1

Changes in This Release for Oracle Database PL/SQL Language Reference

New Features in Release 23ai for Oracle Database PL/SQL Language Reference

For Oracle Database 23ai, PL/SQL Language Reference documents these new features and enhancements.

See Also:

Oracle Database New Features for the descriptions of all of the features that are new in Oracle Database Release 23ai

SQL BOOLEAN Data Type

Although `BOOLEAN` support has already been available with PL/SQL prior to this release, the `BOOLEAN` data type is now supported by SQL as well. This expansion of support provides improved compatibility between PL/SQL and SQL.

PL/SQL stored functions with `BOOLEAN` parameter types are now invokable directly from SQL. While PL/SQL functions with `BOOLEAN` arguments were already callable from SQL (with arguments of `BOOLEAN` type binds), `BOOLEAN` expressions and `BOOLEAN` literals are now supported as well. Additionally, PL/SQL stored procedures and anonymous PL/SQL blocks with host binds that expect values of the `BOOLEAN` type are invokable from C via OCI and other interfaces such as dynamic SQL and `DBMS_SQL`. It is also possible to call a C trusted or safe callout with formal parameters of `BOOLEAN` type.

`BOOLEAN` defines in the `INTO` and `BULK COLLECT INTO` clauses of a `SELECT` statement inside a PL/SQL block are supported.

Implicit conversions between `BOOLEAN` and number and character types are supported. It is possible to assign number and character variables and expressions to a `BOOLEAN` variable. The function `to_boolean` has also been added to convert from number and character types to the `BOOLEAN` data type. The functions `to_number`, `to_char`, and `to_nchar` now have `BOOLEAN` overloads to convert `BOOLEAN` values to number or character types. You can use the `CAST` operator to cast an expression to the `BOOLEAN` type as well.

To enable the support of implicit conversion, the initialization parameter `PLSQL_IMPLICIT_CONVERSION_BOOL` must be set to `TRUE`. Explicit conversions such as `CAST` and `to_char` do not depend on the parameter, so will work regardless of whether `PLSQL_IMPLICIT_CONVERSION_BOOL` is set to `TRUE` or `FALSE`. For more information about using `PLSQL_IMPLICIT_CONVERSION_BOOL`, see *Oracle Database Reference*.

All DML operations from PL/SQL take `BOOLEAN` variables in both `IN` and `OUT` binds, including array `IN` binds and array `OUT` binds. DML triggers support `BOOLEAN` binds (both `IN` and `OUT` binds) and `BOOLEAN` column references in the `WHEN` clause.

Pipelined table functions and polymorphic table functions (PTF) support returning columns of `BOOLEAN` data type.

It is possible to invoke Java and JavaScript stored procedures with parameters of `BOOLEAN` type from PL/SQL using call specifications with parameters of `BOOLEAN` type. PL/SQL procedures and anonymous blocks are also invokable from Java using JDBC in the server or client-server environment. Variables of `BOOLEAN` type passed using OCI, dynamic SQL, or `DBMS_SQL` can be passed directly as `BOOLEAN`.

Note:

`BOOL` can be used as an abbreviation of `BOOLEAN`.

See Also:

- *Oracle Database SQL Language Reference* for more information about the SQL `BOOLEAN` data type
- *Oracle Database SQL Language Reference* for more information about data conversion rules

Example 1-1 Calling a PL/SQL Function with BOOLEAN Argument from SQL

```
CREATE OR REPLACE FUNCTION useBool(p1 BOOLEAN) RETURN NUMBER AS
BEGIN
    IF p1 THEN RETURN 100;
    ELSE
        RETURN 200;
    END IF;
END;
/

SET SERVEROUTPUT ON;
DECLARE
    v1 NUMBER;
    v2 BOOLEAN := TRUE;
BEGIN
    SELECT useBool(v2) INTO v1 FROM dual; --boolean argument function called
from SELECT
    DBMS_OUTPUT.PUT_LINE(v1);
END;
/
```

Result:

```
100
```

SQL VECTOR Data Type

With Oracle Database Release 23ai, PL/SQL supports the `VECTOR` data type along with vector operations.

The `VECTOR` data type is introduced to support Artificial Intelligence workloads with Oracle AI Vector Search. Along with the new data type, vector operations including distance functions are provided to manage vectors in the database. In PL/SQL, the `VECTOR` data type is supported as its own distinct scalar family type and can be used in the same way as any other data type in PL/SQL.

For more information about vectors in PL/SQL, see VECTOR Data Type.

For information about Oracle AI Vector Search and general support for vectors in Oracle Database, see *Oracle Database AI Vector Search User's Guide*.

IF [NOT] EXISTS Syntax Support

The clauses `IF NOT EXISTS` and `IF EXISTS` are supported by `CREATE`, `ALTER`, and `DROP` DDL statements. They are used to suppress potential errors otherwise raised by the existence or non-existence of a given object, allowing you to write idempotent DDL scripts.

The `IF NOT EXISTS` clause is supported by the `CREATE` DDL statement to prevent errors from being thrown if an object with the given name already exists. If the object does already exist, the command is ignored and the original object remains unchanged.

On the flip side, the `IF EXISTS` clause suppresses errors when used with `ALTER` and `DROP` DDL statements. In the case that no object by the given name exists, the command is ignored and no object is affected by `ALTER` or `DROP`.

The use or exclusion of the clause provides you more control depending on whether you need to know if an object exists before executing a DDL statement. With this flexibility, you can determine whether you would rather have the statement ignored or have an error raised in the event the object exists (or doesn't exist).

Note:

`IF NOT EXISTS` cannot be used in combination with `OR REPLACE` in commands using the `CREATE` DDL statement.

