16 Using the PL/SQL Compiler

✅Describe the PL/SQL compiler and features

PL/SQL optimizer: 12.1

Optimizer

- Prior to Oracle Database 10g release 1, the PL/SQL compiler translated your source text to system code without applying many changes to improve performance
- Now, PL/SQL uses an optimizer that can rearrange code for better performance
- The optimizer is enabled by default (see plsql_optimize_level to change this)
- In rare cases, if the overhead of the optimizer makes compilation of very large applications too slow, you can lower the optimization by setting the compilation parameter `PLSQL_OPTIMIZE_LEVEL=1` instead of its default value 2
- In even rarer cases, PL/SQL might raise an exception earlier than expected or not at all. Setting `PLSQL_OPTIMIZE_LEVEL=1` prevents the code from being rearranged

Subprogram Inlining

- One optimization that the compiler can perform is **subprogram inlining**
- This replaces a subprogram invocation with a copy of the invoked subprogram (if both are in the same program unit)
- To allow subprogram inlining, accept default of `PLSQL_OPTIMIZE_LEVEL` (2) or set it to 3.
- With default `PLSQL_OPTIMIZE_LEVEL` you must specify each subprogram to be inlined with the `INLINE` pragma:

```
PRAGMA INLINE (subprogram, 'YES')
```

- If subprogram overloaded, then the pragma applies to every subprogram with the same name.
- With `PLSQL_OPTIMIZE_LEVEL=3`, the compiler seeks opportunities to inline subprograms by itself. You need not specify subprograms to be inlined. However, you can use the `INLINE` pragma (with the preceding syntax) to give a subprogram a high priority for inlining, and then the compiler inlines it unless other considerations or limits make the inlining undesirable
- If a particular subprogram is inlined, performance almost always improves. However, because the compiler inlines subprograms early in the optimization process, it is possible for subprogram inlining to preclude later, more powerful optimizations.
- If subprogram inlining slows the performance of a particular PL/SQL program, then use the PL/SQL hierarchical profiler to identify subprograms for which you want to turn off inlining
- To turn off inlining for subprogram, use the pragma:

```
PRAGMA INLINE (subprogram, 'NO')
```

- The `INLINE` pragma affects only the immediately following declaration or statement, and only some kinds of statements When `INLINE` pragma immediately precedes declaration, it affects:
    - every invocation of the specified subprogram in that declaration
    - every initialization value in that declaration except te default initialization values of records
- When `INLINE` pragma immediately precedes one of these statements: `Assignment, CALL, Conditional, CASE, CONTINUE WHEN, EXECUTE IMMEDIATE, EXIT WHEN, LOOP, RETURN` the pragma affects every invocation of the specified subprogram in that statement
- The `INLINE` pragma does not affect statements that are not in the preceding list
- Multiple pragmas can affect the same declaration or statement. Each pragma applies its own effect to the statement. If `PRAGMA INLINE(subprogram,'YES')` and `PRAGMA INLINE(identifier,'NO')` have the same subprogram, then 'NO' overrides 'YES'
- One `PRAGMA INLINE(subprogram,'NO')` overrides any number of occurrences of `PRAGMA INLINE(subprogram,'YES')`, and the order of these pragmas is not important
- > If two or more pragma `INLINE` have the same subprogram, then 'NO' will override any other occurrence of pragma `INLINE` 'YES' on the same subprogram.

View compiler settings

- To see information about the compiler settings for stored objects available to the current user, query the data dictionary view `ALL_PLSQL_OBJECT_SETTINGS`:

```
SELECT * FROM ALL_PLSQL_OBJECT_SETTINGS [WHERE owner = your_schema_name];
```

- This query will return the owner, name and type of the object as well as the compiler initialization parameters the object was created with.

