Unit 6: DB Performance Tuning

- Introduction
- Tuning methodology
- Tuning concepts
- AADM (Automatic Database Diagnostic Monitor)
- SQL Tuning Advisor
    - AWR Report
    - Virtual Private Database
    - Policy types, selective columns, column masking

Performance Tuning:

One of the biggest responsibilities of a DBA is to ensure that the Oracle database is tuned properly. The Oracle RDBMS is highly tunable and allows the database to be monitored and adjusted to increase its performance. One should do performance tuning for the following reasons:

- The speed of computing might be wasting valuable human time (users waiting for response);
- Enable your system to keep-up with the speed business isconducted; and
- Optimize hardware usage to save money (companies are spending millions on hardware).

As performance bottlenecks are identified during load testing and performance testing, these issues are commonly rectified through a process of performance tuning. Performance tuning can involve configuration changes to hardware, software and network components. A common bottleneck is the configuration of the application and database servers. Performance tuning can also include tuning SQL queries and tuning an applications underlying code to cater for concurrency and to improve efficiency. Performance tuning can result in hardware changes being made. This is a last resort; ideally tuning changes will result in a reduction of resource utilization.

Instance Tuning

When considering instance tuning, take care in the initial design of the database to avoid bottlenecks that could lead to performance problems. In addition, you must consider:

- Allocating memory to database structures
- Determining I/O requirements of different parts of the database
- Tuning the operating system for optimal performance of the database
- After the database instance has been installed and configured, you must monitor the database as it is running to check for performance-related problems.

Performance Principles

Performance tuning requires a different, although related, method to the initial configuration of a system. Configuring a system involves allocating resources in an ordered manner so that the initial system configuration is functional.Tuning is driven by identifying the most significant bottleneck and making the appropriate changes to reduce or eliminate the effect of that bottleneck. Usually, tuning is performed reactively, either while the system is in preproduction or after it is live.

Tuning Methodology:

For best results, tune during the design phase, rather than waiting to tune after implementing your system. You can optimize performance by system tuning. This is the process of making adjustments to the way in which various system resources are used so that they correspond more closely to the overall usage patterns of the system. You can improve the overall response time of the system by locating and removing system bottlenecks.

Two different tuning methods are:

- Proactive Tuning While Designing and Developing Systems
- Reactive Tuning to Improve Production Systems

Proactive Tuning:

The proactive tuning begins with the initial phase of design and development of system. It helps developer to design and develop a system that can perform well and as a result minimize the initialization and on- going administrative cost.

In proactive tuning, the problems are fixed before they occur or at least before they are widely noticed. The main goal of proactive tuning is to detect and repair problems before they affect the system operation. Proactive tuning usually occurs on a regularly scheduled interval, where several performance statistics are examined to identify whether the system behavior and resource usage has changed.

Proactive tuning is done by following given steps:

1. Tune the business rules.
2. Tune the data design
3. Tune the application design
4. Tune the logical structure of database
5. Tune database operations
6. Tune the access paths
7. Tune memory allocation
8. Tune I/O and physical structures
9. Tune resource contention
10. Tune the underlying platforms

After completing these steps, reassess your database performance, and decide whether further tuning is necessary.

Tune the business rules

For optimal performance, you may need to adapt business rules. These concern the high-level analysis and design of an entire system. Configuration issues are considered at this level, such as whether to use a multi-threaded server system-wide. In this way, the planners ensure that the performance requirements of the system correspond directly to concrete business needs.

Tune the data design

In the data design phase, you must determine what data is needed by your applications. You must consider what relations are important, and what their attributes are. Finally, you need to structure the information to best meet performance goals.

The database design process generally undergoes a normalization stage when data is

analyzed to eliminate data redundancy. With the exception of primary keys, any one data element should be stored only once in your database. After the data is normalized, however, you may need to denormalize it for performance reasons. You might decide that the database should retain frequently used summary values.
Another data design consideration is avoiding data contention.

Tune the application design

Business executives and application designers should translate business goals into an effective system design. Business processes concern a particular application within a system, or a particular part of an application.
At this level, you can also consider the configuration of individual processes.

