*Clear*SQL Code Review Rules

Use of the NULL statement

Package:	Severity:	Category:	Code Element:
Built-in	Trivial	Program Structure	NULL

Rationale:

Reasoning:

Null statements are mostly used as placeholders for code to be added at a later time.

Explanation:

Note that in this example the NULL statement is required, as the PL/SQL syntax does not allow an empty code block. Seeing such a construct suggests a code component is not yet complete. This makes readiness of code suspect, and requires additional effort to verify its correctness before testing or use.

Don't use GOTO

Package:	Severity:	Category:	Code Element:
Built-in	Major	Program Structure	GOTO

Rationale:

Reasoning:

GOTO is a dangerous construct which should be avoided wherever possible.

Explanation:

When GOTO statements are contained in the code, understanding and debugging the code becomes quite complex. This leads to higher development and maintenance costs. PL/SQL contains many structured constructs which can be used instead of GOTO statements (LOOP constructs, subroutine calls, CONTINUE, BREAK, EXIT WHEN). Sometimes a GOTO statement is used as a "quick and dirty" fix in a maintenance situation. General recommendations are to refactor code to avoid the use of GOTO. While this may be costlier then the quick & dirty GOTO, in the long run refactoring the code will lower the overall costs of maintaining the module.

An EXIT statement is used in a FOR loop

Package:	Severity:	Category:	Code Element:
Built-in	Trivial	Program Structure	Loop

Rationale:

Reasoning:

Ideally, a loop should have a single entry and a single exit. This is sometimes called the "one way in one way out" principle, and is a fundamental part of the edicts of structured programing.

When in doubt, always follow these rules, for the different looping structures:

- Do not use RETURN inside any kind of LOOP. This violates the “one way in one way out” principle.
- Always include an EXIT or EXIT WHEN in a simple LOOP to avoid an infinite loop.
- Do not use EXIT or EXIT WHEN in a simple LOOP in the middle of loop logic. Otherwise the loop is ended prematurely.
- Do not use EXIT from inside a WHILE loop. Always allow a WHILE loop to cycle through its intended number of iterations rather than leaving prematurely.
- Do not use EXIT from inside a FOR loop. Always allow the FOR loop to cycle through its intended number of iterations rather than leaving prematurely.
- Do not use EXIT from inside an IMPLICIT CURSOR loop. Always allow the loop to cycle through its intended number of iterations rather than leaving prematurely.
- Do not use EXIT from inside an EXPLICIT CURSOR loop. Always allow the loop to cycle through its intended number of iterations rather than leaving prematurely.

The bottom line is, with few exceptions, a loop of any kind should indicate how it will end by using the natural mechanism provided by the specific looping structure. And then it should actually end that way. Code is easier to read when it exhibits expected structures and behaviors.

An EXIT statement is used in a WHILE loop

Package:	Severity:	Category:	Code Element:
Built-in	Minor	Program Structure	Loop

Rationale:

Reasoning:

Ideally, a loop should have a single entry and a single exit. This is sometimes called the "one way in one way out" principle, and is a fundamental part of the edicts of structured programing.

When in doubt, always follow these rules, for the different looping structures:

- Do not use RETURN inside any kind of LOOP. This violates the “one way in one way out” principle.
- Always include an EXIT or EXIT WHEN in a simple LOOP to avoid an infinite loop.
- Do not use EXIT or EXIT WHEN in a simple LOOP in the middle of loop logic. Otherwise the loop is ended prematurely.
- Do not use EXIT from inside a WHILE loop. Always allow a WHILE loop to cycle through its intended number of iterations rather than leaving prematurely.
- Do not use EXIT from inside a FOR loop. Always allow the FOR loop to cycle through its intended number of iterations rather than leaving prematurely.
- Do not use EXIT from inside an IMPLICIT CURSOR loop. Always allow the loop to cycle through its intended number of iterations rather than leaving prematurely.

- Do not use EXIT from inside an EXPLICIT CURSOR loop. Always allow the loop to cycle through its intended number of iterations rather than leaving prematurely.

The bottom line is, with few exceptions, a loop of any kind should indicate how it will end by using the natural mechanism provided by the specific looping structure. And then it should actually end that way. Code is easier to read when it exhibits expected structures and behaviors.

A RETURN statement is used in a FOR loop

Package:	Severity:	Category:	Code Element:
Built-in	Major	Program Structure	Loop

Rationale:

Reasoning:

Ideally, a FOR loop should run to completion. In other words, the body of the FOR loop should be executed a fixed number of times, which is given by the range specified in the loop range clause.

Explanation:

If there is a RETURN statement inside the loop, then there is a second exit from the loop. This makes the loop harder to understand, to trace and to debug. The EXIT should occur after the loop. It may be beneficial to change the FOR loop to a WHILE loop and merge the return condition in the Boolean WHILE condition.

A RETURN statement is used in a WHILE loop

Package:	Severity:	Category:	Code Element:
Built-in	Major	Program Structure	Loop

Rationale:

Reasoning:

The RETURN statement completes the execution of a subprogram immediately.

Explanation:

There is a basic principle called 1-in/1-out when creating procedures and functions. Code with multiple exits is considered more complex than code with only one exit.

A RETURN statement is used in a PROCEDURE

Package:	Severity:	Category:	Code Element:
Built-in	Major	Program Structure	Subprogram

Rationale:

Reasoning:

The RETURN statement completes the execution of a subprogram immediately.

Explanation:

There is a basic principle called 1-in/1-out when creating procedures and functions. Code with multiple exits is considered more complex than code with only one exit.

FUNCTION with more than one RETURN statement in the executable section

Package:	Severity:	Category:	Code Element:
Built-in	Major	Program Structure	Function

Rationale:

Reasoning:

The RETURN statement completes the execution of a subprogram immediately.

Explanation:

There is a basic principle called 1-in/1-out when creating procedures and functions. Code with multiple exits is considered more complex than code with only one exit.

Presence of more than one exit point from a loop

Package:	Severity:	Category:	Code Element:
Built-in	Major	Program Structure	Loop

Rationale:

Reasoning:

Multiple exit points cause an unstructured termination of the loop. This type of design breaks the "one in - one out" ideal of a loop, creating code which can be hard to debug and maintain.

Explanation:

There is a basic principle called 1-in/1-out when creating procedures and functions. Code with multiple exits is considered more complex than code with only one exit.

Unreferenced loop index

Package:	Severity:	Category:	Code Element:
Built-in	Trivial	Program Structure	Loop

Rationale:

Reasoning:

When the loop index is not used in a loop this often implies that the loop may not be needed at all. It is often possible to rewrite any SQL inside such a loop as SQL ONLY without the looping structure.

CASE without ELSE (default) section

Package:	Severity:	Category:	Code Element:
Built-in	Major	Readability	Conditional

Rationale:

Reasoning:

A CASE statement must always have a default condition or this logic construct is non-deterministic. Generally, the default condition should warn the user of an anomalous condition which was not anticipated by the programmer.

Explanation:

There is one common exception, but even this exception should be avoided. Note that in the good/bad examples, the CASE expression is being used as a PL/SQL COMMAND where each section of the CASE statement executes a code block. CASE can also be used as a function in which case the entire CASE statement will return a single value. In this usage, the default value returned when no condition is satisfied is NULL. Experienced PL/SQL developers often use this default behavior if CASE to skip coding an ELSE. It is recommended however that even when NULL is the desired default when no other condition satisfies, an ELSE expression explicitly returning NULL should be coded. This make the CASE statement fully unambiguous, even for new developers who do not yet understand the default behavior of CASE as a function.

FUNCTION exception handler does not contain a RETURN statement

Package:	Severity:	Category:	Code Element:
Built-in	Major	Program Structure	Function

Rationale:

Reasoning:

A function must return a value unless it propagates an unhandled exception. For a function, this means that an exception handler should also issue a RETURN statement unless it is re-raising an exception.

Exception masked by a NULL statement

Package:	Severity:	Category:	Code Element:
Built-in	Critical	Program Structure	Exception

Rationale:

Reasoning:

Code that contains an exception handler which contains only a NULL statement is dangerous, because the code appears to handle that exception, but does not. If the "handled" exception occurs, it is quietly discarded without doing any sort of recovery action. Sometimes this is desired behavior. But more often it is a bug in exceptional handling created by an unskilled developer who has not had to live with the horrors of debugging code that does not provided error messages. The most accepted practice is the write exception handlers ONLY for exceptions that will be explicitly handled by the code, and allow all other exceptions to pass to the caller. One possible variation of this is the use of exception handling sections (INCLUDING WHEN OTHERS) to capture error info. If an exception section is being used to capture error data for example by using an AUTONOMOUS TRANSACTION to save the error message and related debugging info to a database table, then the exception section should re-raise the original exception. It is also recommended to exploit the newer features of the DBMS_UTILITY package. Using exception handlers to capture error info and then re-raising the exception, is highly encouraged.

Explanation:

Consider this example where a WHEN OTHERS is used to capture error info. Note how the exception is re-raised using the RAISE statement. Also note that the called SAVE_ERROR_INFO procedure must be an AUTONOMOUS_TRANSACTION to ensure the error is captured without affecting the transaction semantics of the calling code. In this example, there is only one exception handler. If the exception section had multiple handlers with several explicitly trapped errors, each handler must contain the same error trapping call.

Exception is not handled inside the unit

Package:	Severity:	Category:	Code Element:
Built-in	Major	Program Structure	Exception

Rationale:

Reasoning:

When a function or a procedure does not contain any exception handler, any exception occurring during execution will be passed on up the call stack. This implies that the entry has can either return the normal way or via an exception. This is not necessarily incorrect.

But often a return value of NULL provides a better way to handle some situations, allowing the caller to determine if no result is acceptable.

Explanation:

An additional consideration is use of exceptions as a means of flow control, which should be frowned upon and never done. While it is perfectly legitimate to raise an exception and then trap that exception in the same scope, this is sometimes misused as a means of affecting a kind of GOTO command. One good indicator of this problem may be found in the declaration of the exception. If the name of the exception describes an action as opposed to an error, then there is a very good chance that exceptions are being misused. With few special cases, exceptions should ONLY be used for errors, not to branch execution control. The RAISE statement is an easy and powerful way to abort normal processing in a program and immediately "go to" the appropriate WHEN handler. Never RAISE an exception for this purpose. ONLY raise an exception when an error has occurred AND the code will be explicitly handling that exception. Do not raise an exception to control program flow. The following program demonstrates the problem; it performs a full table scan of a collection and immediately exits when it finds a match. The exit_function exception is used to abort the function if the input title is NULL; it is also used as the last line in the function.

An exception is raised and handled in the same scope

Package:	Severity:	Category:	Code Element:
Built-in	Major	Program Structure	Exception

Rationale:

Reasoning:

An additional consideration is use of exceptions as a means of flow control, which should be frowned upon and never done.

While it is perfectly legitimate to raise an exception and then trap that exception in the same scope, this is sometimes misused as a means of affecting a kind of GOTO command. One good indicator of this problem may be found in the declaration of the exception. If the name of the exception describes an action as opposed to an error, then there is a very good chance that exceptions are being misused. With few special cases, exceptions should ONLY be used for errors, not to branch execution control. The RAISE statement is an easy and powerful way to abort normal processing in a program and immediately "go to" the appropriate WHEN handler. Never RAISE an exception for this purpose. ONLY raise an exception when an error has occurred AND the code will be explicitly handling that exception. Do not raise an exception to control program flow. The following program demonstrates the problem; it performs a full table scan of a collection and immediately exits when it finds a match. The exit_function exception is used to abort the function if the input title is NULL; it is also used as the last line in the function

DELETE or UPDATE without WHERE clause

Package:	Severity:	Category:	Code Element:
Built-in	Major	Maintainability	SQL

Rationale:

Reasoning:

DELETE and UPDATE statements without WHERE clauses are perfectly legal in SQL. Such statements do affect all rows in the table used. With the exception of trunc/load processes, it is unusual to want to change all rows in a table. This makes it very likely that a WHERE clause was left out unintentionally. A double check of the requirements specification if warranted to validate such a global effect on the table is desired.

The Parameter Complexity metric of a stored program exceeds the specified maximum

Package:	Severity:	Category:	Code Element:
Built-in	Minor	Maintainability	Parameter

Rationale:

Reasoning:

Stored programs with a large number of input parameters are difficult to use and are subject to parameter order errors. These types of stored programs are generally poorly designed and should be examined for the design constraint for a subprogram to perform a single purpose.

END of program unit or package is not labeled

Package:	Severity:	Category:	Code Element:
Built-in	Minor	Readability	Label

Rationale:

Reasoning:

PL/SQL allows for code blocks to be named with labels. If labels are being used, then care should be taken to ensure that: - Meaningful labels are selected - The same label appears at the end of a labeled code block as does its beginning. Following these two rules will provide easier reading of labeled code objects, particularly if they are nested as in the case of many layers of nested loops. This is most evident when these loops span multiple screens or pages. The end of a code block being more than one page away from its beginning means that the visual clue of what code belongs to which nested block is lost. The end label helps to restore that.

Explanation:

For packaged procedures and functions with code that goes on for hundreds or thousands of lines, named ENDs are crucial to improving the readability of that package. Imagine a package that consists of 243 procedures and functions, stretching to over 5,000 lines. Without END labels, the following is exposed.

```
    END LOOP;
  END LOOP;
END;
```

It will be much better if the code had been written with well named labels. The real difficulty with this is to ensure that the end label matches the starting label and that choice of labels is meaningful. Failure to do so does not create a problem for the PL/SQL compiler, but it will cause more confusion for those reading the code than it remedies.

```
    END LOOP inner_loop;
  END LOOP outer_loop;
END nested_loop;
```

The label near the END of the block doesn't match the block label, or a block label is missing

Package:	Severity:	Category:	Code Element:
Built-in	Minor	Readability	Label

Rationale:

Reasoning:

Source code must be easily read. Although ORACLE allows any label to be associated with the END statement, it should match the start label. Failure to match labels increases confusion rather than removing it.

END of CASE statement should also be labeled

Package:	Severity:	Category:	Code Element:
Built-in	Trivial	Readability	Conditional

Rationale:

Reasoning:

Repeating label names on the END of CASE statements ensures consistency throughout the code. Use of LABELS is usually optional. But if labels are used, the end label should match the start label. There is no requirement to use a label with a case statement, or for any code block for that matter. However, if a label is used, care must be taken to ensure the same label appears at the end of the case statement as well. This promotes easier reading which provides at least one justification for use of the label.

Cursor reference to an external variable (use a parameter)

Package:	Severity:	Category:	Code Element:
Built-in	Major	Maintainability	PL/SQL

Rationale:

Reasoning:

A cursor may contain direct references to PL/SQL variables. Such references create side effects as the cursor's result depends on the PL/SQL variable. This is easy to overlook, and it may severely limit the re-usability of the cursor. A better approach is to use a PARAMETERISED CURSOR. Parameters can be defined in the cursor specification, for every value the cursor depends upon. This makes it easier to see what the cursor needs to work correctly, and makes the cursor reusable without modification to variable names that may differ between use cases.

The comment percentage of a stored program is less than specified minimum

Package:	Severity:	Category:	Code Element:
Built-in	Minor	Readability	Comment

Rationale:

Reasoning:

The degree of commenting within source code measures the care taken by the programmer to make the source code understandable. Poorly commented code makes the maintenance phase of the software life cycle an extremely expensive one.

Explanation:

- Comments in code are an opportunity for a developer to make themselves very different from all others around them
- A small percentage of people believe that a computer program should have 0 comments because a good developer should write code which is self-explanatory.
- The problem with this is that 99% of all developers are average, so they do not ever write code that is self-explanatory.
- Others believe that the ideal program is one which contains 50% instrumentation, 50% commentary, 50% white space. Naturally this is unachievable adding up to 150%, but at least it reflects a goal with purpose.
- The real opportunity however is in mating code comments with a good documentation tool. CLEARSQL provides one such tool called the PSEUDOCODE EDITOR.

Avoid use of an explicit cursor

Package:	Severity:	Category:	Code Element:
Built-in	Major	Program Structure	Cursor

Rationale:

Reasoning:

The implicit cursor is more concise to write than explicit and does all the appropriate checking, opening and closing for you. Implicit cursors will run FASTER than explicit since typically they require less code to achieve the same task. Implicit cursors also offer more internal optimizing opportunity for the PL/SQL compiler; consider for example the enhancement of FOR LOOP automatically converted by the PL/SQL compiler into something similar to bulk fetch.

Explanation:

Use of cursors in general should be frowned upon. This is because cursors promote a ROW-BY-ROW attitude to writing code which is ALMOST always less efficient, less succinct, and less portable than a SET BASED (SQL ONLY) approach to development. But for those few situations where a ROW-BY-ROW piece of code is warranted, use of IMPLICIT cursors is ALMOST always better than use of EXPLICT cursors. This is because of EXPLICIT cursors suffer from a second detractor which is locality of reference. An explicit cursor must be defined in the declarative section of code. This means it is remote to the code that uses it. In practical terms, this results in "page flipping". You will open a cursor and process its output, but if you want to really know what the cursor itself does, you must refer to the EXPLICIT cursor definition. If your program unit is more than 30 lines long, this will require navigating out of sight of the code you are analyzing. This is disruptive and should be avoided as much as possible. An implicit cursor puts the cursor code close to the code that uses it and thus reduces or even eliminates the "page flipping" phenomenon.