✅Use the PL/SQL compiler initialization parameters

PL/SQL compilation parameters: 1.3.2

PLSCOPE_SETTINGS

Using PL/Scope: READ ONLY

PLSQL_CCFLAGS

PLSQL_CODE_TYPE

PLSQL_OPTIMIZE_LEVEL

Compiling units for native execution: 12.10

Summary of the compilation parameters:

- **PLSCOPE_SETTINGS**: controls compile-time collection, cross-reference, and storage of PL/SQL source text identifier data. Used by the PL/Scope tool
- **PLSQL_CCFLAGS**: lets you control conditional compilation of each unit independently
- **PLSQL_CODE_TYPE**: specifies the compilation mode for PL/SQL units
- **PLSQL_OPTIMIZE_LEVEL**: specifies the optimization level at which to compile PL/SQL units
- **PLSQL_WARNINGS**: Enables or disables the reporting of warning messages by the PL/SQL compiler, and specifies which warning messages to show as errors.
- **NLS_LENGTH_SEMANTICS**: Lets you create `CHAR` and `VARCHAR2` columns using either byte-length or character-length semantics.
- **PERMIT_92_WRAP_FORMAT**: Specifies whether the 12.1 PL/SQL compiler can use wrapped packages that were compiled with the 9.2 PL/SQL compiler. The default value is `TRUE`.

The compile time values of these parameters are stored with the metadata of each stored PL/SQL unit. We can reuse these values when recompiling by using the `REUSE SETTINGS` clause of the `ALTER` statement, e.g.

```
ALTER PROCEDURE procedure_name COMPILE REUSE SETTINGS;
```

PLSCOPE_SETTINGS

- modifiable: `ALTER SESSION, ALTER SYSTEM`
- syntax:

  ```
  PLSCOPE_SETTINGS = 'value_clause [, value_clause ]'
  ```

  - value_clause:

    ```
    { IDENTIFIERS | STATEMENTS } : { ALL | NONE | PUBLIC (for IDENTIFIERS
    only)
                                     | SQL (for IDENTIFIERS only)
                                     | PLSQL (for IDENTIFIERS only) }
    ```

  - values:
    - `IDENTIFIERS:ALL` Enables the collection of all source code identifier data
    - `IDENTIFIERS:NONE` Disables the collection of all source code identifier data
    - `IDENTIFIERS: PUBLIC` Enables the collection of all `PUBLIC` user identifier data (except for `DEFINITION`)
    - `IDENTIFIERS:SQL` Enables the collection of all SQL identifier data
    - `IDENTIFIERS:PLSQL` Enables the collection of all PLSQL identifier data
    - `STATEMENTS:ALL` Enables the collection of all SQL statements used in PL/SQL
    - `STATEMENTS:NONE` Disables the collection of all statements

- default: `'IDENTIFIERS:NONE'`
- `PLSCOPE_SETTINGS` can be set on a session, system, or per-library unit (`ALTER COMPILE`) basis
- The current setting of `PLSCOPE_SETTINGS` for any library unit can be attained by querying the `*_PLSQL_OBJECT_SETTINGS` views
- Any identifier data collected by setting this parameter can be accessed using the `*_IDENTIFIERS` views
- When a `STATEMENTS` setting is not specified, and `IDENTIFIERS` is specified but set to a value other than `NONE`, `STATEMENTS` defaults to a setting of `ALL`, which is equal to:

```
IDENTIFIERS: [ALL|PLSQL|PLSQL|PUBLIC]
```

PLSQL_CCFLAGS

- modifiable by `ALTER SESSION` or `ALTER SYSTEM`
- syntax:

```
PLSQL_CCFLAGS = '[V1:C1] [, V2:C2] [, Vn:Cn]'
```

