Overview of Access Restrictions on Tables, Views, Synonyms, or Rows

( [http://download.oracle.com/docs/cd/B19306\\_01/server.102/b14220/security.htm#sthref2819](http://download.oracle.com/docs/cd/B19306_01/server.102/b14220/security.htm#sthref2819) )

### **The Oracle Database Security Guide:**

Using Virtual Private Database to Implement Application Security Policies

( [http://download.oracle.com/docs/cd/B19306\\_01/network.102/b14266/apdvpoli.htm](http://download.oracle.com/docs/cd/B19306_01/network.102/b14266/apdvpoli.htm) )

Implementing Application Context and Fine-Grained Access Control

( [http://download.oracle.com/docs/cd/B19306\\_01/network.102/b14266/apdvctx.htm](http://download.oracle.com/docs/cd/B19306_01/network.102/b14266/apdvctx.htm) )

### **The PL/SQL Packages and Types Reference:**

DBMS\_SESSION Package

( [http://download.oracle.com/docs/cd/B19306\\_01/appdev.102/b14258/d\\_sessio.htm#BHCCFGIF](http://download.oracle.com/docs/cd/B19306_01/appdev.102/b14258/d_sessio.htm#BHCCFGIF) )

DBMS\_RLS Package

( [http://download.oracle.com/docs/cd/B19306\\_01/appdev.102/b14258/d\\_ols.htm#i1000830](http://download.oracle.com/docs/cd/B19306_01/appdev.102/b14258/d_ols.htm#i1000830) )

### **The Database SQL Reference Manual:**

The SYS\_CONTEXT function

( [http://download.oracle.com/docs/cd/B19306\\_01/server.102/b14200/functions165.htm#i1038176](http://download.oracle.com/docs/cd/B19306_01/server.102/b14200/functions165.htm#i1038176) )

The CREATE CONTEXT statement

( [http://download.oracle.com/docs/cd/B19306\\_01/server.102/b14200/statements\\_5002.htm#i2060927](http://download.oracle.com/docs/cd/B19306_01/server.102/b14200/statements_5002.htm#i2060927) )