Tune the logical structure of database

After the application and the system have been designed, you can plan the logical structure of the database. This primarily concerns fine-tuning the index design to ensure that the data is neither over- nor under-indexed. In the data design stage (Step 2), you determine the primary and foreign key indexes. In the logical structure design stage, you may create additional indexes to support the application.
Performance problems due to contention often involve inserts into the same block or incorrect use of sequence numbers. Use particular care in the design, use, and location of indexes, as well as in using the sequence generator and clusters.

Tune database operations

Before tuning the Oracle server, be certain that your application is taking full advantage of the SQL language and the Oracle features designed to enhance application processing. Use features and techniques such as the following, based on the needs of your application:

- Array processing
- The Oracle optimizer
- The row-level lock manager
- PL/SQL

Understanding Oracle's query processing mechanisms is also important for writing effective SQL statements.
Step 1: Find the Statements that Consume the Most Resources
Step 2: Tune These Statements To Use Fewer Resources

Tune the access paths

Ensure that there is efficient data access. Consider the use of clusters, hash clusters, B*-tree indexes, bitmap indexes, and optimizer hints. Also consider analyzing tables and using histograms to analyze columns in order to help the optimizer determine the best query plan. Ensuring efficient access may mean adding indexes or adding indexes for a particular application and then dropping them again. It may also mean re- analyzing your design after you have built the database. You may want to further normalize your data or create alternative indexes. Upon testing the application, you may find that you are still not obtaining the required response time. If this happens, then look for more ways to improve the design.

Tune memory allocation

Appropriate allocation of memory resources to Oracle memory structures can have a positive effect on performance.

Although you explicitly set the total amount of memory available in the shared pool, the oracle dynamically sets the size of each of the following structures contained within it:

- The data dictionary cache
- The library cache
- Context areas (if running a multi-threaded server)

You can explicitly set memory allocation for the following structures:

- Buffer cache
- Log buffer
- Sequence caches

Proper allocation of memory resources improves cache performance, reduces parsing of SQL statements, and reduces paging and swapping.

Process local areas include:

- Context areas (for systems not running a multi-threaded server)
- Sort areas
- Hash areas

Be careful not to allocate to the system global area (SGA) such a large percentage of the machine's physical memory that it causes paging or swapping.

Tune I/O and physical structures

Disk I/O tends to reduce the performance of many software applications. The Oracle server, however, is designed so that its performance is not unduly limited by I/O. Tuning I/O and physical structure involves these procedures:

- Distributing data so that I/O is distributed to avoid disk contention.
- Storing data in data blocks for best access: setting an adequate number of free lists and using proper values for PCTFREE and PCTUSED.
- Creating extents large enough for your data, to avoid dynamic extension of tables. This adversely affects the performance of high- volume OLTP applications.
- Evaluating the use of raw devices.

Tune resource contention

Concurrent processing by multiple Oracle users may create contention for Oracle resources. Contention may cause processes to wait until resources are available. Take care to reduce the following types of contention:

- Block contention
- Shared pool contention
- Lock contention
- Pinging (in a parallel server environment)
- Latch contention

Tune the underlying platforms

See your platform-specific Oracle documentation for ways to tune the underlying system. For example, on UNIX-based systems you might want to tune the following:

- Size of the UNIX buffer cache
- Logical volume managers
- Memory and size for each process

Reactive Tuning:

Reactive tuning is a response to a known or reported problem. You may begin tuning performance metrics in reaction to user complaints about

- response time,
- instance failures, or
- errors found in the alert log.

Reactive tuning is inevitably necessary sometimes but your goal should be proactive approach to system maintenance.

It is possible to reactively tune an existing production system. To take this approach, start at the bottom of the method and work your way up, finding and fixing any bottlenecks. A common goal is to make Oracle run faster on the given platform. You may find, however, that both the Oracle server and the operating system are working well. To get additional performance gains, you may need to tune the application or add resources. Only then can you take full advantage of the many features Oracle provides that can greatly improve performance when properly used in a well-designed system.

Even the performance of well-designed systems can degrade with use. Ongoing tuning is, therefore, an important part of proper system maintenance.

Tuning concepts: Understanding trade-offs:

Even with the application of a proven tuning methodology that utilizes extensive benchmarks, most tuning efforts involve some degree of compromise. This occurs because every Oracle server is constrained by the availability of three key resources: CPU, Disk (I/O), and memory.