  - V (a.k.a. flag or flag name):
    - is an unquoted PL/SQL identifier
    - can be a reserved word or keyword and is unrestricted
    - insensitive to case
    - can occur more than once and can have different flag values which can also be of different kind
  - C (a.k.a. flag value):
    - can be `BOOLEAN`, `PLS_INTEGER` or literal `NULL`
    - insensitive to case
    - corresponds to flag name
- default: empty string ''
- You can define any allowable value for `PLSQL_CCFLAGS`. However, Oracle recommends that this parameter be used for controlling the conditional compilation of debugging or tracing code
- It is recommended that the following identifiers not be used as flag name values:
  - names of Oracle parameters(e.g. `NLS_LENGTH_SEMANTICS`)
  - identifiers with any of the following prefixes: `PLS_, PLSQL_, PLSCC_, ORA_, ORACLE_, DBMS_, SYS_`
- Examples:

```
ALTER SESSION SET PLSQL_CCFLAGS = 'DeBug:TruE';
```

PLSQL_CODE_TYPE

- modifiable by `ALTER SESSION` or `ALTER SYSTEM`
- syntax:

```
PLSQL_CODE_TYPE = { INTERPRETED | NATIVE }
```

  - `INTERPRETED`: library units will be compiled to PL/SQL bytecode format. Such modules are executed by the PL/SQL interpreter engine. Stored in the calatog, interpreted at run time
  - `NATIVE`: library units (with possible exception of top level anonymous PL/SQL blocks) will be compiled to native (machine) code. Such modules will be executed natively without incurring any interpreter overhead. Stored in the catalog, need not be interpreted at run time, so it runs faster

- default: `INTERPRETED`
- When the value of this parameter is changed, it has no effect on PL/SQL library units that have already been compiled. The value of this parameter is stored persistently with each library unit
- If a PL/SQL library unit is compiled native, all subsequent automatic recompilations of that library unit will use native compilation

Compiling PL/SQL Units for Native Execution

- You can usually speed up PL/SQL units by compiling them into native code, which is stored in the `SYSTEM` tablespace.
- You can natively compile any PL/SQL unit of any type, including those that Oracle Database supplies
- Natively compiled program units work in all server environments, including shared server configuration (formerly called "multithreaded server") and Oracle Real Application Clusters (Oracle RAC)
- On most platforms, PL/SQL native compilation requires no special set-up or maintenance. On some platforms, the DBA might want to do some optional configuration
- You can test to see how much performance gain you can get by enabling PL/SQL native compilation
- If you have determined that PL/SQL native compilation will provide significant performance gains in database operations, Oracle recommends compiling the entire database for native mode, which requires DBA privileges. This speeds up both your own code and calls to the PL/SQL packages that Oracle Database supplies

Determining Whether to Use PL/SQL Native Compilation

- Whether to compile a PL/SQL unit for native or interpreted mode depends on where you are in the development cycle and on what the program unit does
- During debugging phase and frequent recompilation, `INTERPRETED` mode has advantages:
  - can use PL/SQL debugging tools on program units compiled for interpreted mode (only)
  - compiling for `INTERPRETED` mode is faster than compiling for `NATIVE` mode

- After debugging phase you may want to switch to `NATIVE` mode but consider:
  - PL/SQL native compilation provides the greatest performance gains for computation-intensive procedural operations
  - PL/SQL native compilation provides least performance gains for PL/SQL subprograms that spend most of their time running SQL
  - when many program units (typically over 15000) are compiled for native execution, and are simultaneously active, the large amount of shared memory required might affect system performance

How PL/SQL Native Compilation Works

- Without native compilation, the PL/SQL statements in a PL/SQL unit are compiled into an intermediate form, system code, which is stored in the catalog and interpreted at run time
- With PL/SQL native compilation, the PL/SQL statements in a PL/SQL unit are compiled into native code and stored in the catalog. The native code need not be interpreted at run time, so it runs faster
- Because native compilation applies only to PL/SQL statements, a PL/SQL unit that uses only SQL statements might not run faster when natively compiled, but it does run at least as fast as the corresponding interpreted code. The compiled code and the interpreted code make the same library calls, so their action is the same
- The first time a natively compiled PL/SQL unit runs, it is fetched from the `SYSTEM` tablespace into shared memory. Regardless of how many sessions invoke the program unit, shared memory has only one copy it. If a program unit is not being used, the shared memory it is using might be freed, to reduce memory load
- Natively compiled subprograms and interpreted subprograms can invoke each other.
- The `PLSQL_CODE_TYPE` compilation parameter determines whether PL/SQL code is natively compiled or interpreted