Exception is not handled in the BEGIN/END block

Package:	Severity:	Category:	Code Element:
Built-in	Major	Program Structure	PL/SQL

Rationale:

Reasoning:

Raised when the BEGIN/END block does not contain an exception handler.

Explanation:

It is not necessary, indeed not even desirable, for every routine to have its own exception handler. It is desirable to capture an error when it occurs, and to do so at a location as close as possible to where the error occurred. If code is trapping errors, then it should have an exception handler that does the necessary recording of the error. The code can always re-raise the error after handling it so that higher level components can be aware of the error and react accordingly.

Mode of parameter is not specified with IN parameter

Package:	Severity:	Category:	Code Element:
Built-in	Minor	Readability	Parameter

Rationale:

Reasoning:

Two reasons explain why it is better to explicitly state parameter usage than to rely on the implicit rules of a compile.

1. Implicit rule can change over time. Though rare, there are occasions where compiler time rules may change, or be "enhanced". If this happens, it will cause an unexpected behavior change to your process.
2. But more importantly, it simply provides better natural documentation for you system.

Stored program calls itself recursively

Package:	Severity:	Category:	Code Element:
Built-in	Major	Program Structure	Subprogram

Rationale:

Reasoning:

If the stored program's identifier is used within the stored program's block, the stored program is executed recursively. Recursion is supported by the PL/SQL engine, but each recursive (self-calling) stored program should be carefully analyzed to make sure that recursion is valid and efficient.

The initialization section of the package body contains a RETURN statement

Package:	Severity:	Category:	Code Element:
Built-in	Minor	Program Structure	Package

Rationale:

Reasoning:

A return statement causes program execution to stop, and return to the caller. Because of this, if a return statement is in a package initialization section, it will cause the package to terminate and return to the caller BEFORE it executes any code. Though this may not be an error as it is syntactically valid, it is rare and thus deserves an explanation.

This definition hides another one

Package:	Severity:	Category:	Code Element:
Built-in	Major	Program Structure	Variable

Rationale:

Reasoning:

PL/SQL allows us to declare variables with the same names in nested blocks of code. The PL/SQL compiler will be able to tell these structures apart because PL/SQL uses an internal unique identifier for each variables, rather than the generic name. But a person reading and maintaining the code does not have this advantage, and can easily mistake one reference declaration for another. Though not always possible (often due to naming rules), effort should be made to use distinct names throughout the scope of a block of code (including blocks nested within that outer-most block). A more common case of this problem occurs when the index of a FOR LOOP is also declare as a variable. This section often arises when you declare a variable for the index of a FOR loop (an integer or record, depending on the type of loop). You should avoid such an action, for reasons described below. Never declare the FOR loop index. PL/SQL offers two kinds of FOR loops: numeric and cursor. Both have this general format.

Unreferenced parameter

Package:	Severity:	Category:	Code Element:
Built-in	Minor	Maintainability	Parameter

Rationale:

Reasoning:

A parameter has been defined in the parameter list, but it is not used inside the program. This may reflect an aged piece of code that has seen many changes whereby some of the parameter are no longer needed. An unused parameter in code can reflect potentially unfinished code, or a desire to avoid refactoring interfaces due to a change in contract. A serious discussion is needed among the development team to determine how such a situation should be addressed. There are sometimes valid reasons for keeping "dead code" in place. You may sometimes want to "turn off" code temporarily. Also, you may need to comment out some logic, but still show that this action was done and why. In such cases, make sure that you include the necessary documentation in the code. Even better, use problem tracking or bug reporting software to keep a comprehensive history of any changes made to code.

Don't use a backward GOTO

Package:	Severity:	Category:	Code Element:
Built-in	Critical	Program Structure	GOTO

Rationale:

Reasoning:

GOTO… there is a lot of advice, and a long-standing debate over its use. The short of it is this.

GOTO is never actually needed. This was shown several times in formal papers between 1932 and 1990.

It has also been shown that there are several coding idioms wherein use of GOTO provides the most expedient and easiest to read and debug solution over any of the other alternative rewrites. Thus, it is acceptable in these cases to use GOTO. What is not

acceptable however is use of a GOTO that flows backwards through code. A GOTO should always jump forward. Indeed, the most commonly cited valid use of GOTO is for early termination of nested loops which by definition is jumping forward in code flow. In the end, it is a question of using the proper control structures for the task at hand. An interesting note: a study testing skills of inexperienced programmers showed that 80% of them could not code the correct solution for certain problem cases when denied use of GOTO. The alternative coding formulations for the problem cases was too complex for these inexperienced minds. But 100% of these same inexperienced developers got the solution correct when allowed to use GOTO.

Unreferenced local variable

Package:	Severity:	Category:	Code Element:
Built-in	Minor	Maintainability	Variable

Rationale:

Reasoning:

A variable is defined locally, but it is not used inside the program. An unused variable is usually a reflection of an incomplete code component, or an error. In some cases, unused references are debugging features and if so may be quite acceptable if accompanying commentary explains this. But although is potentially a good practice, it is not common. The more likely explanation for dead code is that it is just old and no one wanted to remove it.

Complex expression is not fully parenthesized

Package:	Severity:	Category:	Code Element:
Built-in	Minor	Readability	SQL

Rationale:

Reasoning:

The common precedence rules in PL/SQL make many parentheses unnecessary. When a complex expression occurs, it may be helpful to add parentheses for clarity, even when the precedence rules apply.

Elements in the SELECT list (columns/expressions) are not qualified by a table/view name

Package:	Severity:	Category:	Code Element:
Built-in	Minor	Readability	SQL

Rationale:

Reasoning:

Qualifying select list tokens (indeed all tokens anywhere in a SQL statement) provides three benefits.

1. It eliminates the need to search tables to find what table supplies a column, thus making code easier to read.
2. It avoids potential unwanted capture which can occur when column names change of move across tables, as happens in typical data evolution.
3. It speeds up query parse time since the optimizer also does not need to search tables to find the source of a referenced column.

END of labeled LOOP should also be labeled

Package:	Severity:	Category:	Code Element:
Built-in	Minor	Readability	Label

Rationale:

Reasoning:

PL/SQL allows blocks of code to be named. But when naming a block of code with a label, the end of the block of code should also be named with the same label. This provides an easy reference for anyone reviewing the code, particularly if the code block spans more than one page. It also gives a better management for nested blocks of code. A loop is a block of code, and thus it can be labeled.

IF..THEN..EXIT should be replaced by EXIT WHEN

Package:	Severity:	Category:	Code Element:
Built-in	Minor	Readability	Label

Rationale:

Reasoning:

IF – THEN – ELSE – END IF; is certainly a useful control construct. However, if the only thing it does is EXIT, then alternative constructs might be a more readable choice. EXIT WHEN is the usual alternative.

Nested loops should all be labeled

Package:	Severity:	Category:	Code Element:
Built-in	Trivial	Readability	Label

Rationale:

Reasoning:

PL/SQL allows code blocks to be named by using a label. Labels are not required. But when using labels, care should be taken to

- Choose a meaningful name for each label
- Ensure that the end of the labeled code block contains that same label as its start.

Labels in some cases provide an easy way to identify code, particularly in cases where nested blocks of code span multiple screens or multiple pages. In such a case, the visual clue of what code belongs with which nested code block is lost. Beginning and ending labels restore some of the lost clue.

The column alias name is not specified after the AS keyword

Package:	Severity:	Category:	Code Element:
Built-in	Minor	Readability	SQL

Rationale:

Reasoning:

Raised when the column name is not specified in the column list of the SQL statement after the AS keyword. A column alias provides clarity to a SQL statement by ensuring select elements has a useful and clear name. Normally this is just the column name, in which case this rule can be disregarded. However, for any expression, a column alias is mandatory. An alias is also mandatory when multiple columns with the same name from different tables appear in the select list of a join query. To disambiguate this situation, consider use of a role prefix.

The label near the END of the loop doesn't match the loop label, or a loop label is missing

Package:	Severity:	Category:	Code Element:
Built-in	Minor	Readability	Label

Rationale:

Reasoning:

PL/SQL allows blocks of code to be named with a label. This is not a common practice. But it can provide significant benefit when faced with nested blocks of code. Labels make it easier to see the start and end of each code block. If labels are used, they should be placed at both ends of the code block and they should be the same label.

Avoid using the "magic" hard-coded literal numeric value in the WHERE clause of the SQL statement

Package:	Severity:	Category:	Code Element:
Built-in	Minor	Maintainability	SQL

Rationale:

Reasoning:

Raised when a hard-coded literal numeric value is used in the WHERE clause of the SQL statement.

Do not use a literal value because it has no obvious meaning. Declare a constant to hold a literal value and use it in your code. This helps in maintenance as the human-readable name of the constant explains the meaning and may suggest what it stands for. It also helps to ensure consistency in case the values change.

Avoid using the "magic" hard-coded literal string value in the WHERE clause of the SQL statement

Package:	Severity:	Category:	Code Element:
Built-in	Minor	Maintainability	SQL

Rationale:

Reasoning:

Raised when a hard-coded literal string value is used in the WHERE clause of the SQL statement.

Do not use a literal value because it has no obvious meaning. Declare a constant to hold a literal value and use it in your code. This helps in maintenance as the human-readable name of the constant explains the meaning and may suggest what it stands for. It also helps to ensure consistency in case the values change.

Local program unit reference to an external variable

Package:	Severity:	Category:	Code Element:
Built-in	Major	Maintainability	Variable

Rationale:

Reasoning:

Local procedures and functions offer an excellent way to avoid code redundancy and make code more readable (and thus more maintainable). But when a local program makes a reference to an external data structure, i.e., a variable that is declared outside of the local program, that referenced variable becomes a global variable inside the program. This causes the local program to have an external dependency which is hidden. External

references are the foundation of unwanted side effects, leading to corrupted data. Code (particularly PL/SQL) should be modularized with brutality, and use of parameters to avoid external references is an essential part of modularization. With few exceptions, an externally referenced variable should be replaced with a parameter in the parameter list of the local program. This decouples the local program from other code, making it more reusable, reducing scoping complexity, and eliminating the possibility of side effects from the now no longer existing external reference.

FUNCTION does not contain a RETURN statement

Package:	Severity:	Category:	Code Element:
Built-in	Major	Program Structure	Function

Rationale:

Reasoning:

A function must return a value. To do this it needs a RETURN statement (PIPELINED TABLE FUNCTIONS use PIPE ROW instead of RETURN). Thus, it is an error not to have a RETURN statement in a function. PL/SQL will allow a function to be compiled without a RETURN statement, but will generate an error at runtime with the function ends without returning a value.

FUNCTION has no parameter

Package:	Severity:	Category:	Code Element:
Built-in	Trivial	Program Structure	Function

Rationale:

Reasoning:

It is not an error for a function to not have a parameter list. However, such cases are rare. It is possible that such a function is supposed to always return the same value, or the same logical value. It is also possible that a function may be DATA DRIVEN, relying on contents of tables to provide driving values. It is also possible that the function may refer to external variables, or rely on hard coded constants and these last two cases are generally bad.

FUNCTION has OUT parameter

Package:	Severity:	Category:	Code Element:
Built-in	Major	Program Structure	Function

Rationale:

Reasoning:

There is a fundamental difference between a PROCEDURE and an FUNCTION in programming science. A FUNCTION represents knowledge. A PROCEDURE represents a process. A FUNCTION should never change anything. To make a change to something (data, system state, etc.) a PROCEDURE should be used. Thus, a FUNCTION provides a value back to the caller and it does this through its RETURN definition. This value may be a complex data item but it is still a single value. However, most programming languages do not enforce this definition of a function (cannot change anything, must return only one value using its return definition). PL/SQL also does not enforce function ideals. It is possible to for example, return data from a function both via its RETURN definition AND via an OUT or IN OUT parameter. This however is bad. It causes the function to behave in an unexpected manner (normal programmers do not expect a function to modify its parameter values), and limits the use of the function (a function with an OUT or IN OUT parameter cannot be called from SQL).

If multiple data points must be returned by a function, the function can be adapted using any of these approaches:
- Create a type declaration in a package or a database object/collection such that the type defines all of what must be returned and fill and return said type from the function.
- Break the function into multiple functions that each return one of the data points.
- Use a PIPELINED function.
- Change the function to a procedure and use one or more OUT or IN OUT parameters.
- Any combination of the above that suites the purpose.

Last statement in FUNCTION must be a RETURN

Package:	Severity:	Category:	Code Element:
Built-in	Major	Program Structure	Function

Rationale:

Reasoning:

The last executed statement in a function must be a return. Additionally, it is accepted as good programming practice that a function should have only one way out via a single RETURN statement, and that this statement should be the last statement in the source code of the function. When this advice is not followed, a function that is hard to read is created.

Explanation:

Of note is the fact that early languages allowed for execution to jump into any location of a code stream, and also allowed that code stream to return to any number of multiple locations outside the code stream; the freedom afford by ASSEMBLER LANGUAGE, and Fortran's ALTERNATE ENTRY and ALTERNATE RETURN being excellent examples. The jist of the posted answer at the URL is that the issue of single return was never that your code should have only one RETURN statement, but that when your called routine returned, it should always go back to the same place, the place that called it, since abilities like ALTERNATE ENTRY / ALTERNATE RETURN proved to be amazingly error prone. If was from this realization that SINGLE ENTRY / SINGLE EXIT was born.

Today's languages, PL/SQL included, do not typically support either of these. Thus in a more modern IT world the question of single exit, has lost this history and morphed into coding style concerns focused on readability and maintainability. With that in mind we suggest the following base rule, and noted exceptions:
1. Having only one RETURN statement and having it at the end of your function, is a expected practice. Doing what is expected usually results in more favorable readability and

maintenance results over time. As an example, if you want to debug the return value in your function, this is much easier done with a single return at the end of your routine. Otherwise you will need to locate all exit points and add redundant code to achieve the same debugging results. This is just one example of the value of a single RETURN at the end of your function.
2. HOWEVER... there are some noted exceptions that are at least worth discussing. Two that are most referenced are the concepts of the GUARDING PATTERN and the MONOLITHIC NESTING PATTERN. It can be advantageous to have routines check the validity of their inputs and/or otherwise evaluate inputs to see if containing to proceed at the beginning of the routine is warranted. Errors in inputs should prompt an exception, and discovery of no work to do should prompt an early exit of some kind. In this case, many developer prefer to code a GUARD section fronted in the routine to do this. Your preference to code a RETURN in the GRAUDING code, or to set a variable and GOTO <> of your routine is up to you. Personally I prefer actually using a GOTO (it seems somehow poetic to make a restricted idea more accepted by using yet another restricted idea).

Loop index redeclared

Package:	Severity:	Category:	Code Element:
Built-in	Major	Program Structure	Loop

Rationale:

Reasoning:

The loop index is implicitly declared at run-time and should never be declared explicitly by the programmer. The scope of this loop index variable is restricted to the body of the loop. When a loop index variable is redeclared, a separate variable is declared with block (not loop!) scope. It can be used outside the loop and can be confused with the loop's index variable. Redeclared Loop Index is something of a poor name for this condition. What it really means is the loop index used is the same as a name defined at a higher scope. The problem created is one of clarity as although scoping rules dictate an unambiguous usage, a casual read of the code is none-the-less confusing because it is not clear what references to the index mean.

PARALLEL optimizer hint is used

Package:	Severity:	Category:	Code Element:
Built-in	Major	Program Structure	SQL

Rationale:

Reasoning:

Raised when the SQL statement contains the PARALLEL optimizer hint. Use of PARALLEL hint in a SQL statement is valid syntax. And it will usually produce a parallel execution. However, in database versions 11g and higher, the general wisdom is to let Oracle do the driving when determining if a query should be run in parallel or not with manual intervention only for extreme scenarios. For 99% of databases, letting Oracle choose, results in the most optimum use of parallelism across a database system. The bigger concern should be one of abuse of parallelism. When faced with a slow query, sometimes desperation drives us to invoke parallelism to try and force a faster result. Unfortunately,

this is a dangerous mistake. Consider for example using PARALLEL (10) to try and make a slow query go fast. In this situation, the most likely result is that a slow query that used a lot of resources, will become a slow query that uses 10 times as much resources. Before considering use of parallelism, a query should be tuned first. Most queries once tuned do not need parallelism, and success related to use of parallelism is usually achieved by turning it off.

RULE optimizer hint is used

Package:	Severity:	Category:	Code Element:
Built-in	Major	Program Structure	SQL

Rationale:

Reasoning:

Raised when the SQL statement contains the RULE optimizer hint. RULE hint is still valid. But like all hints, it should be reviewed whenever seen. The issue is to make sure that this hint is not a mistake, but is instead used to make certain an expected query plan does not change. There are three significant issues behind use of RULE hint:

- in the future RULE may become invalid which will eventually lead to unexpected plan changes, and RULE prevents use of
- RULE hint prevents use of certain features
- The reverse is also true; if the query in question is modified to include an advanced feature (for example analytics) then the RULE hint will be ignored.