CPU:

Tuning Oracle's memory and I/O activity will provide little benefit if the server's processor is already overburdened. However, even in high-end multi-CPU servers, considerations should be given to the impact that changing memory or device configurations will have on CPU utilization. Several Oracle Server Configuration parameter changes dynamically when CPUs are added or removed from the server.

Disk (I/O):

The more Oracle server activity occurs in memory, the lower the physical I/O will be. However, placing too many demands on the available memory by oversizing Oracle's memory structures can cause undesirable additional I/O in the form of Operating System paging and swapping. Modern disk- caching hardware and software also complicate database I/O tuning, since I/O activity on these systems may result in reads from the disk controller's cache and not directly from disk.

Memory:

The availability of memory for Oracle's memory structures is key to good performance. Managing that memory is important so that it is used to maximum benefit without wasting any that could be better used by other server processes. Oracle offers several memory tuning options that help you make the most efficient use of the available memory.

Common Tuning Problem Areas:

While examining your system for possible performance improvement, it is important to first examine those areas that are most likely the cause of poor performance.
These areas include:

- Poorly written application SQL
- Inefficient SQL execution plans
- Improperly sized System Global Area (SGA) memory structures
- Excessive file I/O
- Inordinate waits for access to database resources.

ADDM (Automatic Database Diagnosis Monitor)

For Oracle, the statistical data needed for accurate diagnosis of a problem is saved in the Automatic Workload Repository (AWR). The Automatic Database Diagnosis Monitor is the one that analyzes the AWR data on a regular basis and then locates the root causes of performance problems, provides recommendations for correcting any problems and identifies non- problem area of the system. Because AWR is a repository of historical performance data, ADDM can be used to analyze performance issues after the event, often saving time and resources reproducing a problem. The analysis is performed top down, first identifying symptoms and then refining them to reach the root cause of performance problem.
The goal of the analysis is to reduce a single throughput metric called DB time. DB time is the cumulative time spent by the database server in processing user requests. It includes wait time and CPU time of all non-idle user sessions.
ADDM doesn't target tuning of individual user response times; for that use tracing technique. By reducing DB time, the database server is able to support more user requests using the same resources, which increases throughput.
Some of the types of problem that ADDM considers includes are:

- **CPU bottlenecks-** Is the system CPU bound by Oracle or some other application?
- **Undersized Memory Structures**- Are the Oracle memory structures, such as the SGA, PGA, and buffer cache adequately sized?
- **I/O capacity issues**- Is the I/O subsystem performing as expected?
- **Concurrency Issues**- Are there buffer busy problem?
- **Database Configuration Issues**- Is there any evidence of incorrect sizing of log files, archiving issues, excessive checkpoints or sub- optimal parameter settings?

ADDM also documents the non-problem areas of the system. In addition to problem diagnostics, ADDM recommends possible solutions. When appropriate ADDM recommends multiple solutions for the DBA to choose from. ADDM considers variety of changes to a system while generating its recommendations that includes:

- Hardware Changes
- Database Configuration

- Schema changes
- Application changes
- Using other advisors

SQL Tuning Advisor

The automatic SQL Tuning capabilities are exposed through a server utility called the SQL Tuning Advisor that takes one or more SQL statements as an input and invokes the Automatic Tuning Optimizer to perform SQL tuning on the statement. The output of the SQL tuning advisor is in the form of advice or recommendations, along with a rationale for each recommendations and its expected benefit. The recommendation relates to collection of statistics on objects, creation of new indexes, restructuring of the statement, or creation of SQL profile. A user can choose to accept the recommendation to complete the tuning of SQL statements.

The SQL tuning advisor's input can be a single SQL statement or a set of statements. For tuning multiple statements, a SQL Tuning Set (STS) has to be first created. An STS is a database object that stores statements along with their execution context. It can be created manually using command line APIs or automatically using Oracle Enterprise Manager.

Input Sources:

The input for the SQL tuning advisor can come from several sources such as:

Automatic Database Diagnostic Monitor

It is the primary input source for SQL Tuning Advisor. By default, ADDM runs pro-actively once every hour and analyzes key statistics gathered by the AWR over last hour to identify any performance problem including High-load SQL statements. If high-load SQL statements are identified, ADDM recommends running SQL tuning advisor in SQL.