Dependencies, Invalidation, and Revalidation

- Recompilation is automatic with invalidated PL/SQL modules. For example, if an object on which a natively compiled PL/SQL subprogram depends changes, the subprogram is invalidated. The next time the same subprogram is called, the database recompiles the subprogram automatically
- Because the `PLSQL_CODE_TYPE` setting is stored inside the library unit for each subprogram, the automatic recompilation uses this stored setting for code type
- Explicit recompilation does not necessarily use the stored `PLSQL_CODE_TYPE` setting

Setting Up a New Database for PL/SQL Native Compilation

- If you have DBA privileges, you can set up a new database for PL/SQL native compilation by setting the compilation parameter `PLSQL_CODE_TYPE` to `NATIVE`
- The performance benefits apply to the PL/SQL packages that Oracle Database supplies, which are used for many database operations
- > Note: If you compile the whole database as `NATIVE`, Oracle recommends that you set `PLSQL_CODE_TYPE` at the system level

Compiling the Entire Database for PL/SQL Native or Interpreted Compilation

- If you have DBA privileges, you can recompile all PL/SQL modules in an existing database to `NATIVE` or `INTERPRETED`, using the `dbmsupgnv.sql` and `dbmsupgin.sql` scripts respectively

PLSQL_OPTIMIZE_LEVEL

- modifiable by `ALTER SESSION` or `ALTER SYSTEM`
- syntax:

```
PLSQL_OPTIMIZE_LEVEL = { 0 | 1 | 2 | 3 }
```

  - 0: maintains evaluation order and behaviour of Oracle releases prior to 10g. Also removes the new semantic identity of `BINARY_INTEGER` and `PLS_INTEGER` and restores the earlier rules for the evaluation of integer expressions. Use of level 0 will forfeit most of the performance gains of PL/SQL in Oracle 10g.
  - 1: applies wide range of optimizations such as elimination of unnecessary computations and exceptions, but generally does not move source code out of its original source order.
  - 2: more modern optimization techniques beyond those of level 1 and may move source code relatively far from its original location.
  - 3: wide range of optimization techniques beyond those of level 2, including techniques not specifically requested.
- default: level 2
- Generally, setting this parameter to 2 pays off in better execution performance. If, however, the compiler runs slowly on a particular source module or if optimization does not make sense for some reason, then setting this parameter to 1 will result in almost as good a compilation with less use of compile-time resources.
- The value of this parameter is stored persistently with the library unit

EXAMPLES

Using `ALTER SESSION` or `ALTER SYSTEM` to change PL/SQL compilation parameters

```
alter session set plsql_ccflags = 'flag:TRUE, flag:5';
```

Using `ALTER program_unit` to recompile it with reused settings

```
alter procedure procedure_name compile reuse settings;
```

Using `ALTER program_unit` to recompile it with different compilation parameters

```
alter procedure procedure_name
compile plsql_optimize_level = 3
    plsql_ccflags = 'hello:true, world:5'
    plsql_code_type = NATIVE;
```