Unreferenced local subprogram

Package:	Severity:	Category:	Code Element:
Built-in	Minor	Program Structure	PL/SQL

Rationale:

Reasoning:

A local subprogram was defined, but it is not used inside the package. An unreferenced subprogram is considered dead code in most scenarios. A package that does not have such dead code is easier to maintain, but sometimes there may be a reason to keep this dead code in place. In such cases the necessary documentation should be included in the code. A local subprogram in PL/SQL is one whose specification does not appear in the package spec. This means the local program cannot be called from outside the package code itself. If in addition, the package code does not call the local subprogram, then this means the local subprogram is in fact not used at all. This is unreachable code.

- unreachable code can in fact cause unexpected code execution behavior even though the code is never executed, depending upon compiler options used.
- unreachable code has the obvious drawbacks of consuming memory, changing locality of reference of other code when it is loaded into memory, and has a maintenance cost.
- there is in fact at least one legitimate use of unreachable code, which is that it can be used later in a program debugger to help debug other code by manually jumping

to the dead code after a breakpoint is reached. In reality for PL/SQL developers, 99% of us do not routinely use a debugger at all, and of the other 1%, who among us has actually thought of this idea?

Unreferenced package/type method or standalone subprogram

Package:	Severity:	Category:	Code Element:
Built-in	Major	Program Structure	PL/SQL

Rationale:

Reasoning:

A package/type method or standalone subprogram was defined, but it is not used inside the project. A project that does not have such dead code is easier to maintain, but sometimes there may be a reason to keep this code in place. For example, if it is used in a string literal of a dynamic SQL statement. In such cases the necessary documentation should be included in the code.

An EXIT statement in a labeled loop should have the loop's label

Package:	Severity:	Category:	Code Element:
Built-in	Major	Readability	Loop

Rationale:

Reasoning:

Labels in PL/SQL are not commonly used. But they do provide at least two benefits.

1. They support early exit out of a loop.
2. They provide natural documentation that may be useful in long sections of code that span multiple pages.

Mandatory comment header is missing or incorrect in the script

Package:	Severity:	Category:	Code Element:
Built-in	Trivial	Maintainability	Comment

Rationale:

Reasoning:

Raised when the script does not contain the comment header defined in Code Analyzer Options or is incorrect. Header template definition may contain {$ANY_LINE_ENDING} and {$ANY_LINES} wildcards to make the template definition flexible. The comment header is an indication that code has been formatted. Formatting is a necessary element of readability yet is often skipped.

The true purpose of this rule is to ensure that code has actually be run through some formatter. Since this tool is CLEARSQL, we tag on the existence of the comment header. The importance of good formatting cannot be overstated. Formatting is one of the top 3 mechanisms for improving readability, so considerable time should be spent with your formatting tool of choice to standardize formatting as much as possible. It is easy to format code in a tool so it should be done such that all code has the same feel. You can exploit the CLEARSQL COMMENT HEADER as described below.

- The "MANDATORY COMMENT HEADER" is a CLEARSQL thing. This rule enforces the existence of a comment at the beginning of your script. You use the comment header definition to specify a template the comment must match.
- {$ANY_LINE_ENDING} is a wildcard for any text on a single line.
- {$ANY_LINES} allows for an infinite line of anything
- notice how TEXT:{$ANY_LINE_ENDING} appears after the DECSCRIPTION line. The existence of a new wildcard (in this case {$ANY_LINE_ENDING}) ends the previous wildcard fit.
- Use can comment headers to enforce a minimum comment standard.
- A future feature enhancement may allow for this header to actually be embedded inside the code unit. This is currently not possible.

The script contains more than one CREATE statement

Package:	Severity:	Category:	Code Element:
Built-in	Trivial	Maintainability	SQL

Rationale:

Reasoning:

It is convenient to store the code of each single object in a separate file to ensure you can maintain version control independently. This also makes it easier to install a new version of an object while having minimal impact on other schema objects.

The eLOC within a stored program exceeds the specified maximum

Package:	Severity:	Category:	Code Element:
Built-in	Minor	Maintainability	PL/SQL

Rationale:

Reasoning:

An extremely large stored program is very difficult to maintain and understand. These types of stored programs are generally not well designed and could be broken down into several programs. A long program is not a program with lots of lines. A large SELECT

statement with many columns can add hundreds of lines to a PL/SQL program. Several of these will add thousands. A long PL/SQL program will though have many PL/SQL lines. Long is somewhat ambiguous, so feel free to apply whatever standard your company employs, or whatever you feel is reasonable. One common reason programs can get long is due to lack of MODULARITY.

The Interface Complexity metric of a stored program exceeds the specified maximum

Package:	Severity:	Category:	Code Element:
Built-in	Minor	Maintainability	Subprogram

Rationale:

Reasoning:

Stored programs with many input parameters are difficult to use on a routine basis and are problematic for parameter ordering. A procedure or function with a large number of parameters usually indicates one of two things:

- the code component is not modular and should be refactored
- the wrong algorithm has been chosen for a specific task

GOTO used in a loop

Package:	Severity:	Category:	Code Element:
Built-in	Major	Program Structure	GOTO

Rationale:

Reasoning:

GOTO is in general something to be avoided. It is an old holdback from the early days of programming (before 3GL), when there were less abstract alternatives. However, it is also a unwarrantedly maligned coding construct. It is true that there is no requirement to code a GOTO as it has been proven mathematically that it is possible to always rewrite a code snippet that includes a GOTO into a variation that does not. However, one of the principles of programming is that READABILITY is of high value. There are two scenarios where GOTO is a valid statement. Remember, the validity lies in the ability of the GOTO statement to improve the READABILITY of the code, not in its convenience of a quick fix.

1. GUARDING code that sends execution of a code component to its end so that it may exit.
2. Early exit from a series of nested loops.

A predefined exception is raised

Package:	Severity:	Category:	Code Element:

Built-in	Major	Program Structure	Exception

Rationale:

Reasoning:

The program has explicitly raised an Oracle Predefined Exception. RAISE ZERO_DIVIDE is an example of a predefined exception. These exceptions are owned by the database and should not normally be used by application code explicitly. Instead of raising a predefined exception, a program unit should declare its own exceptions using the EXCEPTION statement. Doing so allows the application to give names to business defined errors. A user-defined exception should be declared for every condition that is considered an error. This allows the application to document clearly, the full list of error conditions, and provides for different exception handlers to handle these exceptions later should they be raised.

A PREDEFINED EXCEPTION is an error which the database has explicitly given a name, because it is common and thus benefits from being more recognizable in use.

- ZERO_DIVIDE
- NOT_LOGGED_ON
- TOO_MANY_ROWS
- DUP_VAL_ON_INDEX
- INVALID_NUMBER

In addition to pre-defined exceptions, and user-defined exceptions, the RAISE_APPLICATION_ERROR() function can be used to dynamically raise an error in code.

In addition, any Oracle error can be turned into a "pre-defined" exception of a sort using the EXCEPTION_INIT pragma. So for example, although the DEADLOCK DETECTED error has no predefined exception, a PL/SQL unit could name the error and thus create a "pre-defined exception" for it.

The Functional Complexity metric of a stored program exceeds the specified maximum

Package:	Severity:	Category:	Code Element:
Built-in	Minor	Maintainability	Subprogram

Rationale:

Reasoning:

The stored program's functional complexity is comprised of Interface Complexity and Cyclomatic Complexity. An extremely complex stored program is very difficult to understand and maintain. Stored programs of this type could be broken down into a more modular design of smaller subroutines.

Currently, *Clear*SQL defines Interface Complexity as simply the count of required parameters, plus the number of RETURN statements.

An opened cursor must be closed

Package:	Severity:	Category:	Code Element:
Built-in	Critical	Program Structure	Cursor

Rationale:

Reasoning:

Cursor require memory. Thus, closing them is somewhat important. In the case of PL/SQL it is true (most of the time) that when a cursor goes OUT-OF-SCOPE, the PL/SQL engine will automatically close the cursor for you.

However, there are two problems with this.

1. There are special situations where this rule does not apply.
2. If a REF CURSOR is passed outside the oracle database (external java routine for example), then Oracle no longer controls it and it may remain open indefinitely. This can cause ORA-01000 maximum open cursors exceeded.

Thus, it is always good practice for a code component to explicitly close any cursor it has opened.

Explanation:

REFCURSORS notwithstanding:

- Not closing an open cursor when you are done with it, is sloppy, confusing, and potentially a memory leak.
- PL/SQL does close a cursor when it goes out of scope. But, the rules for when a cursor goes out-of-scope are undocumented and have "special situations".

END of labeled block should also be labeled

Package:	Severity:	Category:	Code Element:
Built-in	Major	Readability	Label

Rationale:

Reasoning:

In PL/SQL, pieces of code can be given names by using LABELS. Labels are usually associated with creating target points for GOTO commands, but they can also be used to name pieces of code. Doing so creates a form of natural documentation of the code, and allows for the disambiguation of same named variables, in nested code components. But perhaps most importantly, it provides an visual hook to the END of a code unit, which can be very valuable in cases when the code unit is many lines long and thus spans multiple pages or screens.

Repeating names on the end of these compound statements ensures consistency throughout the code. In addition, the named end provides a reference for the reader if the unit spans a page or screen boundary or if it contains a nested unit. While PL/SQL labels (identifiers within double angle brackets, like <>) are most often associated with GOTOs, they can be a big help in improving the readability of code. Use a label directly in front of loops and nested anonymous blocks: To give a name to that portion of code and thereby self-document what it is doing. So that you can repeat that name with the END statement of that block or loop. This recommendation is especially important when you have multiple nestings of loops (and possibly inconsistent indentation), as in the following.

COMMIT and ROLLBACK are allowed if pragma is used

Package:	Severity:	Category:	Code Element:
Built-in	Major	Program Structure	PL/SQL

Rationale:

Reasoning:

Raised when a COMMIT or ROLLBACK statement used in the stored program that does NOT contain the pragma AUTONOMOUS_TRANSACTION statement. The issue with COMMIT/ROLLBACK is that these statements end a transaction and start a new one. This then implies some form of TRANSACTION DESIGN has taken place as part of the design of the application. Unfortunately, it is common that no such design has been done. Thus, it is important to study the use of COMMIT/ROLLBACK to see if proper transactions do in fact exist for the process in question.

Not all use of COMMI/ROLLBACK is wrong. At some point one or the other must be done. In addition, there are situations where it is desirable to record information without disturbing the current state of a process's transaction. For this purpose, the pragma AUTONOMOUS_TRANSACTION was created. This pragma indicates that the associated piece of code intends to do a commit or rollback, but wishes to do so as a separate transaction. The most common occurrence of this is in capturing error information in an exception handler.

DROP statement is used

Package:	Severity:	Category:	Code Element:
Built-in	Trivial	Program Structure	SQL

Rationale:

Reasoning:

Raised when the script contains a DROP statement. The rule for DROP contained in a script is merely a cautionary note, asking you to revalidate the need for the DROP statement, and the potential for error. Common and valid uses of DROP and when replacing code, or when recreating a table as part of some ETL process. Both of these however have alternatives that do not require use of DROP.

Dynamic SQL is used

Package:	Severity:	Category:	Code Element:
Built-in	Minor	Program Structure	SQL

Rationale:

Reasoning:

Raised when DBMS_SQL.PARSE call, EXECUTE IMMEDIATE statement or OPEN FOR statement with dynamic expression is used. Like all tools, DYNAMIC SQL can be useful at the proper time. However, 9 out of 10 times that DYNAMIC SQL is used, it is unnecessary. This is of significant concern since there are several problems associated with DYNAMIC SQL. DYNAMIC SQL suffers from these detractors.

- It is slower than precompiled code in most cases requiring compilation every time it is used.
- It cannot be analyzed by code analysis tools like CLEARSQL that will provide things CALL TREES, CRUD MATRIX, FLOW CHART, PSEUDO CODE, DEPENDENDY DIAGRAMS.
- Mapping executed SQL to code locations becomes more difficult.
- It bypasses Oracle's normal object dependency tracking. In fact, DYNAMIC SQL is often abused as a hack to do just this very thing.
- MOST IMPORTANTLY, it opens the door to SQL INJECTION. DYNAMIC SQL is the most commonly used exploit for SQL INJECTION.

Never use DYNAMIC SQL unless you absolutely must. (Personal observation -- 9/10 cases where I see DYNAMIC SQL, it is not needed). SQL should be hard compiled. Embedded SQL that is compiled will always be faster, safer, trackable via metadata in the data dictionary, and identifiable inside utility tools. DYNAMIC SQL is none of these things.

Global public variables defined in package specification

Package:	Severity:	Category:	Code Element:
Built-in	Major	Program Structure	Package

Rationale:

Reasoning:

PACKAGE GLOBAL VARIABLES do have a few specific use cases.

1. For defining CONSTANTS (values that cannot change), that are for use across a body of PL/SQL components, and thus will be used by many PL/SQL code units. These can readily be placed in a common package for the specific purpose of organizing them and making them easy to use. The recommend advice here is that constants which are used across a set of PL/SQL code components (an application?) should go in a single package specification so that all constants for this group of code components can be centralized; and constants that are used in only one package can be defined globally in that package alone.
2. For those who DO NOT have available, the Enterprise version of the database and thus do not have access to the FUNCTION RESULT CACHE. In this case PACKAGE GLOBAL VARIABLES can be used "Old School" to simulate this functionality and thus acquire its performance benefit.

Aside from these use cases though, PACKAGE GLOBAL VARIABLES present at least two significant issues:

1. A "SECURITY" problem.
2. Increased risk of buggy code.

Consider a system control table. This table is used to control the behavior of PL/SQL applications at runtime, allowing someone to dynamically set information collection settings. Notice how considerable effort has been expended in the initial database design

to secure access, and to ensure quality of expected values. However, as the BAD and GOOD code examples show, even this is not enough. Depending upon how the instrumentation package that collects the different type of instrumentation data is written, it may have to make many (possibly millions) of calls and thus a corresponding number of selects from the control table (once every time it wants to collect instrumentation). This can be a performance issue. So, it sounds like a great opportunity for PACKAGE GLOBAL VARIABLES. The app can load the state of instrumentation when it first starts, into some PACKAGE GLOBAL VARIABLES. Then all apps for the session will have the same behavior. But if so, the data has now been "disconnected" so to speak from the underlying source. And we have introduced potential for bugs, since a PACKAGE GLOBAL VARIABLE can be changed by anyone who can see it. BAD code would allow this to happen. GOOD code would hide the variables under a pair of GET/SET routines.

Place default parameters at the end of the formal parameter list

Package:	Severity:	Category:	Code Element:
Built-in	Minor	Program Structure	Parameter

Rationale:

Reasoning:

Placing default parameters at the end of the formal parameter list allows the caller to use positional association on the call. Otherwise, defaults are available only when a named association is used.

On the other hand, when new parameters of a procedure are placed at the end of the parameter list, callers of this procedure that user positional notation, aside from the need to have to recompile their application, will not be affected by broken code, at least in the sense that their calls are still valid. However, it is preferred to use NAMED NOTATION instead. This provides the same protection, eliminates ambiguity of parameter placement, and provides additional "natural documentation".

If NAMED NOTATION is not a standard for your shop, then by all means place DEFAULTED parameters at the end of your procedures/function to avoid compilation issues. However, since NAMED NOTATION is a better approach (though it takes more typing), when using NAMED NOTATION, this rule does not matter.

PROCEDURE has no parameter

Package:	Severity:	Category:	Code Element:
Built-in	Trivial	Program Structure	Subprogram

Rationale:

Reasoning:

It is not an error for a procedure to not have a parameter list. However, such cases are rare. It is possible that the procedure may be DATA DRIVEN, relying on contents of tables to provide driving values. It is also possible that the procedure may refer to external variables, or rely on hard coded constants and these last two cases are generally bad.

Initialization to NULL is superfluous

Package:	Severity:	Category:	Code Element:
Built-in	Trivial	Readability	NULL

Rationale:

Reasoning:

By default, NON-OBJECT variables are always initialized to NULL, unless they are explicitly set otherwise. Thus, it is somewhat verbose to include the NULL keyword. The issue at hand is one of expectation. Most PL/SQL developers are lazy (which is usually a good thing for someone in IT since laziness is a primary driver of innovation among IT PROFESSIONALS). Hence, most PL/SQL developers will not expect to see the NULL keyword in this location. When it does then appear, this causes delay in processing meaning. One must slow down to process the unexpected yet redundant information. This specific rule is maybe a small thing, but it does open the door to discussion on the topic of how the way you write code affects other people's ability to comprehend your code, and thus what you can do the improve upon a piece of code's ability to convey meaning about itself. It is a good example of how less can be more.