High-load SQL Statements

The AWR takes regular snapshots of system activities including high- load SQL statements ranked by relevant statistics, such as CPU consumption and wait time. A user can view the AWR and identify the high-load SQL of interest and run SQL tuning advisor on them.

Cursor Cache

This source is used for tuning recent SQL statements that are yet to be captured in the AWR. The cursor cache and AWR together provides the capability to identify and tune high-load SQL statements from the current time going as far back as the AWR retention allows, which is by default 7 days.

SQL Tuning Set

It is a user defined set of SQL statements that are yet to be deployed, with the goal of measuring their individual performance, or identifying the ones whose performance falls short of expectation. When a set of SQL statements are used as input, a SQL tuning set has to be first constructed and stored.

Tuning Options:

SQL tuning advisor provides options to manage the scope and duration of a tuning task. The scope of tuning task can be set to **limited** or **comprehensive.**

- For limited, the SQL tuning advisor produces recommendations based on the statistics checks, access path analysis, and SQL structure analysis. SQL profile recommendations are not generated.
- For comprehensive, the SQL tuning advisor carries out all the analysis it performs under limited scope plus SQL profiling. You can also specify a time limit for the tuning task, which by default is 30 minutes.

SQL Tuning Advisor Output:

After analyzing the SQL statements, the SQL Tuning Advisor provides advice on optimizing the execution plan, the rationale for the proposed optimization, the estimated performance benefit, and the command to implement the advice. You simply have to choose whether or not to accept the recommendations to optimize the SQL statements.

Running the SQL tuning advisor:

The recommended interface for running the SQL Tuning Advisor is the Oracle Enterprise Manager. Whenever possible, you should run the SQL Tuning Advisor using Oracle Enterprise Manager. If Oracle Enterprise Manager is unavailable, you can run the SQL Tuning Advisor using procedures in the DBMS_SQLTUNE package. To use the APIs, the user must be granted specific privileges.

Running SQL Tuning Advisor using DBMS_SQLTUNE package is a multi-step process:

1. Create a SQL Tuning Set (if tuning multiple SQL statements)
2. Create a SQL tuning task
3. Execute a SQL tuning task
4. Display the results of a SQL tuning task
5. Implement recommendations as appropriate

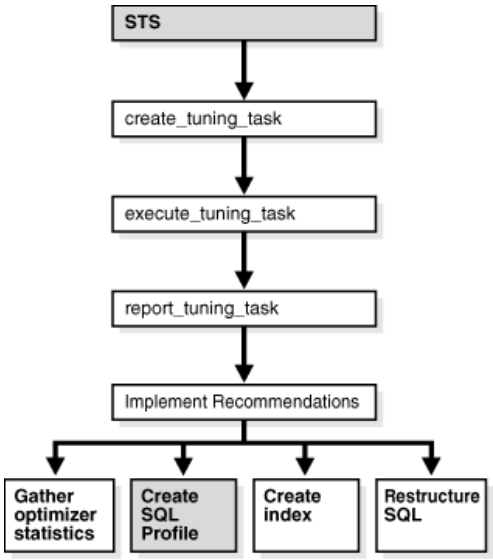

Fig: SQL Tuning Advisor APIs Creating a SQL tuning

task:

You can create tuning tasks from the text of a single SQL statement, a SQL Tuning Set containing multiple statements, a SQL statement selected by SQL identifier from the cursor cache, or a SQL statement selected by SQL identifier from the Automatic Workload Repository.

For example, to use the SQL Tuning Advisor to optimize a specified SQL statement text, you need to create a tuning task with the SQL statement passed as a CLOB argument. For the following PL/SQL code, the user HR has been granted the ADVISOR privilege and the function is run as user HR on the employees table in the HR schema.

```
DECLARE

my_task_name VARCHAR2(30); my_sqltext  CLOB;
BEGIN
my_sqltext := 'SELECT /*+ ORDERED */ * '
||

'FROM employees e, locations l, departments d ' ||
'WHERE e.department_id = d.department_id AND ' ||
'l.location_id = d.location_id
AND '   ||
'e.employee_id < :bnd';



my_task_name := DBMS_SQLTUNE.CREATE_TUNING_TASK( sql_text         => my_sqltext,
bind_list         => sql_binds(anydata.ConvertNumber(100)),
user_name      => 'HR',
scope   => 'COMPREHENSIVE',
time_limit       => 60,
task_name      => 'my_sql_tuning_task', description => 'Task to tune a query on a
specified employee'); END;
/
```