✅Use the PL/SQL compile time warnings

Compile-time warnings: 11.1

DBMS_WARNING

PLSQL_WARNINGS

Compile-Time Warnings

- While compiling stored PL/SQL units, the PL/SQL compiler generates warnings for conditions that are not serious enough to cause errors and prevent compilation—for example, using a deprecated PL/SQL feature
- To see warnings (and errors) generated during compilation, either query the static data dictionary view `*_ERRORS` or, in the SQL*Plus environment, use the command `SHOW ERRORS`
- The message code of a PL/SQL warning has the form `PLW-nnnnn`
- Compile-Time Warning Categories
    - `SEVERE`
        - Description: Condition might cause unexpected action or wrong results
        - Example: Aliasing problems with parameters
    - `PERFORMANCE`
        - Description: Condition might cause performance problems
        - Example: Passing a `VARCHAR2` value to a `NUMBER` column in an `INSERT` statement
    - `INFORMATIONAL`
        - Description: Condition does not affect performance or correctness, but you might want to change it to make the code more maintainable
        - Example: Code that can never run (unreachable code)
- By setting the compilation parameter `PLSQL_WARNINGS`, you can:
    - Enable and disable all warnings, one or more categories of warnings, or specific warnings
    - Treat specific warnings as errors (so that those conditions must be corrected before you can compile the PL/SQL unit)
- You can set the value of `PLSQL_WARNINGS` for:
    - Your Oracle database instance (Use the `ALTER SYSTEM` statement)
    - Your session (Use the `ALTER SESSION` statement)
    - A stored PL/SQL unit (Use an `ALTER` statement with its compiler_parameters_clause)
- In any of the preceding `ALTER` statements, you set the value of `PLSQL_WARNINGS` with this syntax:

```
PLSQL_WARNINGS = 'value_clause' [, 'value_clause' ] ...
```

- To display the current value of `PLSQL_WARNINGS`, query the static data dictionary view `ALL_PLSQL_OBJECT_SETTINGS`
- Examples
    - enable all warnings

```
ALTER SESSION SET PLSQL_WARNINGS='ENABLE:ALL';
```

    - enable severe, disable performance, treat plw-06002 as error

```
ALTER SESSION SET PLSQL_WARNINGS='ENABLE:SEVERE',
'DISABLE:PERFORMANCE', 'ERROR:06002';
```

DBMS_WARNING package

- If you are writing PL/SQL units in a development environment that compiles them (such as SQL*Plus), you can display and set the value of `PLSQL_WARNINGS` by invoking subprograms in the `DBMS_WARNING` package
- `DBMS_WARNING` subprograms are useful when you are compiling a complex application composed of several nested SQL*Plus scripts, where different subprograms need different `PLSQL_WARNINGS` settings
- With `DBMS_WARNING` subprograms, you can save the current `PLSQL_WARNINGS` setting, change the setting to compile a particular set of subprograms, and then restore the setting to its original value
- The DBMS_WARNING package provides a way to manipulate the behavior of PL/SQL warning messages, in particular by reading and changing the setting of the PLSQL_WARNINGS initialization parameter to control what kinds of warnings are suppressed, displayed, or treated as errors. This package provides the interface to query, modify and delete current system or session settings
- Subprograms
    - `ADD_WARNING_SETTING_CAT`
        - Modifies the current session or system warning settings of the warning_category previously supplied
    - `ADD_WARNING_SETTING_NUM`
        - Modifies the current session or system warning settings of the or warning_number previously supplied
    - `GET_CATEGORY`
        - Returns the category name, given the message number
    - `GET_WARNING_SETTING_CAT`
        - Returns the specific warning category in the session
    - `GET_WARNING_SETTING_NUM`
        - Returns the specific warning number in the session
    - `GET_WARNING_SETTING_STRING`
        - Returns the entire warning string for the current session
    - `SET_WARNING_SETTING_STRING`
        - Replaces previous settings with the new value

PLSQL_WARNINGS

- `PLSQL_WARNINGS` enables or disables the reporting of warning messages by the PL/SQL compiler, and specifies which warning messages to show as errors
- modifiable: `ALTER SESSION, ALTER SYSTEM`
- default: `'DISABLE:ALL'`
- syntax:

```
PLSQL_WARNINGS = 'value_clause' [, 'value_clause' ]
```

  - value_clause

```
{ ENABLE | DISABLE | ERROR } { ALL | SEVERE | INFORMATIONAL |
PERFORMANCE
| {integer | (integer [, ...])} }
```

    Multiple value clauses may be specified, enclosed in quotes and separated by commas. Each value clause is composed of a qualifier, a colon [:], and a modifier