Positional parameters are used instead of named parameters

Package:	Severity:	Category:	Code Element:
Built-in	Minor	Maintainability	Parameter

Rationale:

Reasoning:

Once you start using NAMED NOTATION for passing parameters in procedure calls you will start to like it. It is not generally popular because it takes additional effort (more typing) and it was not valid to use named notation in SQL that called function, but this limitation has been removed in recent DBMS versions. Named notation provides additional natural documentation to your code and reduces chances of parameter passing errors because it insulates a calling procedure from certain types of changes to the calling contract by disambiguating the mapping of parameters. Positional notation relies on correctly coded parameter order. Named notation does not, it instead relies on correctly named parameters.

Specify a full column list (as opposed to using "*") in each DML statement and cursor

Package:	Severity:	Category:	Code Element:
Built-in	Major	Maintainability	SQL

Rationale:

Reasoning:

Specifying a fully named column list instead of using * in a SELECT statement or CURSOR insulates the query change column changes in the underlying tables, views, and materialized views. A dropped/renamed column will cause a compile error, but an added column will not.

A similar but alternate thinking revolves around typing using %ROWTYPE.

SQL*Plus command SHOW ERRORS is missing in the script

Package:	Severity:	Category:	Code Element:
Built-in	Major	Maintainability	SQL*Plus

Rationale:

Reasoning:

Raised when there are no "SHOW ERROR" commands in the script. SHOW ERRORS provides feedback on compile time errors of stored code. When using it, a second query of USER_ERRORS can be skipped. It is a convenience. Though sounding small, this is a simple example of techniques that take on more importance in a world governed by SOX compliance.

The Cyclomatic Complexity [v(G), McCabe] metric of a stored program exceeds the specified maximum

Package:	Severity:	Category:	Code Element:
Built-in	Major	Maintainability	PL/SQL

Rationale:

Reasoning:

Cyclomatic complexity [McCabe] is the degree of logical branching within a stored program. A high degree of v(G) indicates that the stored program could be broken down into a set of smaller stored programs, supporting the design concept that a stored program should be specific to one purpose. Macabe complexity tries to track how many different paths there are through code. More paths mean more complex code. Flowcharts show this well.

The Halstead Volume metric of a stored program exceeds the specified maximum

Package:	Severity:	Category:	Code Element:

Package:	Severity:	Category:	Code Element:
Built-in	Major	Maintainability	PL/SQL

Rationale:

Reasoning:

In simple terms Halstead suggests that: "the more things" a program has in it, the harder it will be to write and understand. So, if you have "too many" things in your code, maybe you should break your code into smaller chunks, each of which has a lot less things, and thereby reduce the amount juggling in your head you need to do at any one moment.

Explanation:

Halstead Metrics were developed by the late Maurice Halstead as a means of determining a quantitative measure of complexity directly from the operators and operands in the module to measure a program module's complexity directly from source code.

Among the earliest software metrics, they are strong indicators of code complexity. Because they are applied to code, they are most often used as a maintenance metric. There is evidence that Halstead measures are also useful during development, to assess code quality in computationally-dense applications. Because maintainability should be a concern during development, the Halstead measures should be considered for use during code development to follow complexity trends.

Halstead´s metrics is based on interpreting the source code as a sequence of tokens and classifying each token to be an operator or an operand:

N1 – number of operators
N2 – number of operands
n1 – number of unique operators
n2 – number of unique operands

The following five metrics of the subprogram are calculated from the preceding primitives:

1. Program Length
N = N1+ N2
2. Program Vocabulary
n = n1+ n2
3. Volume
V = N * log2(n)
4. Difficulty
D = n1/2 * (N2/n2)
5. Effort
E = D * V

ClearSQL treats the Halstead Volume (HV) metric as one the main metric in PL/SQL code measurement.

The Maintainability Index metric of a stored program is lower than the specified minimum

Package:	Severity:	Category:	Code Element:
Built-in	Major	Maintainability	PL/SQL

Rationale:

Reasoning:

Maintainability Index is a software metric that measures how maintainable (easy to support and change) source code is. The Coleman-Oman model is the most used model for determining the Maintainability Index (MI) of source code. Maintainability Index is based on Halstead Volume, Cyclomatic Complexity, the average number of lines of code per module, and the percentage/ratio (depends on the setting) of comment lines per module. The higher the MI, the more maintainable a system is deemed to be. A value lower than 64 indicates that the routine is probably difficult to maintain.

Maintainability Index is a computed value, derived from "adding" several metrics together, and is used mostly to simplify comparisons of code maintainability between modules. The value of this metric for code development is debatable. Since it does not identify any single factor as a problem, it provides no guidance on a course of action that will achieve remediation for writing more maintainable code. This metric does however provide a single number easy for management to understand, which in turn allows for a relative comparison of multiple software systems against each other, for long term maintainability. If one system has a significantly higher value, it is deemed more maintainable than the other. Similarly, this metric provides value in judging the overall effectiveness of reengineering efforts. A successful reengineering effort should result in a lower percentage of high maintenance modules; and this metric can measure that. It is in these ways that this metric is normally used, and not as a directive in how to change a piece of code to make it more maintainable.

Though this metric provides no guidance on how to make any single piece of code better, since this metric uses other metrics as its base, more maintainable code can be achieved by adhering to coding practices that favor those metrics used by this metric. In particular: MODULARITY, COMMENTARY, decoupling through removal of EXTERNAL REFERENCES, and significant use of WHITE SPACE (blank lines and consistent formatting) will go a long way to improving this metric.

Identifier redeclared

Package:	Severity:	Category:	Code Element:
Built-in	Minor	Program Structure	PL/SQL

Rationale:

Reasoning:

All declared identifiers must be unique within the same scope. Variables, constants, and parameters cannot share the same name even if they have different datatypes.

Trigger may potentially cause mutating table error

Package:	Severity:	Category:	Code Element:
Built-in	Major	Program Structure	Trigger

Rationale:

Reasoning:

Mutating table exceptions occur when trying to query or modify a table from a row-level trigger that the triggering statement is modifying. You can avoid the mutating table error in either of these ways: use a compound trigger and a collection; use a temporary table or a collection in the package variable and a combination of row-level and statement-level triggers; etc.

Raise of a user-defined exception with the RAISE statement

Package:	Severity:	Category:	Code Element:
Built-in	Minor	Program Structure	Exception

Rationale:

Reasoning:

Avoid raising a programmer-defined exception with the RAISE statement because there is no way to provide an explanation in an error message. Use the RAISE_APPLICATION_ERROR procedure because it associates a user-defined exception with a user-defined message.

The alias is missing for a table reference in a multi-source query

Package:	Severity:	Category:	Code Element:
Built-in	Minor	Readability	SQL

Rationale:

Reasoning:

Raised when the alias is not specified after a table reference clause in the FROM clause of a query, except for single-table queries and references to the DUAL table. A table alias clarifies which table you are referring to in a query and improves the readability of a SELECT statement.

The object type doesn't correspond to the file extension

Package:	Severity:	Category:	Code Element:
Built-in	Trivial	Maintainability	Script

Rationale:

Reasoning:

It is convenient to store the code of each single object in a separate file to ensure you can maintain version control independently. This also makes it easier to install a new version

of an object while having minimal impact on other schema objects. A distinct file extension helps to identify the type of the object and allows storing the package specification and body in files with different extensions, but same names.

Avoid using the "magic" hard-coded literal numeric value in PL/SQL code

Package:	Severity:	Category:	Code Element:
Built-in	Trivial	Maintainability	PL/SQL

Rationale:

Reasoning:

Do not use a literal value because it has no obvious meaning. Declare a constant to hold a literal value and use it in your code. This helps in maintenance as the human-readable name of the constant explains the meaning and may suggest what it stands for. It also helps to ensure consistency in case the values change.

EXCEPTIONS:

Because the meaning/purpose of the literal value is obvious in such code, this rule is not regarded as violated in the following cases:

- initialization of the standalone subprogram parameter
  CREATE PROCEDURE DO_IT(PARAM1 NUMBER DEFAULT 1)
- simple loop range value when it's 0 or 1
  FOR I IN 0..COUNT-1 LOOP
  FOR I IN 0..1 LOOP
- compare results of COUNT and LENGTH functions with 0
  COUNT(SOME_TABLE) > 0, LENGTH(STR_VAR) = 0...
- increment/decrement by 1 in the assignment statement
  I := I + 1;
  I := I - 1;

Avoid using the "magic" hard-coded literal string value in PL/SQL code

Package:	Severity:	Category:	Code Element:
Built-in	Trivial	Maintainability	PL/SQL

Rationale:

Do not use a literal value because it has no obvious meaning. Declare a constant to hold a literal value and use it in your code. This helps in maintenance as the human-readable name of the constant explains the meaning and may suggest what it stands for. It also helps to ensure consistency in case the values change.

EXCEPTION:

Because the meaning/purpose of the literal value is obvious in such code, this rule is not regarded as violated in the following case:

- initialization of the standalone subprogram parameter
  CREATE PROCEDURE DO_IT(PARAM1 VARCHAR(10) DEFAULT 'DUMMY')

Use INNER JOIN instead of NATURAL JOIN

Package:	Severity:	Category:	Code Element:
Built-in	Major	Maintainability	SQL

Rationale:

Reasoning

A NATURAL JOIN is a join based on the columns appearing in both tables to be joined. While this may seem like a useful short-hand at first glance, more thought will show that this makes maintaining the application a nightmare as table maintenance on a table invloved in a natural join may alter the semantics of the SELECT statement implicitly if the maintenance operation causes the set of common column names to change. A DBA changes a column name in a table, and queries with NATURAL JOIN start returning a different answer. NATURAL JOIN should be avoided at all costs and, when found, should be rewritten as normal joins with explicit column names. NATURAL JOIN was originally an academic concept in the original conception of SQL and was intended only in that context as a means of teaching SQL concepts. It was never meant to be implemented. Natural join is a bug waiting to happen. This is because it brings ambiguity and potential for side effects into code.

Avoid use of AUTONOMOUS_TRANSACTION pragma

Package:	Severity:	Category:	Code Element:
Built-in	Minor	Program Structure	Package

Rationale:

Reasoning

PRAGMA AUTONOMOUS_TRANSACTION allows developers to code nested transactions in Oracle. Such nested transactions can only be coded in PL/SQL. The original and still possibly the only valid use of AUTONOMOUS_TRANSACTION was to allow recording error messages and debugging information which requires a commit, without disturbing a process’s current transaction semantics. For example, an error message should be preserved in the database even if the error causes the active transaction to be rolled back. Any other use of AUTOMONUS_TRANSACTION is a bug of some kind that will eventually lead to data corruption and increased debugging issues.

Calls to DBMS_ADDM require the Diagnostic and Tuning Packs licenses

Package:	Severity:	Category:	Code Element:
Built-in	Major	License	PL/SQL

Rationale:

Reasoning

Some Oracle accessible code in part of an Oracle OPTION. See the reference below for details. Oracle suggests that if you do not have a license for said option, you should not be calling the associated code components individually. The legality of such a restriction has been debated since it seems unrealistic to make a piece of code available in a standard distribution, but then claim in a license agreement that it should not be called. It has been suggested that Oracle should control such code components to prevent them from being called if a user is unlicensed for them rather than allowing customers to become reliant upon them and then at a later date point out their use and demand additional license fees. In any event, DBMS_ADDM is one such package. It is part of the Diagnostic and Tuning management pack. This pack is one of several EXTRA COST features of the Oracle database. Whenever any EXTRA COST component is referenced, it is advised that the customer using them should double check licensing availability. See the reference below for a full list of OPTIONS / PACKS / OTHER SOFTWARE that is EXTRA COST for an Oracle database.

Calls to DBMS_ADVISOR require the Diagnostic and Tuning Packs licenses

Package:	Severity:	Category:	Code Element:
Built-in	Major	License	PL/SQL

Rationale:

Reasoning

Some Oracle accessible code in part of an Oracle OPTION. See the reference below for details. Oracle suggests that if you do not have a license for said option, you should not be calling the associated code components individually. The legality of such a restriction has been debated since it seems unrealistic to make a piece of code available in a standard distribution, but then claim in a license agreement that it should not be called. It has been suggested that Oracle should control such code components to prevent them from being called if a user is unlicensed for them rather than allowing customers to become reliant upon them and then at a later date point out their use and demand additional license fees. In any event, DBMS_ADDM is one such package. It is part of the Diagnostic and Tuning management pack. This pack is one of several EXTRA COST features of the Oracle database. Whenever any EXTRA COST component is referenced, it is advised that the customer using them should double check licensing availability. See the reference below for a full list of OPTIONS / PACKS / OTHER SOFTWARE that is EXTRA COST for an Oracle database.

Calls to DBMS_SQLPA require Real Application Testing option and Database Tuning Pack license

Package:	Severity:	Category:	Code Element:
Built-in	Major	License	PL/SQL

Rationale:

Reasoning

Some Oracle accessible code in part of an Oracle OPTION. See the reference below for details. Oracle suggests that if you do not have a license for said option, you should not be calling the associated code components individually. The legality of such a restriction has been debated since it seems unrealistic to make a piece of code available in a standard distribution, but then claim in a license agreement that it should not be called. It has been suggested that Oracle should control such code components to prevent them from being called if a user is unlicensed for them rather than allowing customers to become reliant upon them and then at a later date point out their use and demand additional license fees. In any event, DBMS_ADDM is one such package. It is part of the Diagnostic and Tuning management pack. This pack is one of several EXTRA COST features of the Oracle database. Whenever any EXTRA COST component is referenced, it is advised that the customer using them should double check licensing availability. See the reference below for a full list of OPTIONS / PACKS / OTHER SOFTWARE that is EXTRA COST for an Oracle database.

Calls to DBMS_SQLTUNE require Database Tuning Pack license

Package:	Severity:	Category:	Code Element:
Built-in	Major	License	PL/SQL

Rationale:

Reasoning

Some Oracle accessible code in part of an Oracle OPTION. See the reference below for details. Oracle suggests that if you do not have a license for said option, you should not be calling the associated code components individually. The legality of such a restriction has been debated since it seems unrealistic to make a piece of code available in a standard distribution, but then claim in a license agreement that it should not be called. It has been suggested that Oracle should control such code components to prevent them from being called if a user is unlicensed for them rather than allowing customers to become reliant upon them and then at a later date point out their use and demand additional license fees. In any event, DBMS_ADDM is one such package. It is part of the Diagnostic and Tuning management pack. This pack is one of several EXTRA COST features of the Oracle database. Whenever any EXTRA COST component is referenced, it is advised that the customer using them should double check licensing availability. See the reference below for a full list of OPTIONS / PACKS / OTHER SOFTWARE that is EXTRA COST for an Oracle database.

Calls to DBMS_WORKLOAD_CAPTURE require Real Application Testing option license

Package:	Severity:	Category:	Code Element:
Built-in	Major	License	PL/SQL

Rationale:

Reasoning

Some Oracle accessible code in part of an Oracle OPTION. See the reference below for details. Oracle suggests that if you do not have a license for said option, you should not be calling the associated code components individually. The legality of such a restriction has been debated since it seems unrealistic to make a piece of code available in a standard distribution, but then claim in a license agreement that it should not be called. It has been suggested that Oracle should control such code components to prevent them from being called if a user is unlicensed for them rather than allowing customers to become reliant upon them and then at a later date point out their use and demand additional license fees. In any event, DBMS_ADDM is one such package. It is part of the Diagnostic and Tuning management pack. This pack is one of several EXTRA COST features of the Oracle database. Whenever any EXTRA COST component is referenced, it is advised that the customer using them should double check licensing availability. See the reference below for a full list of OPTIONS / PACKS / OTHER SOFTWARE that is EXTRA COST for an Oracle database.

Calls to DBMS_WORKLOAD_REPLAY require Real Application Testing option license

Package:	Severity:	Category:	Code Element:
Built-in	Major	License	PL/SQL

Rationale:

Reasoning

Some Oracle accessible code in part of an Oracle OPTION. See the reference below for details. Oracle suggests that if you do not have a license for said option, you should not be calling the associated code components individually. The legality of such a restriction has been debated since it seems unrealistic to make a piece of code available in a standard distribution, but then claim in a license agreement that it should not be called. It has been suggested that Oracle should control such code components to prevent them from being called if a user is unlicensed for them rather than allowing customers to become reliant upon them and then at a later date point out their use and demand additional license fees. In any event, DBMS_ADDM is one such package. It is part of the Diagnostic and Tuning management pack. This pack is one of several EXTRA COST features of the Oracle database. Whenever any EXTRA COST component is referenced, it is advised that the customer using them should double check licensing availability. See the reference below for a full list of OPTIONS / PACKS / OTHER SOFTWARE that is EXTRA COST for an Oracle database.