In this example, 100 is the value for bind variable :bnd passed as function argument of type SQL_BINDS, HR is the user under which the CREATE_TUNING_TASK function analyzes the SQL statement, the scope is set to COMPREHENSIVE which means that the advisor also performs SQL Profiling analysis, and 60 is the maximum time in seconds that the function can run. In addition, values for task name and description are provided.
The CREATE_TUNING_TASK function returns the task name that you have provided or generates a unique task name. You can use the task name to specify this task when using other APIs. To view the task names associated with a specific owner, you can run the following:

```
SELECT task_name FROM DBA_ADVISOR_LOG WHERE owner = 'HR';
```

Configuring a SQL Tuning Task

You can fine tune a SQL tuning task after it has been created by configuring its parameters using the SET_TUNING_TASK_PARAMETER procedure in the DBMS_SQLTUNE package:

```
BEGIN
DBMS_SQLTUNE.SET_TUNING_TASK_PARAMETER(
task_name => 'my_sql_tuning_task', parameter => 'TIME_LIMIT', value => 300);
END;
/
```

In this example, the maximum time that the SQL tuning task can run is changed to 300 seconds.

Executing a SQL Tuning Task

```
BEGIN
DBMS_SQLTUNE.EXECUTE_TUNING_TASK( task_name =>
'my_sql_tuning_task' ); END;
/
```

After you have created a tuning task, you need to execute the task and start the tuning process. For example:

Like any other SQL Tuning Advisor task, you can also execute the automatic tuning task SYS_AUTO_SQL_TUNING_TASK using the EXECUTE_TUNING_TASK API. The SQL Tuning Advisor will perform the same analysis and actions as it would when run automatically. You can also pass an execution name to the API to name the new execution.

Automatic Workload Repository (AWR):

The Automatic Workload Repository (AWR) collects, processes, and maintains performance statistics for problem detection and self-tuning purposes. This data is both in memory and stored in the database. The gathered data can be displayed in both reports and views.
The statistics collected and processed by AWR include:

- Wait events to identify performance problems
- Time model statistics indicating the amount of DB Time associated with a process from the V$SESS_TIME AND V$SYS_TIME_MODEL views.
- Active Session History (ASH) statistics from the V$ACTIVE_SESSION_HISTORY view.
- Some system and session statistics from the V$SYSSTAT and V$SESSTAT views.
- Object access and usage statistics.
- Resource intensive SQL statements.

The STATISTICS_LEVEL initialization parameter must be set to the TYPICAL or ALL to enable the Automatic Workload Repository. If the value is set to BASIC, you can manually capture AWR statistics using procedures in the DBMS_WORKLOAD_REPOSITORY package. However, because setting the STATISTICS_LEVEL parameter to BASIC turns off in- memory collection of many system statistics, such as segments statistics and memory advisor information, manually captured snapshots will not contain these statistics and will be incomplete.

Snapshots:

Snapshots are sets of historical data for specific time periods that are used for performance comparisons by ADDM. By default, AWR automatically generates snapshots of the performance data once every hour and retains the statistics in the workload repository for 7 days. You can also manually create snapshots, but this is usually not necessary. The data in the snapshot interval is then analyzed by the Automatic Database Diagnostic Monitor

(ADDM).
AWR compares the difference between snapshots to determine which SQL statements to capture based on the effect on the system load. This reduces the number of SQL statements that need to be captured over time.

Baselines:

A baseline is a pair of snapshots that represents a specific period of usage. Once baselines are defined they can be used to compare current performance against similar periods in the past. You may wish to create baseline to represent a period of batch processing.

```
BEGIN
  DBMS_WORKLOAD_REPOSITORY.create_baseline (
    start_snap_id => 210,
    end_snap_id   => 220,
    baseline_name => 'batch baseline');
END;
/
```

The pair of snapshots associated with a baseline are retained until the baseline is explicitly deleted.

```
BEGIN
DBMS_WORKLOAD_REPOSITORY.drop_baseline ( baseline_name => 'batch baseline',
cascade                                                => FALSE); -- Deletes associated snapshots if
TRUE.
END;
/
```

Baseline information can be queried from the DBA_HIST_BASELINE view.

Generating Automatic Workload Repository Reports

An AWR report shows data captured between two snapshots (or two points in time). The AWR reports are divided into multiple sections. The HTML report includes links that can be used to navigate quickly between sections. The content of the report contains the workload profile of the system for the selected range of snapshots.

The primary interface for generating AWR reports is Oracle Enterprise Manager. Whenever possible, you should generate AWR reports using Oracle Enterprise Manager. If Oracle Enterprise Manager is unavailable, you can generate AWR reports by running SQL scripts:

- The awrrpt.sql SQL script generates an HTML or text report that displays statistics for a range of snapshot Ids.
- The awrrpti.sql SQL script generates an HTML or text report that displays statistics for a range of snapshot Ids on a specified database and instance.
- The awrsqrpt.sql SQL script generates an HTML or text report that displays statistics of a particular SQL statement for a range of snapshot Ids. Run this report to inspect or debug the performance of a SQL statement.
- The awrsqrpi.sql SQL script generates an HTML or text report that displays statistics of a particular SQL statement for a range of snapshot Ids on a specified database and instance. Run this report to inspect or debug the performance of a SQL statement on a specific database and instance.
- The awrddrpt.sql SQL script generates an HTML or text report that compares detailed performance attributes and configuration settings between two selected time periods.
- The awrddrpi.sql SQL script generates an HTML or text report that compares detailed performance attributes and configuration settings between two selected time periods on a specific database and instance.

To run these scripts, you must be granted the DBA role.

Virtual Private Database:

Virtual Private Database (VPD) is a database security feature that is built into an Oracle database server, as opposed to being part of an application that is accessing the data. The user is only allowed to see the data they have been given permission to see. VPD enables you to create security policies to control database access at the row and column level. Essentially, Oracle VPD adds a dynamic WHERE clause to a SQL statement that is issued against the table, view, or synonym to which an Oracle Virtual Private Database security policy was applied.

You can apply Oracle VPD policies to SELECT, INSERT, UPDATE, INDEX, and DELETE statements.

For example:

Suppose a user performs following query: SELECT * FROM OE.ORDERS;

The Oracle VPD policy dynamically appends the statement with a WHERE clause as:

SELECT * FROM OE.ORDERS WHERE SALES_REP_ID=149;

In this example, the user can view orders by Sales Representative 149;

Oracle Virtual Private Database doesn't support filtering for DDLs such as TRUNCATE OR ALTER TABLE statements.

Benefits of Using Oracle Virtual Private Database:

Oracle VPD policy provides following benefits:

- Basing security policies on database objects rather than applications
- Controlling how Oracle database evaluates policy functions

Basing Security Policies on Database Objects rather than Applications

Attaching Oracle VPD policies to database tables, views, or synonyms rather than applications, provides following benefits:

Security

Associating a policy with a database table, view, or synonym can solve a potentially serious application security problem. Suppose a user is authorized to use an application, and then drawing on the privileges associated with that application wrongfully modifies the database by using an ad hoc query tool, such as SQL *Plus. By attaching security policies directly to tables, views or synonyms fine- grained access control ensures that the same security is in force, no matter how a user accesses the data.

Simplicity

You can add security policy to a table, view or synonym only once, rather than repeatedly adding it to each of your table-based, view- based or synonym-based applications.

Flexibility

You can have one security policy for SELECT statement, another for INSERT statement, and yet another for UPDATE AND DELETE statement.

Five Oracle VPD Policy Types: Dynamic Policy:

The Dynamic Policy type runs the policy functions each time a user accesses the VPD protected database objects. If you do not specify a policy type in the DBMS_RLS.ADD_POLICY procedure, then by default, your policy will be dynamic.

This policy type doesn't optimize database performance as the static and context sensitive policy type. However, Oracle recommends that before you set policies as either static or context sensitive, you should test them as DYNAMIC policy types, which runs every time which enables you to observe how the policy function affects each query, because nothing is cached.