Calls to DBMS_WORKLOAD_REPOSITORY require the Diagnostic and Tuning Packs licenses

Package:	Severity:	Category:	Code Element:
Built-in	Major	License	PL/SQL

Rationale:

Reasoning

Some Oracle accessible code in part of an Oracle OPTION. See the reference below for details. Oracle suggests that if you do not have a license for said option, you should not be calling the associated code components individually. The legality of such a restriction has been debated since it seems unrealistic to make a piece of code available in a standard distribution, but then claim in a license agreement that it should not be called. It has been suggested that Oracle should control such code components to prevent them from being called if a user is unlicensed for them rather than allowing customers to become reliant upon them and then at a later date point out their use and demand additional license fees. In any event, DBMS_WORKLOAD_REPOSITORY is one such package. It is part of the Diagnostics and Tuning pack. This pack is one of several EXTRA COST features of the Oracle database. Whenever any EXTRA COST component is referenced, it is advised that the customer using them should double check licensing availability. See the reference below for a full list of OPTIONS / PACKS / OTHER SOFTWARE that is EXTRA COST for an Oracle database.

Avoid deprecated data types RAW, LONG and LONG RAW

Package:	Severity:	Category:	Code Element:
Built-in	Major	Maintainability	PL/SQL

Rationale:

Reasoning

In Oracle 8i data types LOB, CLOB and BFILE have been introduced to supplant the old data types LONG, RAW and LONG_RAW. The newer data types offer numerous advantages over the older ones and lift restrictions:

- Tables can have more than one column of CLOB/LOB/BFILE data types
- Columns of CLOB/LOB/BFILE data types can be used in WHERE-Clauses
- Columns of CLOB/LOB/BFILE data types can be used in PL/SQL
- Tables containing columns of CLOB/LOB/BFILE data types can be used in SQL statements like INSERT INTO...SELECT, CREATE TABLE AS SELECT

Eventually these old types will be discontinued. It is advised to replace deprecated data types with their newer versions as part of a tech currency initiative.

Avoid use of DBMS_DDL package's deprecated subprograms

Package:	Severity:	Category:	Code Element:
Built-in	Major	Maintainability	PL/SQL

Rationale:

Reasoning

The entry ALTER_COMPILE in the DBMS_DDL package has been deprecated by Oracle. Calls to DBMS_DDL.ALTER_COMPILE should be rewritten to use an equivalent EXECUTE IMMEDIATE statement.

Avoid use of DBMS_ADVISOR package's deprecated subprograms

Package:	Severity:	Category:	Code Element:
Built-in	Major	Maintainability	PL/SQL

Rationale:

Reasoning

The package DBMS_ADVISOR contains deprecated entries. Oracle recommends that you do not use deprecated procedures in new applications. Support for deprecated features is for backward compatibility only.

The following subprograms are currently deprecated in DBMS_ADVISOR:

- ADD_SQLWKLD_REF
- CREATE_SQLWKLD
- DELETE_SQLWKLD
- DELETE_SQLWKLD_REF
- DELETE_SQLWKLD_STATEMENT
- IMPORT_SQLWKLD_SCHEMA
- IMPORT_SQLWKLD_SQLCACHE
- IMPORT_SQLWKLD_STS
- IMPORT_SQLWKLD_SUMADV
- IMPORT_SQLWKLD_USER
- RESET_SQLWKLD
- SET_SQLWKLD_PARAMETER
- UPDATE_SQLWKLD_ATTRIBUTES
- UPDATE_SQLWKLD_STATEMENT

Please check the Oracle Documentation for additional information. See additional references below.

Avoid use of DBMS_CDC_PUBLISH package's deprecated subprograms

Package:	Severity:	Category:	Code Element:
Built-in	Major	Maintainability	PL/SQL

Rationale:

Reasoning

The package DBMS_CDC_PUBLISH contains deprecated entries. Oracle recommends that you do not use deprecated procedures in new applications. Support for deprecated features is for backward compatibility only.

The following subprograms are currently deprecated in DBMS_CDC_PUBLISH:

- DROP_SUBSCRIPTION
- DROP_SUBSCRIBER_VIEW

Please check the Oracle Documentation for additional information. See additional references below.

Avoid use of DBMS_CDC_SUBSCRIBE package's deprecated subprograms

Package:	Severity:	Category:	Code Element:
Built-in	Major	Maintainability	PL/SQL

Rationale:

Reasoning

The package DBMS_CDC_SUBSCRIBE contains deprecated entries. Oracle recommends that you do not use deprecated procedures in new applications. Support for deprecated features is for backward compatibility only.

The following subprograms are currently deprecated in DBMS_CDC_SUBSCRIBE:

- DROP_SUBSCRIBER_VIEW
- GET_SUBSCRIPTION_HANDLE
- PREPARE_SUBSCRIBER_VIEW

Please check the Oracle Documentation for additional information. See additional references below.

Avoid use of DBMS_DATA_MINING package's deprecated subprograms

Package:	Severity:	Category:	Code Element:
Built-in	Major	Maintainability	PL/SQL

Rationale:

Reasoning

The package DBMS_DATA_MINING contains deprecated entries. Oracle recommends that you do not use deprecated procedures in new applications. Support for deprecated features is for backward compatibility only.

The following subprograms are currently deprecated in DBMS_DATA_MINING:

- GET_DEFAULT_SETTINGS
- GET_MODEL_SETTINGS
- GET_MODEL_SIGNATURE
- DM_USER_MODELS

Please check the Oracle Documentation for additional information. See additional references below.

Avoid use of deprecated package DBMS_IOT

Package:	Severity:	Category:	Code Element:
Built-in	Major	Maintainability	PL/SQL

Rationale:

Reasoning

The DBMS_IOT package creates a table into which references to the chained rows for an index-organized table can be placed using the ANALYZE command. DBMS_IOT can also create an exception table into which references to the rows of an index-organized table that violate a constraint can be placed when a constraint is enabled. Since it is a deprecated package, DBMS_IOT is not loaded during database installation. To install DBMS_IOT run dbmsiotc.sql, available in the ADMIN directory. With the introduction of logical rowids for IOTs with Oracle Database Release 8.1, you no longer need to use the procedures contained in this package which is retained for backward compatibility only. It is however required for servers running with Oracle Database Release 8.0.

Avoid use of DBMS_MGWADM package's deprecated subprograms

Package:	Severity:	Category:	Code Element:
Built-in	Major	Maintainability	PL/SQL

Rationale:

Reasoning

The package DBMS_MGWADM contains deprecated entries. Oracle recommends that you do not use deprecated procedures in new applications. Support for deprecated features is for backward compatibility only.

The following subprograms are currently deprecated in DBMS_MGWADM:

- ADD_SUBSCRIBER
- ALTER_PROPAGATION_SCHEDULE
- ALTER_SUBSCRIBER
- DB_CONNECT_INFO
- DISABLE_PROPAGATION_SCHEDULE
- ENABLE_PROPAGATION_SCHEDULE
- REMOVE_SUBSCRIBER
- RESET_SUBSCRIBER
- SCHEDULE_PROPAGATION
- UNSCHEDULE_PROPAGATION

Please check the Oracle Documentation for additional information. See additional references below.

Avoid use of DBMS_RESOURCE_MANAGER package's deprecated subprograms

Package:	Severity:	Category:	Code Element:
Built-in	Major	Maintainability	PL/SQL

Rationale:

Reasoning

The package DBMS_RESOURCE_MANAGER contains deprecated entries. Oracle recommends that you do not use deprecated procedures in new applications. Support for deprecated features is for backward compatibility only.

The following subprograms are currently deprecated in DBMS_RESOURCE_MANAGER:

- SET_INITIAL_CONSUMER_GROUP

Please check the Oracle Documentation for additional information. See additional references below.

Avoid use of DBMS_STATS package's deprecated subprograms

Package:	Severity:	Category:	Code Element:
Built-in	Major	Maintainability	PL/SQL

Rationale:

Reasoning

The package DBMS_STATS contains deprecated entries. Oracle recommends that you do not use deprecated procedures in new applications. Support for deprecated features is for backward compatibility only.

The following subprograms are currently deprecated in DBMS_STATS:

- GET_PARAM
- SET_PARAM
- RESET_PARAM_DEFAULTS
- GENERATE_STATS

Please check the Oracle Documentation for additional information. See additional references below.

Calls to UTL_SMTP may cause security issues

Package:	Severity:	Category:	Code Element:

Built-in	Major	Security	PL/SQL

Rationale:

Reasoning

UTL_SMTP provides a low-level interface to the simple mail transfer protocol. This package allows sending emails from within the database, in other words from the database server. In versions up to Oracle 10.2, you could send emails to any host and port completely unchecked. This was a massive security problem, because there was no way to ensure that emails were sent to authorized email servers only. Starting with Oracle 11.1, access needs to be explicitly granted to the host and port of the email server. Access is granted by creating an Access Control List (ACL). As ACLs are typically created by DBAs only, the security risk is much less in the more recent versions of the database.

Another concern is that database sent emails are not transactional. An email is sent as soon as the commands to UTL_SMTP have been completed, regardless of the active transaction, i.e. the email gets sent even if the active transaction is later rolled back. The non-transactional nature of UTL_SMTP also means using this package in "PRE" triggers poses a real problem. A pre-insert/update/delete trigger can internally roll back and restart. If this happens and an email had been sent, that email is not undone by the implicit roll back and the trigger restart will send the mail again. This will happen as many times as the trigger decides to do a rollback/restart cycle. This trigger's implied rollback/restart capability is not a well-documented characteristic. The issue is the same issue affecting package variables set from these triggers as well.

In Oracle DML statements like INSERT/UPDATE/DELETE statement can get re-executed, that is to say
- in case of a problem the statement gets rolled back
- execution starts again.

If you write an email in a pre-{insert|update|delete} trigger, and the statement gets re-executed, you will end up having sent your email multiple times, which might be embarrassing, depending on who the email is being sent to.

Calls to UTL_TCP may cause security issues

Package:	Severity:	Category:	Code Element:
Built-in	Major	Security	PL/SQL

Rationale:

Reasoning:

UTL_TCP provides a low-level interface to the TRANSPORT COMMUNICATION PROTOCOL. TCP is among other things a common communication protocol for the Internet. The UTIL_TCP package enables Oracle databases to create outbound connections to remote hosts on specified TCP ports. As such, it is a useful way to move data from the database to remote servers. There are legitimate uses for this, but it can also be used for illegitimate, i.e. undesired data transfers. In versions before Oracle 11g, Oracle did not perform any checks on the remote server being connected to. Connections could be made to any server whose IP address was known. Of course, you also had to find an open TCP port on the server. This, however, is easy as many well-known ports are often open on servers. Examples for that would be:

Port Service

- 22 SSH
- 25 SMTP
- 53 DNS
- 80 HTTP
- 443 HTTPS
- 3389 Terminal Services

In version 11g Oracle introduced Fine-Grained Access to external network resources by creating access control lists (ACL) for every network resource to be used from the database. Typically, the Database Administrators will create an ACL for the external network resources which should be accessible from the database. Attempts to access any resource for which no ACL exists will fail. By default, PUBLIC has the EXECUTE privilege on UTL_TCP. Just as with all network related packages, this should be changed for security reasons.

Calls to UTL_HTTP may cause security issues

Package:	Severity:	Category:	Code Element:
Built-in	Major	Security	PL/SQL

Rationale:

Reasoning:

UTL_HTTP is a utility package that has execute rights granted to PUBLIC by default. This package is owned by SYS, it is used to give the database access to the outside world by allowing the database to send and receive HTTP and HTTPS requests and responses. The UTL_HTTP package itself does not pose a direct threat to the database, nor can it be used to compromise the host operating system. At the same time, it is a real security threat. You need to act on this and revoke PUBLIC access from UTL_HTTP. UTL_HTTP provides the power to freely communicate across a network using HTTP, or even to communicate privately using HTTPS, which is SSL-protected. With the ability, an attacker may be able to send himself data without notice, often regardless of firewalls or data classification / extrusion detection systems.

In versions before Oracle 11g, Oracle did not perform any checks on the remote server to connect to. You could connect to any server whose IP address you knew. Of course, you also had to find an open TCP port on the server. This, however, is easy as many well-known ports are often open on servers. Examples for that would be:

Port Service
- 22 SSH
- 25 SMTP
- 53 DNS
- 80 HTTP
- 443 HTTPS
- 3389 Terminal Services

In version 11g Oracle introduced Fine-Grained Access to external network resources by creating access control lists (ACL) for every network resource to be used from the database. Typically, the Database Administrators will create ACL for the external network resources which should be accessible from the database. Attempts to access any resource for which no ACL exists will fail.

By default, PUBLIC has the EXECUTE privilege on UTL_HTTP. This should be changed for security reasons.

Calls to UTL_FILE may cause security issues

Package:	Severity:	Category:	Code Element:
Built-in	Major	Security	PL/SQL

Rationale:

Reasoning:

UTL_FILE is a utility package that has execute rights granted to PUBLIC by default. UTL_FILE allows the database to interact with the operating system files system. It opens the potential for several hacks including denial of service attacks caused by corrupting essential binary files, and the opportunity to retrieve sensitive data, including, for example, password hashes. Additionally, LOG files are a juicy target as they contain a record of changes made to data that can be interpreted with special decoding tools. UTL_FILE runs with the OS permissions of the Oracle account and so can read/write any file to which Oracle has access.

In versions before Oracle 11g, Oracle did not perform any checks on the remote server to connect to. You could connect to any server whose IP address you knew. Of course, you also had to find an open TCP port on the server. This, however, is easy as many well-known ports are often open on servers. Examples for that would be:

Port Service
- 22 SSH
- 25 SMTP
- 53 DNS
- 80 HTTP
- 443 HTTPS
- 3389 Terminal Services

In version 11g Oracle introduced Fine-Grained Access to external network resources by creating access control lists (ACL) for every network resource to be used from the database. Typically, the Database Administrators will create ACL for the external network resources which should be accessible from the database. Attempts to access any resource for which no ACL exists will fail.

By default, PUBLIC has the EXECUTE privilege on UTL_FILE. This should be changed for security reasons.

Calls to DBMS_RANDOM may cause security issues

Package:	Severity:	Category:	Code Element:
Built-in	Major	Security	PL/SQL

Rationale:

Reasoning:

DBMS_RANDOM presents a security risk when combined with other database settings. It opens the door to what is known as LATERAL SQL INJECTION. DBMS_RANDOM.VALUE is affected by NLS attack strategies. But there are other random number generator functions

available in Oracle which are not. DBMS_CRYPTO offers some. It is recommended not to use DBMS_RANDOM but instead use DBMS_CRYPTO.

Calls to DBMS_LOB may cause security issues

Package:	Severity:	Category:	Code Element:
Built-in	Major	Security	PL/SQL

Rationale:

Reasoning:

DBMS_LOB executes under the Oracle account. DBMS_LOB also permits the reading and writing of OS files. This means the DBMS_LOB package can be used to access any file in the system as the owner of the Oracle software installation. Like all other database supplied packages that allow for reading and/or writing of OS files, package access should be restricted to an as needed basis only. Grants to PUBLIC should also be revoked.

Calls to DBMS_BACKUP_RESTORE may cause security issues

Package:	Severity:	Category:	Code Element:
Built-in	Major	Security	PL/SQL

Rationale:

Reasoning:

DBMS_BACKUP_RESTORE is a PL/SQL command line interface for replacing native RMAN commands. It has very little documentation from Oracle. As using this package allows overwriting files required for the operation of the database instance (see examples), you should not allow your users to have execute rights on this package. At least, make sure that PUBLIC has execution rights to the DBMS_BACKUP_RESTORE package revoked.

Please check with Oracle support before you decide to use calls to this package.

Calls to EMD_SYSTEM may cause security issues

Package:	Severity:	Category:	Code Element:
Built-in	Major	Security	PL/SQL

Rationale:

EMD_SYSTEM package is vulnerable because it contains methods to access the file system.

Calls to DBMS_NAMESPACE may cause security issues

Package:	Severity:	Category:	Code Element:
Built-in	Major	Security	PL/SQL

Rationale:

Reasoning:

There is no official documentation for the DBMS_NAMESPACE package. That alone is enough warrant not using it. The WEB does provide some notes on obscure usages. One example is for executing shell functions. As this package is undocumented, we suggest that you use the well-documented DBMS_SCHEDULER package to execute shell scripts instead.

Calls to DBMS_SCHEDULER may cause security issues

Package:	Severity:	Category:	Code Element:
Built-in	Major	Security	PL/SQL

Rationale:

Reasoning:

The package DBMS_SCHEDULER is extremely powerful. Given the required GRANTS, the following attacks can be initiated by calling functions in the DBMS_SCHEDULER package:
- privilege escalation
- execution of external code on the database server like scripts and executables
Check that EXECUTE on DBMS_SCHEDULER is revoked from PUBLIC, and that only users who really need to use DBMS_SCHEDULER have EXECUTE on this package.

The DBMS_OUTPUT.PUT_LINE procedure is not intended for production use

Package:	Severity:	Category:	Code Element:
Built-in	Trivial	Program Structure	PL/SQL

Rationale:

Reasoning:

DBMS_OUTPUT.PUT_LINE provides only marginal value, and is best used for quick and dirty debugging that is not intended for production use.

Avoid use of DBMS_OUTPUT.PUT_LINE unless encased within a conditional compilation directive

Package:	Severity:	Category:	Code Element:
Built-in	Minor	Program Structure	PL/SQL

Rationale:

Reasoning:

DBMS_OUTPUT.PUT_LINE provides only marginal value, and is best used for quick and dirty debugging that is not intended for production use. Conditional compilation offers one alternative that allows the PL/SQL component to include or not to include these calls depending upon environment settings. This allows for a single piece of code to operate with or without these DBMS_OUTPUT.PUT_LINE calls as needed. Use them in DEV environments, then not compile them everywhere else.

Replace the "NOT %FOUND" expression with the %NOTFOUND cursor attribute

Package:	Severity:	Category:	Code Element:
Built-in	Minor	Readability	Cursor

Rationale:

Reasoning:

PL/SQL offers two closely related cursor attributes to check whether a cursor has been completely processed:

- cursorname%FOUND
- cursorname%NOTFOUND

These two cursor attributes complement each other: cursorname%FOUND is true if and only if cursorname%NOTFOUND is false.

We recommend that you prefer using cursorname%NOTFOUND over NOT cursorname%FOUND. cursorname%NOTFOUND is easier to read and understand. In addition, cursorname%NOTFOUND requires less work from the PL/SQL engine than NOT cursorname%FOUND. NOT cursorname%FOUND forces the PL/SQL engine to execute one extra instruction (Granted this is not a large gain, but still it is a gain you get for free!).

One additional consideration is that BEFORE THE FIRST ROW IS FETCHED, both cursor attributes will return NULL.

Use the SAVE EXCEPTIONS keywords in FORALL statements

Package:	Severity:	Category:	Code Element:
Built-in	Trivial	Program Structure	Exception

Rationale:

Reasoning

Since version 9, instead of raising an exception immediately during a bulk operation, it is possible to collect all the errors to a SQL%BULK_EXCEPTIONS array and to continue bulk processing. This array stores error details for each exception.

- %BULK_EXCEPTIONS(i).ERROR_INDEX - iteration number of the FORALL loop.
- %BULK_EXCEPTIONS(i).ERROR_CODE - Oracle error code.

Use the SAVE EXCEPTIONS keywords with FORALL and check for error -24381 to improve error handling

Package:	Severity:	Category:	Code Element:
Built-in	Major	Program Structure	PL/SQL

Rationale:

Reasoning:

Use the SAVE EXCEPTIONS keywords with FORALL and check for error -24381 to improve error handling. This is considerably more work, but it allows a process to continue in the face of what are non-terminating errors.

Using BULK COLLECT and FORALL instead of a loop that contains DML statements can significantly improve performance

Package:	Severity:	Category:	Code Element:
Built-in	Major	Program Structure	Collection

Rationale:

Reasoning

Bulk SQL minimizes the performance overhead of the communication between PL/SQL and SQL.

PL/SQL and SQL communicate as follows: To run a SELECT INTO or DML statement, the PL/SQL engine sends the query or DML statement to the SQL engine. The SQL engine runs the query or DML statement and returns the result to the PL/SQL engine.

The PL/SQL features that comprise bulk SQL are the FORALL statement and the BULK COLLECT clause. The FORALL statement sends DML statements from PL/SQL to SQL in batches rather than one at a time. The BULK COLLECT clause returns results from SQL to PL/SQL in batches rather than one at a time. If a query or DML statement affects four or more database rows, then bulk SQL can significantly improve performance.

You cannot perform bulk SQL on remote tables.

Assigning values to PL/SQL variables that appear in SQL statements is called binding. PL/SQL binding operations fall into these categories:

An In-bind occurs when an INSERT, UPDATE, or MERGE statement stores a PL/SQL or host variable in the database.
An Out-bind occurs when the RETURNING INTO clause of an INSERT, UPDATE, or DELETE statement assigns a database value to a PL/SQL or host variable.
A DEFINE occurs when a SELECT or FETCH statement assigns a database value to a PL/SQL or host variable.

For in-binds and out-binds, bulk SQL uses bulk binding; that is, it binds an entire collection of values at once. For a collection of *n* elements, bulk SQL uses a single operation to perform the equivalent of *n* SELECT INTO or DML statements. A query that uses bulk SQL can return any number of rows, without using a FETCH statement for each one.

Parallel DML is disabled with bulk SQL.

The below example shows a progression of PL/SQL code timings from 6200 seconds 5000 seconds 3 seconds 2 seconds (pure SQL). This is achieved through implementation of specific features. It should be noted that the fastest code is also the simplest code, which is not surprisingly the pure SQL approach.

Use the IS [NOT] NULL condition instead of comparing a variable with the NULL value

Package:	Severity:	Category:	Code Element:
Built-in	Major	Maintainability	NULL

Rationale:

Reasoning

Comparing a value to NULL will not return TRUE or FALSE, but NULL. This is not a bug but the intended behavior.

The ANSI SQL standard states that an operation must return NULL when at least one of its operands is NULL.

This intended behavior can cause confusion, because most people intuitively assume that a comparison can only be TRUE or FALSE, i.e. that the normal two-valued Boolean logic applies to comparisons.

This leads to bugs which are extremely hard to spot (after all, the code looks sensible at first glance) as well as hard to trace or debug.

You should therefore use the operators IS NULL or IS NOT NULL to check for NULL. These two operators always return TRUE or FALSE, which makes the code easier to understand and debug.

Avoid assigning empty strings to variables

Package:	Severity:	Category:	Code Element:
Built-in	Minor	Maintainability	NULL

Rationale:

Reasoning

In all Oracle versions up to 12.1, Oracle treats empty strings as NULL. According to the ANSI SQL standard, however, empty strings are not equivalent to NULL.

Using the ANSI SQL standard interpretation of empty string changes the semantics (the meaning) of many string operations, especially of string concatenations. According to the ANSI SQL standard, the result of any operation is NULL if at least one of the operands is NULL. With the Oracle interpretation of zero-length strings, 'A'||'' is the same as 'A', while the ANSI SQL standard requires it to be NULL. Should Oracle ever decide to use the ANSI SQL standard for empty strings, this would break most of the existing SQL and PL/SQL code.

Note that changing the code to make it deal with both interpretations of empty strings identically, is a major non-trivial project. Though the likelihood of Oracle changing its semantics is very small due to this problem, avoiding the problem in the first place seems best.

An IN OUT parameter is used as IN only; consider changing the mode to IN

Package:	Severity:	Category:	Code Element:
Built-in	Minor	Maintainability	Parameter

Rationale:

Reasoning

PL/SQL supports three different modes for a formal parameter of a function or procedure:

- IN - The parameter can be referenced by the procedure or function. The value of the parameter cannot be overwritten by the procedure or function.
- OUT - The parameter cannot be referenced by the procedure or function, but the value of the parameter can be overwritten by the procedure or function.
- IN OUT - The parameter can be referenced by the procedure or function and the value of the parameter can be overwritten by the procedure or function.

If a parameter mode is not specified explicitly, the default mode is IN. Explicitly showing the mode of all parameters is good programming practice as each reader of the code can be confident about the intended behavior. You should always use the mode that reflects the actual use of the parameter. Do not declare parameters as IN OUT unless you really want the functionality of an IN/OUT parameter. This means you should not declare a parameter of mode IN OUT if all you need is an IN parameter. IN/OUT parameters are slower than IN parameters due to the necessary copying of the value back to the actual parameter on every return point from the function or procedure.

More importantly, however, is potential confusion. When a parameter is specified as OUT or IN/OUT, anyone reviewing the code expects to see that the parameter is set to some value at least once in the code. If this is not seen, then this causes confusion about the use of the parameter, including raising the question as to if the parameter is being used correctly.

An IN OUT parameter is used as OUT only; consider changing the mode to OUT

Package:	Severity:	Category:	Code Element:
Built-in	Minor	Maintainability	Parameter

Rationale:

Reasoning

PL/SQL supports three different modes for a formal parameter of a function or procedure:

- IN - The parameter can be referenced by the procedure or function. The value of the parameter cannot be overwritten by the procedure or function.
- OUT - The parameter cannot be referenced by the procedure or function, but the value of the parameter can be overwritten by the procedure or function.
- IN OUT - The parameter can be referenced by the procedure or function and the value of the parameter can be overwritten by the procedure or function.

If a parameter mode is not specified explicitly, the default mode is IN. Explicitly showing the mode of all parameters is good programming practice as each reader of the code can be confident about the intended behavior. You should always use the mode that reflects the actual use of the parameter. Do not declare parameters as IN OUT unless you really want the functionality of an IN OUT parameter. This means you should not declare a parameter of mode IN OUT if all you need is an OUT parameter. IN OUT parameters are slower than OUT parameters due to the necessary copying of the parameter's actual value into the actual parameter on entering the function or procedure. It makes understanding the function or procedure more difficult.

Avoid use of WM_CONCAT; consider LISTAGG function

Package:	Severity:	Category:	Code Element:
Built-in	Major	Program Structure	SQL

Rationale:

Reasoning:

WM_CONCAT was an undocumented and unsupported feature and, thus, should never have been used in production code. WM_CONCAT was removed from the database version 12c. It is recommended to use the LISTAGG function instead.

Use the CASE statement instead of the IF-THEN-ELSIF statement

Package:	Severity:	Category:	Code Element:
Built-in	Minor	Readability	Conditional

Rationale:

Reasoning:

From the early days, IF THEN ELSE logic was the primary mechanism for implanting BRANCHING in 3GL languages. At some point the concept of a SWITCH statement, and the COMPUTED SWITCH statement, were recognized as useful ideas that should be given their own command representation in 3GL languages. In PL/SQL this is done with the CASE statement. Many believe that CASE is generally superior over IF. This can be debated. But the general idea is that CASE is aligned to a specific use scenario within the scope of branching theory. CASE may indeed be superior to IF, if only because it prompts anyone who sees it to think in a particular way. Thus, when the common branching pattern is used, it is generally preferable to use CASE rather than IF ELSIF syntax. But be warned, for any given 3GL language, implementation of CASE may not be exactly equivalent to IF logic which CASE may be replacing. This is in fact true for PL/SQL.

Explanation:

The real difference between CASE and IF is that the two statements are in fact not equivalent. The semantic differences between these two flow control statements is often language specific. In the case of PL/SQL, one difference is in how each control statement behaves when no match is detected. Consider the execution differences between these two "equivalent" commands as shown below. Though most would consider these two variations of code as logically identical, they are not. The CASE statement DEMANDS an accounting of NOT FOUND whereas IF does not. Thus, the two statements are not identical in their behavior.

Because CASE forces an accounting of what to do when no match is found among the provided conditions, many feel CASE is a superior construct. One ramification of this difference between behaviors of IF vs. CASE in PL/SQL is that it is not possible to simply substitute CASE for IF in existing code. In many instances, the ELSE option must be coded in the CASE variant even though it does not exist in the original IF that is being replaced, to mimic the original behavior of the code.

Another difference between CASE vs. IF is that the original intent of CASE was for use as a "SWITCH" statement. PL/SQL offers this version of CASE as well and it is called a SEARCHED CASE. The behavior of this variant of CASE is radically different and for use in a specific scenario.

First, we look at point #1, wherein CASE demands an accounting of NOMATCH. Note how the IF statement simply continues when no match is found, but the CASE statement raises the exception CASE_NOT_FOUND when no condition is satisfied. You must code ELSE NULL; at the end of the CASE if you wish to replicate the behavior of the IF statement when you replace it with a CASE statement.

Runtime concatenations of string literals affect performance

Package:	Severity:	Category:	Code Element:
Built-in	Minor	Maintainability	Literal

Rationale:

Reasoning:

This code concatenates two literal strings at runtime. This is seen often in production code. It has the disadvantage that the concatenation must be performed at runtime. From

a performance perspective it would be better to code a single string literal which contains the desired content. This makes the code faster, easier to read and to maintain.

There are at least three issues to look out for, when dealing with string concatenation issues:

1. Readability.
2. Flooding the SGA with a massive number of distinct SQL statements which if BIND VARIABLES were used, would result in only one reusable copy of the statement.
3. The possibility of SQL INJECTION. Most SQL INJECTION is done through code that does string concatenation followed by DYNAMIC SQL.

Use FETCH BULK COLLECT with the LIMIT clause instead of implicit SELECT BULK COLLECT to avoid unexpected session memory usage

Package:	Severity:	Category:	Code Element:
Built-in	Major	Program Structure	Collection

Rationale:

Reasoning:

BULK-COLLECT (and the usually accompanying FORALL) construct is a performance based feature. Its purpose is to allow a loop to fetch many rows which are put into process memory, instead of one row at a time in a simple loop. By using BULK-COLLECT, a process can bundle what would have been multiple calls to the database into just one. Fetching a million rows in groups of 1000 instead of 1 at a time requires only 1000 pings of the database instead of one million. This reduction in database communication events translates directly into a performance savings and it can be significant, particularly when paired with its sister feature FORALL.

However, fetching a set of rows in one go requires memory to house all rows fetched. Thus, when a BULK-COLLECT does not use a LIMIT clause, it is required to fetch all rows the underlying query will produce, and put them all into process memory. Using our case above, that would men putting one million rows in memory. It is easy to see how using BULK-COLLECT can for very large datasets, cause memory deprivation. The simple solution however is to use the LIMIT clause that goes with BULK-COLLECT. In this way, the code can control the amount of memory needed. A value of 100 is most common, and provides 99% of the performance gain that can be achieved by using the feature, while keeping the associated memory cost low.

The one drawback of BULK-COLLECT and FORALL when used together, is more extensive code needed for exception handling. Since code is now manipulating an array of rows rather than one row at a time, an error can occur anywhere in the array, even for multiple rows. Thus, if an error is detected, finding the offending row (or rows) requires code which is a bit more sophisticated, though quite doable, and clearly documented by Oracle.

Never save ROWIDs and UROWIDs to a table

Package:	Severity:	Category:	Code Element:
Built-in	Major	Maintainability	SQL

Rationale:

Reasoning:

For each row in the database, the ROWID pseudo column returns the address of the row in a normal (heap-organized) table. Note that the ROWID is not just a number, but has to contain multiple pieces of information:

- the file number of the data file in which the row is stored
- the data block number of the data block in which the row is stored
- the number of the row within the block in which the row is stored with 0 being the first record.
- the object number of the object to which the row belongs.

A UROWID (Universal ROWID) is a form of address which can be used as address in both heap-organized and index-organized tables. Because the abstract properties of the ROWID and UROWID pseudo columns are largely identical, we will talk about (U)ROWIDs in the remainder of this article. Accessing a row in the database via a known (U)ROWID value is ultra-fast. On the code sample accessing the data via the (U)ROWID column is almost twice as fast as accessing the code via the primary key. It is definitely okay to use a variable of the type (U)ROWID in your program code if you intend to reuse this value in short time. It is a pretty bad idea to try and reuse (U)ROWID values after a longer period of time. Still if you created a table which contains columns of the data type (U)ROWID in your application tables, you might encounter the following problems:

- the address of a row in the database may change over time. An example for that would be a partitioned table, where a row would have to be stored in a different partition, and hence in a different file or block within the same file, and hence the ROWID of that row would change. Had you stored the (U)ROWID of the row before the update to the partition key, that stored (U)ROWID would then point to an invalid address which does not point to the table you wanted it to point to.
- you can store (U)ROWIDS to rows in different tables in the same column, which would make accessing the data pointed to by the (U)ROWID quite complex and error-prone.
- you cannot move a table which contains a column of (U)ROWID type to another computer, neither by CREATE TABLE AS SELECT nor by IMPORT/EXPORT. Though the operation may success, no matter which variant you choose, all your ROWID values will be invalid because the data moved will not have the same address on the target computer.

Reserved syntax notation for VARCHAR2 parameters

Package:	Severity:	Category:	Code Element:
Built-in	Major	Program Structure	Parameter

Rationale:

Reasoning:

A quote from Oracle Documentation:

The specifications of many packages and types that Oracle Database supplies declare formal parameters with this notation:

- i1 IN VARCHAR2 CHARACTER SET ANY_CS
- i2 IN VARCHAR2 CHARACTER SET i1%CHARSET

Do not use this notation when declaring your own formal or actual parameters. It is reserved for Oracle implementation of the supplied package types.

Avoid Cartesian queries, consider adding a WHERE clause or convert to an ANSI INNER JOIN

Package:	Severity:	Category:	Code Element:
Built-in	Major	Program Structure	SQL

Rationale:

Reasoning:

A CARTESIAN PRODUCT is a join between two sets where each element from set one is paired with each element of set 2. In the case of row sets that means each row from row source 1 is paired with each row for row source 2. If both row sources contain 1000 rows, then the result of a CARTESIAN PRODUCT between them will contain one million rows.

CARTESIAN joins are rare. Normally a CARTESIAN join is an indication of a missing join condition. If indeed a true CARTESIAN PRODUCT is desired, then the recommended approach is to use ANSI syntax that allows the code component to explicitly declare the CARTESIAN join. This is done by using a CROSS JOIN, CROSS JOIN being the key words for this type of join.