Static Policy and Shared Static Policy:

The static policy type enforces the same predicate for all users in the instance. Oracle database stores policy predicates in SGA, so policy functions do not re-run for each query. This results in faster performance. You can enable static policy by setting the policy_type parameter of the DBMA_RLS.ADD_POLICY procedure to either STATIC or SHARED_STATIC, depending on whether or not you want the policy to be shared across multiple objects.

Static policies are ideal for environments where every query requires the same predicate and fast performance is essential, such as hosting environments. For these situations, when the policy function appends the same predicate to every query, rerunning the policy finction each time adds unnecessary overhead to the system.

Context-Sensitive Policy:

In contrast to static properties, context sensitive policies do not always cache the predicate. With context-sensitive policies, the database assumes that the predicate will change after statement parse time. But if there is no change in local application context, Oracle database does not return the policy function within the user session. If there was a change in context, then the database reruns the policy function to ensure that it captures any change to the predicate since the initial parsing.

Context-Sensitive policies are useful when different predicates should be applied depending on which user is executing the query.

Shared Context-Sensitive Policy:

This policy operates in the same was as regular context-sensitive policies, except they can be shared across multiple database objects. For this policy type, all objects can share the policy function from the UGA, where the predicate is cached until the local session context changes.

When using shared context-sensitive policies, ensure that the policy predicate doesn't contain attributes that are specified to a particular database object, such as a column name.

Policy Types	**When the Policy Function Executes**	**Usage Example**	**Shared Across Multiple Objects?**
DYNAMIC	Policy function re- executes every time a policy- protected database object is accessed.	Applications where policy predicates must be generated for each query, such as time-dependent policies where users are denied access to database objects at certain times during the day	No
STATIC	Once, then the predicate is cached in the SGA.[Foot 1]	View replacement	No
SHARED_STATIC	Same as STATIC	Hosting environments, such as data warehouses where the same predicate must be applied to multiple database objects	Yes
CONTEXT_SENSITIVE	At statement parse time At statement execution time when the local application context changed since the last use of the	Three-tier, session pooling applications where policies enforce two or more predicates for different users or groups	No

	cursor		
SHARED_CONTEXT_SENSITIVE	First time the object is reference in a database session. Predicates are cached in the private session memory UGA so policy functions can be shared among objects.	Same as CONTEXT_SENSITIVE, but multiple objects can share the policy function from the session UGA	Yes

Foot 1: Each execution of the same cursor could produce a different row set for the same predicate because the predicate may filter the data differently based on attributes such as SYS_CONTEXTor SYSDATE.

Controlling the display of Column data with policies:

You can create policies that enforce row-level security when a security- relevant column is referenced in a query.

- Adding Policies for Column-Level Oracle Virtual Private Database
- Displaying Only the Column Rows Relevant to the Query
- Using Column Masking to Display Sensitive Columns as NULL Values

Adding Policies for Column-Level Oracle Virtual Private Database

Column-level policies enforce row-level security when a query references a security-relevant column. You can apply a column-level Oracle Virtual Private Database policy to tables and views, but not to synonyms.

To apply the policy to a column, specify the security-relevant column by using the sec_relevant_cols parameter of the DBMS_RLS.ADD_POLICYprocedure. This parameter applies the security policy whenever the column is referenced, explicitly or implicitly, in a query.

For example, users who are not in a Human Resources department typically are allowed to view only their own Social Security numbers. A sales clerk initiates the following query:

```
SELECT fname, lname, ssn FROM emp;
```

The function implementing the security policy returns the predicate ssn='my_ssn'. Oracle Database rewrites the query and executes the following:

Example 1 shows a Oracle Virtual Private Database policy in which sales department users cannot see the salaries of people outside the department (department number 30) of the sales department users. The relevant columns for this policy are sal and comm. First, the Oracle Virtual Private Database policy function is created, and then it is added by using the DBMS_RLS PL/SQL package.

```
SELECT fname, lname, ssn FROM emp
WHERE ssn = 'my_ssn';
```