Use the CASE expression instead of the DECODE function

Package:	Severity:	Category:	Code Element:
Built-in	Minor	Readability	SQL

Rationale:

Reasoning:

DECODE is an Oracle function (only usable from SQL in Oracle) which allows for the simple translation of one expression into another. It can take many such expressions and translate them, doing that translation for the first match it finds. From Oracle 8i onwards, the CASE statement was introduced, including the SEARCHED CASE variation. It is generally recommended to use CASE now instead of DECODE, since CASE is supported in both SQL and PL/SQL, so the same code can be deployed to both environments. CASE is much more powerful than DECODE. The CASE expression can do all that DECODE does and a lot of other things in addition.

However, this recommendation is primarily for NEW DEVELOPMENT, not maintenance. It is not recommended that maintenance teams undertake the task of replacing DECODE with CASE, even when they are already upgrading a code component for some other reason. There is simply not enough benefit to warrant the additional testing required, unless the specific lines with DECODE are already party of the maintenance activity and would thus require testing anyway.

Predefined identifier redeclared

Package:	Severity:	Category:	Code Element:
Built-in	Critical	Maintainability	PL/SQL

Rationale:

Reasoning

An identifier is just a name. Redeclaring an identifier leads to ambiguity and dramatically changes the code behavior.

There is an article in Oracle Documentation that recommends against redeclaring predefined exceptions and describes the consequences. However, redeclaring identifiers has a wider scope involving any Oracle reserved word.

Unreachable code after RETURN, EXIT, RAISE or GOTO statements

Package:	Severity:	Category:	Code Element:
Built-in	Major	Program Structure	PL/SQL

Rationale:

Reasoning:

DEAD ZONES are sections of code which will never be executed. Aside from use scenarios surrounding testing, dead zones should not appear in production code. Dead Zones cause confusion, potentially lead to errors later during code maintenance, and often are an indication of logic that is more complex than it needs to be. Dead Zones should be removed from production code.

In cases where DEAD ZONES must be maintained, ample documentation should accompany the DEAD ZONE to explain why it is still there.

Assign sequence pseudocolumns directly to variables instead of selecting them from the DUAL table

Package:	Severity:	Category:	Code Element:
Built-in	Major	Maintainability	SQL

Rationale:

Reasoning:

One example of this is fetching of sequences. In version 11 Oracle improved handling of sequences in PL/SQL. Starting with version 11, you can assign the CURRVAL and NEXTVAL of a sequence directly to variables. In earlier versions, you had to use a SELECT ... INTO statement to achieve the same effect. In PL/SQL, direct assignment should now be used instead of SELECT INTO.

Avoid unnecessary references to schema names

Package:	Severity:	Category:	Code Element:
Built-in	Minor	Program Structure	SQL

Rationale:

Reasoning:

- A reference to an object in the same schema is normally made WITHOUT using a SYNONYM and WITHOUT using SCHEMA.OBJECT notation.
- A reference to an object in another schema is made using SCHEMA.OBJECT notation, or using a local (in the working schema) WRAPPER VIEW, or by using a local SYNONYM.
- WRAPPER VIEWS may also choose to use SYNONYMS to reference “other schema” objects, indeed this is recommended.
- Procedural code objects (packages / procedures / functions / triggers / java source / types) will follow the same rules when they reference objects.

Synonyms are a valuable feature for avoiding the use of SCHEMA.OBJECT notation in code. The synonym was created so that applications could be agnostic about the owner, name, and location of objects they reference. This provides freedom for these objects to change any of these three aspects of an object. Thus, when a table changes its name, or moves to another schema, or is made remote as part of some database consolidation activity, all code that references that object through a schema is protected from the change. The synonym can be modified to account for the change, and code objects can remain unchanged.

The INSTRUMENTATION percentage of a stored program is below the specified minimum

Package:	Severity:	Category:	Code Element:
Built-in	Trivial	Maintainability	Subprogram

Rationale:

Reasoning

The term instrumentation refers to the ability to monitor or measure the level of a product's performance and to diagnose errors. Developers implement instrumentation in the form of code instructions. Instrumentation should be an integral part of the code. Code instrumentation helps determine logical, performance and other problems and fix them. You can use tracing by means of DBMS_MONITOR and DBMS_APPLICATION_INFO to find time-consuming procedures in an application or produce a debug output using the DBMS_OUTPUT package. DBMS_UTILITY.GET_TIME can be used to isolate performance problems. But the best way is to write the enhanced logging functionality keeping in mind the problem of logging multiple sessions at the same time.

Use an explicit ANSI JOIN instead of an implicit join

Package:	Severity:	Category:	Code Element:
Built-in	Trivial	Readability	SQL

Rationale:

Reasoning:

This is one of those hotly debated issues. There is in fact no inherent superiority of Oracle Syntax over ANSI JOIN syntax; nor is there a superiority of ANSI JOIN syntax over Oracle syntax. Supporters of ANSI syntax say that ANSI is easier to read. Supporters of ORACLE syntax say that ANSI is not easier to read, but that Oracle syntax is easier to read. Other reasons why ANSI syntax is sometimes touted as superior is for its support of FULL OUTER JOIN, and that ANSI syntax is common across all databases that support SQL which makes transitioning between database systems easier.

Here are some safe recommendations:

1. Unless you are a software shop, writing code for multiple databases, go ahead and use Oracle syntax if you want to. Join syntax actually plays very little role in transitioning between databases when compared to the plethora of other Oracle specific components you will need to work around.
2. If your company has an explicit mandate to use ANSI SQL for new development, go ahead, so you can keep your job.
3. NEVER MIX BOTH SYNTAXES IN THE SAME SQL STATEMENT (this gets confusing and exposes you to bugs).
4. If you have legacy code, do not bother to rewrite it just to get out of Oracle syntax.

Replace column numbers with column names or aliases in the ORDER BY clause

Package:	Severity:	Category:	Code Element:
Built-in	Major	Program Structure	SQL

Rationale:

Reasoning:

ORDER BY allows for columns to be referenced by column alias, or by column position in the select list. Many professionals use the shortcut of using the column position because it is quick. This is fine for test work, but should not be used in code that is destined for production. Indeed, it should not generally be used at all. There are few reasons why:

- Using the column position in the select list renders the query subject to side effects when the select list is changed. This will happen if a developer forgets to change the ORDER BY clause after adding or removing or moving columns in the select list.
- Using the column position obscures what column is doing the sorting and so the SQL fails to convey potentially significant information about itself.

Explicitly specify the ASC keyword in the ORDER BY clause

Package:	Severity:	Category:	Code Element:
Built-in	Trivial	Readability	SQL

Rationale:

Reasoning:

Ascending sorting order is implied when the ASC keyword is omitted. The explicitly specified ASC keyword improves the readability of a complex ORDER BY clause.

Specify the AS keyword for an alias of the column expression

Package:	Severity:	Category:	Code Element:
Built-in	Trivial	Readability	SQL

Rationale:

Reasoning:

SQL allows the creation of aliases for column names in SELECT statements. Newer versions of the SQL standard allow such alias names to be introduced by the keyword AS, while older versions of the standard did not support the keyword AS at all.

When your version of Oracle supports the use of the keyword AS, we recommend that you use this version. It makes the intended meaning much clearer, and reduces the chance of involuntarily changing the meaning of a SELECT list by missing to write a comma in the SELECT list.

Avoid DDL statements except partition maintenance operations

Package:	Severity:	Category:	Code Element:
Built-in	Major	Program Structure	PL/SQL

Rationale:

Reasoning:

DDL from inside PL/SQL has many issues, but three stand out.

- DDL from inside PL/SQL can cause recompilation problems. It is possible to create situations where your code locks itself.
- DDL from inside PL/SQL may be an indication of an incomplete database design. Incomplete data models ultimately lead to problems later (maybe years later). A contemplation of solution design could yield significant benefits.

- DDL from inside PL/SQL splits transactions because DDL always does an implied COMMIT. When faced with DDL from inside PL/SQL a consideration of transaction design is in order.

Do not use compound queries with more than two UNION operators

Package:	Severity:	Category:	Code Element:
Built-in	Minor	Program Structure	SQL

Rationale:

Reasoning:

UNION is a valuable relational operation, as is its sister operation UNION ALL. But UNION and UNION ALL can also be easily abused. Though multiple UNION and particularly UNION ALL queries are not necessarily wrong, they are often an indication of inefficient code, sometimes very inefficient code. It pays to re-evaluate such queries as many times, expensive scans and expensive join chains can be removed from a query with multiple UNION operations, given a bit of thought. We recommend that a query with more than two UNION operations as being something to re-evaluate.

Features that are effective in reducing UNION and UNION ALL queries include the following (often in combination).

- NVL
- CASE
- OUTER JOIN
- ANALYTICS
- AGGREGATION

Avoid explicit DISTINCT in a subquery of the IN condition and UNION set operator

Package:	Severity:	Category:	Code Element:
Built-in	Major	Program Structure	SQL

Rationale:

Reasoning:

In nine cases out of ten, use of DISTINCT is an error. Semantically, the need for DISTINCT should be obvious. Whenever DISTINCT is coded in a place where it is not needed, it suggests someone does not fully understand the nature of the problem they are trying to solve. Whenever there is doubt about having the correct understanding of the problem space, it is wise to review the situation.

FULL OUTER JOIN may return very large result set. Consider using another technique.

Package:	Severity:	Category:	Code Element:
Built-in	Minor	Maintainability	SQL

Rationale:

Reasoning:

FULL OUTER JOIN is rare. As such this rule exists to remind us to review any use of FULL OUTER JOIN for clarity, necessity, and correctness; as use of FULL OUTER JOIN is often an indication that a developer has not fully understand the nature of a requirement, or the data model of a database.

Consider for example the BAD/GOOD code examples presented in a moment. They join DEPT and EMP using a FULL OUTER JOIN. What then is the KEY of the relation that is produced? Said another way: What makes the rows coming out of the FULL OUTER JOIN unique? Having answered this, can you explain: What is the "thing"? What is the single BUSINESS ENTITY these rows map to?

The entity produced almost never maps to a BUSINESS ENTITY of any kind. At best it is an ARTIFICIAL ENTITY of convenience. At worst, it is a mashup of multiple BUSINESS ENTITIES into a single result masquerading as a relation, but with no UNIQUENESS that clearly identifies what "BUSINESS THING" the result represents. It is true that SQL allows for the generation of row sets which are not true relations, but normally these are still mappable to a single BUSINESS ENTITY that can be easily understood. Not so with a FULL OUTER JOIN. Though, as long as two true relations are the inputs to the FULL OUTER JOIN, the result of the operation is technically also a relation from a theoretical perspective because it provides relational closure, FULL OUTER JOIN is a good example of how technical correctness does not always map to business correctness.

Group standalone procedures and functions in packages

Package:	Severity:	Category:	Code Element:
Built-in	Minor	Program Structure	PL/SQL

Rationale:

Reasoning:

This rule finds any standalone PROCEDUREs and FUNCTIONs (with optional CREATE), as such program units should not exist outside a package.

Standalone procedures and functions are rarely necessary. You should always ask yourself: What does this function do? Why is it not in a package? Do I need a new package? The reasons most standalone procedures and functions exist is because people are in a hurry, and deciding on how to package up a procedure/function for future use and maintenance takes thought and that takes time.

One problem with standalone procedures and functions is that they cannot be overloaded. Placing them in a package allows for smarter code because it allows you to overload.

Another reason to use packages is that packages separate specification from implementation. Since a package has two pieces, the specification which says what it

exposes to the outside world as callable entry points, and the body which provides the implementation details of the code. A specification once landed on, rarely changes, but code details change all the time. The separation of specification from executable code, means that changes to the code (which happen all the time) do not cause invalidations of linked database objects and this is good for performance and more importantly simplifies release processes.

Limit use of subqueries in SELECT clauses

Package:	Severity:	Category:	Code Element:
Built-in	Trivial	Readability	SQL

Rationale:

Reasoning:

A subquery in a SELECT clause is known as a SCALAR SUBQUERY. It is called SCALAR SUBQUERY because it must by design return at most one value. If it returns more than one value, it will generate a TOO MANY ROWS error. Because of this, it is advisable to know for certain that at most one value will be returned. This should be verifiable by looking at constraints that match the correlating predicates in the subquery, or by some construct in the subquery which cannot return more than one row (for example AND ROWNUM = 1).

Optimizer behavior for SCALAR SUBQUERIES is highly dependent upon constraint declarations as well. If the optimizer chooses to use NESTED LOOPS join to do the SCALAR, this forces the join order of the query such that the main query must be executed first. Thus, there are lost opportunities for optimization that might have been better choices, if the SCALAR SUBQUERY could have instead been joined directly to the main query (MERGED in a rewrite). So, JOIN ORDER can be impacted by use of SCALAR SUBQUERIES.

The good news is that newer versions of the database (11g/12c) are much smarter than previous releases, and do a better job of finding the more performant method. But there are still plenty of gaps. If you find your query is doing NESTED LOOPS join to get at tables in the SCALAR SUBQUERY, evaluate its correctness for the alternative direct join or outer join.

Explanation:

This rule refers to use of SCALAR SUBQUIERIES in select lists. In older versions of Oracle, a SCALAR SUBQUERY was evaluated one row at a time. This resulted in either a NESTED LOOPS join or FILTER operation in the query plan which could be very inefficient for large numbers of rows. As of 11.2.0.4, Oracle can un-nest these constructs into joins.

Avoid more than 1000 items in an IN clause list

Package:	Severity:	Category:	Code Element:
Built-in	Critical	Program Structure	SQL

Rationale:

Reasoning:

Though syntactically correct, the existence of a large list of items in a query, is often an indication of a missing table. There is presumably a reason for the limit of 1000 items in an IN list that exists in Oracle afterall.

Personally, I have never had the need for a large IN list of items. If you find yourself using one, ask the following questions:

- Where did this data come from? A table? If so why am I not referencing the table directly?
- Would it make sense to have the data model upgraded to account for the "thing" that this list of items represents? Then that new table can be used in the query.
- In PL/SQL, a collection could be loaded with the IN-LIST VALUES and then the query joined to the collection. This circumvents the 1000 item limit of IN lists to some degree.

There is a clear philosophical attitude at work here. It is about RESPECTING DATA. Just creating a large IN-LIST does not pay attention to what the data is. Asking the above questions reflects a desire to make the database speak about the data, by having it retain the meaning of the data, giving the entity, the data represents, an identity. A large IN-LIST may for sure be nothing more than a convenient way to generate code without consideration of changes to a data model; and this is appealing in a world that seems to favor speed of delivery over completeness and quality. Acknowledging that there must be balance, you still must decide which side of this philosophical fence you will live on.

One possible exception to the rule against a large IN-LIST may be code generating apps that generate code on the fly with unique queries. But even this has issues.

1. As noted above, there is a 1000 element limit. Going beyond this will cause an app to fail.
2. Dynamically generated code usually creates many variations of the same SQL which floods the cursor cache and slows the system down. In extreme cases this has been known to causes systems to hang.
3. Dynamic SQL is always an invitation to SQL INJECTION which is why dynamic SQL should be avoided unless absolutely necessary.

Too many OR operators at the same level of the WHERE clause

Package:	Severity:	Category:	Code Element:
Built-in	Minor	Readability	SQL

Rationale:

Reasoning:

This is a somewhat arbitrary rule. Its purpose is of course to bring attention to complex conditions so that they may be potentially refactored into something easier to understand. Unfortunately, options are somewhat limited.

The variable was not modified anywhere in the code. Declare it as a constant.

Package:	Severity:	Category:	Code Element:
Built-in	Minor	Readability	PL/SQL

Rationale:

Reasoning:

There are two elemental issues with hard-coded values found in PL/SQL;

1. If a constant value changes, multiple instances of the same constant must be found and changed also.
2. A value which is a constant but which is not labeled as such, is at risk of being modified accidentally.

Consider creating a new function instead of using a parameter declaration with the DEFAULT value

Package:	Severity:	Category:	Code Element:
Built-in	Minor	Maintainability	PL/SQL

Rationale:

Reasoning:

This rule is a specific variation of a more general problem, involving the approach a software shop uses to release code. Any shop that wants to be 100% up, or any shop that otherwise wants to do HOT RELEASES (release code while people are on the system using that code), will have to contend with the error ORA-04061: existing state of package body has been invalidated, and its associated variations and related errors. Any time new code is released to an Oracle system (packages / procedures / functions / types / views), live sessions must stop using the old version of the code and start using the new version. Normally Oracle manages this behind the scenes without any interruption to active sessions. All objects that were made invalid due to a change will be automatically re-compiled when released, and all sessions will acquire these re-compiled versions the next time they use them. But… there are some programming practices that negate this automatic process.

PACKAGE GLOBAL VARIABLES ARE THE PRIMARY REASON FOR THESE ORA-04061 AND ASSOCIATED ERRORS.

The issue arises in part with how Oracle decided to differentiate between the significance of program logic, and the significance of session state. Oracle decided that Program Logic is not significant in the sense that new program logic can be re-compiled and then made available for use at any time, since no one is relying on that program logic if they are not in the middle of using it. Thus, Oracle decided that there was no need to tell people when a new version of code had been made available. But session state is different. Session state refers to data that is being kept live between invocations of program logic. Oracle decided that session state is important because subsequent invocations of code that reference data values saved in session state, will pick up where they left off, using the values saved in this session state. Therefore, for any package that created or modified session state, Oracle must keep the memory structure of that package in memory until the

session ends, because Oracle is expecting that the same package will be called again and want to use that session saved session state, to pick up where it left off.

PACKAGE GLOBAL VARIABLES are the primary way that session state is created.

But what happens when a package that created session state for some session, gets upgraded during a HOT RELEASE? Oracle must recompile the package and each session that subsequently uses the package must acquire the new code. But this also means that the current session state for this package must be thrown out for every session that has session state for that package, since the memory structure for the package will be different after the re-compile. This is a problem because it means that the next use of the package cannot pick up where it left off by using the saved session state that was left by the previous use of the package, because it no longer exists (it was tossed out because of the re-compile).

Since the expectation of session state is that it will persist for the life of the session, and since a re-compile has now negated that promise, Oracle tells the user that the promise of persisted session state has been broken for one of their packages. Hence the ORA-04061.

So, what do you do? Well there are three ways to deal with this issue:

1. Do not be a 100% UP service center. Instead, be 99% UP and adopt a 1% down-time release strategy the includes customer notifications, and kicking active user sessions off the database, and locking down the database; before the release.
2. DO NOT USE PACKAGE GLOBAL VARIABLES. This one change alone will eliminate 99% of occurrences of these errors. Instead use one of several alternative persistence mechanisms to persist data that must live for the duration of a session. These include:
    1. SYSCONTEXT
    2. GLOBAL TEMPORARY TABLES
    3. ADVANCED QUEUEING
3. Employ a robust error handling system at the client level, that traps for these errors, and gracefully handles resubmitting of procedure calls.

All three of these require some thought, but if you want to avoid these kinds of errors, they are what you must do. As you will see in the upcoming example, there is a lot of work needed to persist state for a session without using PACKAGE GLOBAL VARIABLES. This is one reason why people will use them, and because they do not know of the problem. But this work it is what is needed to avoid the issue of package invalidation.

Do not mix ANSI joins and Oracle joins in a single query

Package:	Severity:	Category:	Code Element:
Built-in	Major	Readability	SQL

Rationale:

Reasoning:

There are three problems with mixing JOIN syntax.

1. It is confusing. This alone should be sufficient cause to avoid the practice, and writing code that is easy to understand and maintain is an inherent part of good programming.
2. It can lead to ORA-600 errors due to bugs in Oracle’s optimizer and query rewrite system.

3. It can lead to incorrect answers due to bugs in Oracle's optimization and query rewrite system.

Nested comments are not supported and should be avoided

Package:	Severity:	Category:	Code Element:
Built-in	Trivial	Maintainability	Comment

Rationale:

Reasoning:

A comment opening character sequence (/*) is detected within the comment. As PL/SQL does not support nested comments, this is a bug, which might prove hard to detect. This situation may arrive when you comment out entire blocks of code.

When commenting out blocks of code, make sure that you remove all comments which exist in that block by either

1. removing all existing comment delimiters (/* and */) from that block
2. putting a space between the / and the * in comment delimiters
3. replacing the nested comments by (a series of) single-line comments (--)

Write an exception handler for the associated exception name instead of the error number check

Package:	Severity:	Category:	Code Element:
Built-in	Major	Readability	PL/SQL

Rationale:

Reasoning:

Declaring exceptions is a way to dramatically improve the natural documentation of a PL/SQL unit, for those situations where such documentation is needed most. A well picked name makes it easier to understand what might have gone wrong. It also documents up front what exceptions anyone reading the code should expect to see handled in the code.

Store a dynamic SQL statement in a local variable and execute the statement from it

Package:	Severity:	Category:	Code Element:
Built-in	Minor	Maintainability	SQL

Rationale:

Reasoning:

When you want to dynamically execute a SQL statement by using the EXECUTE IMMEDIATE PL/SQL statement, it's advantageous to store the statement in a local variable and execute from that variable. One of the advantages this gives you is easy access to the statement text, which if there is an error, will be very useful.

A function call between a DML statement and an implicit cursor attribute test may cause incorrect attribute test results

Package:	Severity:	Category:	Code Element:
Built-in	Major	Program Structure	Cursor

Rationale:

Reasoning:

PL/SQL provides cursor attributes that allow code to inspect the state of a cursor at runtime; the most commonly used cursor attribute being %NOTFOUND used to detect EOF (end-of-file) (aka no more rows) in the cursor. These cursors can be EXPLICIT using the name of a specific cursor, or IMPLICT referencing the last cursor operated on. Implicit cursor attributes can change every time any cursor (implicit or explicit) is referenced.

If then, a procedure of function call is made between the time an operation is done on an implicit cursor, and the time its attribute is checked, it is possible that the value returned by the implicit cursor attribute will not be related to the cursor the code really wanted to check, since the called procedure or function may have performed another cursor operation without the calling code knowing about it. This will lead to incorrect results.

To avoid this issue:

- only reference cursor attribute values immediately after executing the associated statement
- or save the value of the cursor attribute into a variable immediately after executing the associated statement and refer to the saved value later

The cursor state may be unexpected

Package:	Severity:	Category:	Code Element:
Built-in	Trivial	Program Structure	Cursor

Rationale:

Reasoning:

PL/SQL allows for cursors to be obtained from code OUTSIDE the current code component. CURSOR VARIABLES are an excellent example. These can be acquired from general use routines. But when a cursor is obtained in this manner, we have no direct control over how this cursor us manipulated, and PL/SQL allows for INVALID CURSORS to be passed around between routines.

Because of this, it is always good practice to check the state of a PACKAGE GLOBAL VARIABLE that contains a cursor, or any other CURSOR VARIABLE obtained from outside local code, before using it. For example, do not assume a valid open cursor will always be returned from a function or procedure that returns a cursor variable, or that a "fetcher" routine has closed automatically closed a cursor after receiving a NO_DATA_FOUND; or the opposite, that it will always close a cursor when it receives a NO_DATA_FOUND.

The rule is violated when the cursor state is unknown in the current scope.

Examples:

- An OPEN or FETCH statement without %ISOPEN checks for the cursor passed by a parameter and wasn't explicitly closed in the current scope.
- A CLOSE statement without %ISOPEN checks for the cursor passed by a parameter and wasn't explicitly opened in the current scope.
- A FETCH or CLOSE statement for a cursor that was opened by an OPEN statement embraced in a conditional statement.

The cursor is already open

Package:	Severity:	Category:	Code Element:
Built-in	Critical	Program Structure	Cursor

Rationale:

Reasoning:

A cursor can be OPEN or CLOSED. Only a cursor which is OPEN can have FETCH and CLOSE operations performed upon it, and only a cursor which is NOT OPEN can have an OPEN operation performed on it. If there is a possibility that the state of a cursor may not be what is expected, then fault tolerant code can be written by checking the state of the cursor before using it. In the case of an OPEN, it may be judicious to write a wrapper OPEN routine that checks the state of the cursor first and performs the correct action.

The cursor is not open

Package:	Severity:	Category:	Code Element:
Built-in	Critical	Program Structure	Cursor

Rationale:

Reasoning:

Rule is violated when you try to FETCH from, to CLOSE or to test the attribute of the cursor that is not opened.

The cursor may remain opened when it goes out of the scope

Package:	Severity:	Category:	Code Element:
Built-in	Major	Program Structure	Cursor

Rationale:

Reasoning:

Every explicit cursor that was once opened should be closed in order to release the memory allocated by the cursor. When the cursor is declared not in the package (locally), the database will automatically close it when the PL/SQL block ends execution and the cursor goes out of scope. But it's still a good programming style to close cursors in your code.

SELECT INTO is a better choice to fetch one row than CURSOR FOR LOOP

Package:	Severity:	Category:	Code Element:
Built-in	Trivial	Readability	PL/SQL

Rationale:

Reasoning:

One rule of good coding is to DO WHAT IS EXPECTED. Following this rule helps create code that is easier to understand because it follows a more normal flow. A CURSOR FOR LOOP implies the need to fetch multiple rows. But if the need is to fetch only one row, the a SELECT INTO is a better choice, because everyone expects a SELECT INTO to fetch only one row.

Consider that the choice of fetch mechanism conveys information to anyone looking at the code. Choosing the wrong mechanism results in code conveying incorrect information about itself. Using a CURSOR FOR LOOP implies the need to fetch multiple rows, which is incorrect information if the need is to fetch only one row. Choosing strategies that improve the ability of code to convey a better understanding of itself to whoever is looking at it, is always a good thing. So when only one row is needed, use SELECT INTO and not CURSOR FOR LOOP.

Always list columns in the INSERT statement

Package:	Severity:	Category:	Code Element:
Built-in	Major	Maintainability	SQL

Rationale:

Reasoning:

When INSERTING data, always list the columns receiving this data. This does two things for you:

1. It provides better documentation of what the receiving items are, hopefully helping to reduce the number of incorrect assignments due to column mismatch.

2. It insulates code form changes to underlying data structures.

Using ROWID or UROWID may cause data integrity problems

Package:	Severity:	Category:	Code Element:
Built-in	Trivial	Maintainability	DataType

Rationale:

Reasoning:

Starting with Oracle database version 8.1, ROWIDs can change that makes referencing ROWIDs unreliable in many cases. If row movement is enabled and a row's partition key is changed then the row will be moved and it's ROWID changes. Many data segment physical storage operations such as SHRINK SPACE or MOVE will cause row movement from one physical data block to another and the ROWID will change.

In addition, PL/SQL code written using ROWID or UROWID becomes problematic for transfer to other SQL databases.

Consider migrating from LONG columns to LOB

Package:	Severity:	Category:	Code Element:
Built-in	Minor	Program Structure	DataType

Rationale:

Reasoning:

1. LONG exists only for backward compatibility and should not be used.
2. LONG is not supported by many common features of the database and thus causes problems in otherwise simple data operations.
3. Use VARCHAR2 for items <= 4000, in 12c (if you are willing to set parameter max_string_size=EXTENDED) VARCHAR2 for items <32767 , otherwise use CLOB.

Avoid use of CHAR; consider VARCHAR2

Package:	Severity:	Category:	Code Element:
Built-in	Minor	Program Structure	DataType

Rationale:

Reasoning:

1. Due to right padding of spaces for a CHAR data type, use of CHAR leads to unexpected results in string comparisons.

2. Implicit type conversion of non-string values to string format almost always results in VARCHAR2 which adds comparison problems with CHAR semantics.

Avoid use of VARCHAR; consider VARCHAR2

Package:	Severity:	Category:	Code Element:
Built-in	Minor	Program Structure	DataType

Rationale:

Reasoning:

Do not use the VARCHAR data type. Use the VARCHAR2 data type instead. Although the VARCHAR data type is currently synonymous with VARCHAR2, the VARCHAR data type is scheduled to be redefined as a separate data type used for variable-length character strings compared with different comparison semantics.

A subtype can be declared for variables and parameters with the same type

Package:	Severity:	Category:	Code Element:
Built-in	Trivial	Maintainability	DataType

Rationale:

Reasoning:

The basic rules are:

1. When a variable is intended to hold a value from a column, use %TYPE.
2. Otherwise, use a subtype declaration and pattern the variable after the subtype.

Using subtype effectively gives you data model style control over variable types.

It makes it easier to change all variables patterned after the same subtype. For example, maybe your dollar amounts need to be increased from 12,2 to 15,2.

Specify CHAR, VARCHAR, VARCHAR2, NCHAR, NVARCHAR2 maximum size

Package:	Severity:	Category:	Code Element:
Built-in	Trivial	Program Structure	DataType

Rationale:

Reasoning:

It is good practice to specify the maximum size of the character variable even for the types for which it not mandatory because it is convenient to make validation for an input data. Oracle issues an error "character string buffer too small" if you store a character string whose size exceeds the maximum size of the VARCHAR2 column.

You must specify the maximum length of the VARCHAR2 variable when you define it.

The whitespace percentage of a stored program is less than specified minimum

Package:	Severity:	Category:	Code Element:
Built-in	Minor	Readability	Subprogram

Rationale:

Reasoning:

This rule monitors the existence of whitespace in a PL/SQL code unit. Whitespace includes the number of blank lines that separate components. Ultimately whitespace is a very useful method of improving the readability of a program unit.

In using blank lines for grouping code for readability, there is basic consideration to create smaller logical units of singular ideas.

The amount of whitespace is a matter of programming style. Adding blank lines improves legibility up to a point. Too many blanks will make reading harder, though, as one has to scroll through more pages than otherwise necessary.

Avoid use of DBMS_PIPE in RAC environment

Package:	Severity:	Category:	Code Element:
Built-in	Major	RAC / Exadata	PL/SQL

Rationale:

Reasononing:

If an application uses the DBMS_PIPE feature, it needs to be ensured that both producer and consumer are connected to same instance. This happens naturally in a single instance database but in the RAC case when there are multiple instances, the application needs to ensure that the two sessions connect to the same RAC instance. DBMS_AQ is better alternative to DBMS_PIPE.

Use gv$ system views instead of v$ ones in RAC environment

Package:	Severity:	Category:	Code Element:
Built-in	Major	RAC / Exadata	SQL

Rationale:

Reasononing:

If application reads information from any of the dynamic views (v$ tables), in RAC environment they will need to read the equivalent gv$ table to get a global view. Most applications do not have to do this at all but many scripts to manage the Database need to consider this.

Consider moving from DBMS_ JOB to DBMS_SCHEDULER in RAC environment

Package:	Severity:	Category:	Code Element:
Built-in	Major	RAC / Exadata	PL/SQL

Rationale:

Reasononing:

If application is using DBMS_JOB, consider moving to using the dbms_scheduler feature. If not possible, ensure that the resources required by the job when it starts are available on all instances of the DB (including ones where the service may not run). Use instance affinity feature to bind a job to a specific instance if a job needs to run from a specific instance.

Setting a global application context value in an Oracle RAC environment has performance overhead

Package:	Severity:	Category:	Code Element:
Built-in	Minor	RAC / Exadata	PL/SQL

Rationale:

Reasoning:

Global Application Context does not work correctly in Oracle RAC versions up to 11.1 (inclusive) because context values are not synchronized between cluster nodes. Starting from version 11.2 developer should be aware that setting a global application context value in an Oracle RAC environment has performance overhead of propagating the context value consistently to all Oracle RAC instances. Consider using Advanced Queuing mechanism instead.

Intensive use of DBMS_ALERT package can reduce RAC performance

Package:	Severity:	Category:	Code Element:
Built-in	Minor	RAC / Exadata	PL/SQL

Rationale:

Reasoning:

Intensive use of DBMS_ALERT package makes DBMS_ALERT_INFO table “hot” in RAC environment and can limit RAC performance. Performance optimization techniques should be applied to eliminate “hot block” contention.

Using the DBMS_SCHEDULER package subprograms commits transactions implicitly

Package:	Severity:	Category:	Code Element:
Built-in	Minor	Program Structure	PL/SQL

Rationale:

Reasoning:

Using the CREATE_JOB and CREATE_JOBS subprograms of the DBMS_SCHEDULER package may break the transaction logic of a PL/SQL procedure, and it will not be possible to undo changes.

SELECT statement with GROUP BY clause should contain an ORDER BY clause

Package:	Severity:	Category:	Code Element:
Built-in	Minor	Program Structure	SQL

Rationale:

Reasoning:

One of the major problems when migrating from a 9.2.0.x (or earlier) database to a 10.1.0.x (or newer) database is that the behaviour of the GROUP BY clause changed. People assumed that the output of a select statement was always sorted by the group by items. This was never guaranteed by Oracle, but it seemed to work.

This behaviour changed with Oracle 10. If you need the output of a select-statement containing a group by clause to be sorted, you have to code the required order by clause. As this causes major problems during migration, we should be able to alert customers about occurrences of this situation.

Avoid double-quoted identifiers

Package:	Severity:	Category:	Code Element:
Built-in	Minor	Maintainability	PL/SQL

Rationale:

Reasoning:

Don't use double-quoted identifiers, and always use names that are valid unquoted in order to avoid dependency on a database implementation case and provide "one universal version" of the identifier. It helps avoid errors introduced by developers who are unaware of or unwilling to follow case conventions. Using double-quoted reserved words and keywords as identifiers, using symbols allowed in quoted identifiers that are not allowed in unquoted identifiers, having two different variables of the same name with just different characters cases is not desirable and it is not good for readability.

OTHERS exception handler does not end in RAISE or RAISE_APPLICATION_ERROR

Package:	Severity:	Category:	Code Element:
Built-in	Major	Program Structure	Exception

Rationale:

Reasoning:

Suppressing of all exceptions is a very wrong way to deal with them. All unexpected exceptions will be not propagated if someone inherits this code. All code that contains a WHEN OTHERS exception handler must also include RAISE or RAISE_APPLICATION_ERROR to re-raise the exception to be a bug in order to keep atomicity of a PL/SQL block.