Example 1: Creating a Column-Level Oracle Virtual Private Database Policy

```
CREATE OR REPLACE FUNCTION hide_sal_comm ( v_schema IN VARCHAR2,
v_objname IN VARCHAR2)


RETURN VARCHAR2 AS con VARCHAR2 (200);

BEGIN
con := 'deptno=30'; RETURN (con);
END hide_sal_comm;
/
```

Then you configure the policy with the DBMS_RLS.ADD_POLICY procedure as follows:

```
BEGIN
 DBMS_RLS.ADD_POLICY (
  object_schema      => 'scott',
  object_name        => 'emp',
  policy_name        => 'hide_sal_policy',
  policy_function    => 'hide sal comm',
```

```
   sec_relevant_cols => 'sal,comm');
END;
/
```

Displaying Only the Column Rows Relevant to the Query

The default behavior for column-level Oracle Virtual Private Database is to restrict the number of rows returned for a query that references columns containing sensitive information. You specify these security-relevant columns by using the sec_relevant_columns parameterof the DBMS_RLS.ADD_POLICY procedure as shown in the Example 1.

For example, consider sales department users with the SELECT privilege on the emp table, which is protected with the column-level Oracle Virtual Private Database policy created in Example 1. The user (for example, user SCOTT) runs the following query:

```
SELECT ENAME, d.dname, JOB, SAL, COMM
FROM emp e, dept d
WHERE d.deptno = e.deptno;
```

The database returns the following rows:

ENAME COMM	DNAME	JOB	SAL
---------- ------	--------------	---------	---------- ----
ALLEN 300	SALES	SALESMAN	1600
WARD 500	SALES	SALESMAN	1250

```
2 rows selected.
```

The only rows that are displayed are those that the user has privileges to access all columns in the row.

Using Column Masking to Display Sensitive Columns as NULL Values

If a query references a sensitive column, then the default action of column- level Oracle Virtual Private Database restricts the number of rows returned.

In contrast to the default action of column-level Oracle Virtual Private Database, column-masking displays all rows, but returns sensitive column values as NULL. To include column-masking in your policy, set the sec_relevant_cols_opt parameter of the DBMS_RLS.ADD_POLICY procedure to dbms_rls.ALL_ROWS.

Example below shows column-level Oracle Virtual Private Database column-masking. It uses the same VPD policy as Example 1, but with sec_relevant_cols_opt specified as dbms_rls.ALL_ROWS.

Example: Adding a Column Masking to an Oracle Virtual Private Database Policy

```
BEGIN
 DBMS_RLS.ADD_POLICY(
   object_schema          => 'scott',
   object_name            => 'emp',
   policy_name            => 'hide_sal_policy',
   policy function        => 'hide sal comm',
```

```
   sec_relevant_cols     =>' sal,comm',
   sec_relevant_cols_opt => dbms_rls.ALL_ROWS);
END;
/
```

Assume that a sales department user with SELECT privilege on the emp table (such as user SCOTT) runs the following query:

```
SELECT ENAME, d.dname, job, sal, comm FROM emp e, dept d
WHERE d.deptno = e.deptno;
```

The database returns all rows specified in the query, but with certain values masked because of the Oracle Virtual Private Database policy:

ENAME COMM	DNAME	JOB	SAL
CLARK	ACCOUNTING	MANAGER	
KING	ACCOUNTING	PRESIDENT	
SCOTT	RESEARCH	ANALYST	
WARD 500	SALES	SALESMAN	1250
JAMES	SALES	CLERK	950
MARTIN 1400	SALES	SALESMAN	1250

```
6 rows selected.
```

The column-masking returned all rows requested by the sales user query, but made the sal and comm columns NULL for employees outside the sales department.
The following considerations apply to column-masking:

- Column-masking applies only to SELECT statements.
- Column-masking conditions generated by the policy function must be simple Boolean expressions, unlike regular Oracle Virtual Private Database predicates.
- For applications that perform calculations, or do not expect NULL values, use standard column-level Oracle Virtual Private Database, specifying sec_relevant_cols rather than the sec_relevant_cols_opt column-masking option.
- Column-masking used with UPDATE AS SELECT updates only the columns that users are allowed to see.
- For some queries, column-masking may prevent some rows from displaying. For example:

```
SELECT * FROM emp
WHERE sal = 10;
```

Because the column-masking option was set, this query may not return rows if the salary column returns a NULL value.