See Also:

- SQL Statements for Stored PL/SQL Units for information about the semantics used to implement `IF [NOT] EXISTS` with different object types
- *Oracle Database Development Guide* for more information about using the `IF [NOT] EXISTS` clause

Example 1-2   CREATE PROCEDURE with IF NOT EXISTS

Executing this statement one time results in the creation of procedure `hello`, assuming a procedure by the same name does not already exist in your schema.

```
CREATE PROCEDURE IF NOT EXISTS hello AS
BEGIN
  DBMS_OUTPUT.PUT_LINE('Hello there');
END;
/
```

Executing the statement additional times, even with an altered procedure body, results in no error. The original body remains unchanged.

```
CREATE PROCEDURE IF NOT EXISTS hello AS
BEGIN
  DBMS_OUTPUT.PUT_LINE('Second hello');
END;
/
```

Note:

The same output message will be displayed, in this case `Procedure created`, regardless of whether the command is ignored or executed. This ensures that you can write DDL scripts that are idempotent. The same holds true for `ALTER`, `CREATE`, and `DROP` statements.

The procedure text is the same before and after the second statement is executed.

```
SELECT TEXT FROM USER_SOURCE WHERE NAME='HELLO';

TEXT
-------------------------------------------
procedure            hello
AS BEGIN
DBMS_OUTPUT.PUT_LINE('Hello there');
END;
```

Extended CASE Controls

The simple `CASE` statement is extended in PL/SQL to support the use of dangling predicates and choice lists, allowing for simplified and less redundant code.

Dangling predicates are ordinary expressions with their left operands missing that can be used as a `selector_value` either instead of or in combination with any number of literals or expressions. With dangling predicates, more complicated comparisons can be made without requiring a searched `CASE` statement.

Note:

Currently, the dangling predicates `IS JSON` and `IS OF` are not supported.

Comma separated lists of choices are now supported in the `WHEN` clause(s) of a simple `CASE` statement. They can help streamline code by allowing for the consolidation of multiple `selector_value` options that correspond to the same result.

See Also:

- Simple CASE Expression for more information about using the extended case controls and for an example that uses a choice list
- Simple CASE Statement for more information about using the extended `CASE` controls and for an example that uses dangling predicates
- CASE Statement for information on the syntax and semantics of simple `CASE` statements, including the extended `CASE` controls

JSON Constructor and JSON_VALUE Support of PL/SQL Aggregate Types

The JSON constructor can now accept a PL/SQL aggregate type and return a JSON object or array populated with the aggregate type data. Conversely, the built-in function `json_value` now supports PL/SQL aggregate types in the `RETURNING` clause, mapping from JSON to the specified aggregate type.

All PL/SQL record field and collection data element type constraints are honored by `json_value`, including character max length, integer range checks, and not null constraints.

SQL objects and PL/SQL record type instances, including implicit records created by the `%ROWTYPE` attribute, are allowed as valid input to the JSON constructor. Expanded support for user defined types as input streamlines data interchange between PL/SQL applications and languages that support JSON.

See Also:

- PL/SQL and JSON Type Conversions for more information about using `json_value` and the JSON constructor with PL/SQL aggregate types
- *Oracle Database JSON Developer's Guide* for details about the JSON constructor
- *Oracle Database JSON Developer's Guide* for information about using `json_value` to instantiate a user-defined object-type or collection-type instance

SQL Transpiler

The SQL Transpiler automatically and wherever possible converts (transpiles) PL/SQL functions within SQL into SQL expressions, without user intervention.

The conversion operation is transparent to users and can improve performance by reducing overhead accrued from switching between the SQL and PL/SQL runtime.

See Also:

- SQL_MACRO Clause for information about how standard PL/SQL functions can be used as an alternative to `SCALAR` macros due to the SQL Transpiler
- *Oracle Database SQL Tuning Guide* for details about the SQL Transpiler

Oracle Database 23ai, Release Update 23.7

For Oracle Database 23ai, Release Update 23.7, PL/SQL Language Reference documents these new features and enhancements.

Jaccard Distance Metric

With Oracle Database Release 23ai, Release Update 23.7, PL/SQL supports the Jaccard distance metric, used with vector similarity search.

The Jaccard similarity is used to determine the share of significant (non-zero) attributes common between two asymmetric `BINARY` vectors. The distance metric is applicable only to `BINARY` vectors.

For more information about using vectors in PL/SQL, see VECTOR Data Type.

For more information about the Jaccard similarity, see *Oracle Database AI Vector Search User's Guide*.

BINARY Vector Support

With Oracle Database Release 23ai, Release Update 23.7, PL/SQL supports the `BINARY` vector format.

The `BINARY` format is now supported by the `VECTOR` data type in PL/SQL.

For more information about vectors in PL/SQL, see VECTOR Data Type.

For information about `BINARY` vectors, see *Oracle Database AI Vector Search User's Guide*.

VECTOR Dimension-wise Arithmetic Support

With Oracle Database Release 23ai, Release Update 23.7, PL/SQL supports the `VECTOR` data type in arithmetic operations.

Dimension-wise addition, subtraction, and multiplication can be performed on variables and columns of the `VECTOR` data type.

For more information about vector arithmetic in PL/SQL, see VECTOR Operations Supported by PL/SQL.

For information about `VECTOR` arithmetic and aggregation, see *Oracle Database AI Vector Search User's Guide*.

Oracle Database 23ai, Release Update 23.8

For Oracle Database 23ai, Release Update 23.8, PL/SQL Language Reference documents these new features and enhancements.

SPARSE Vector Support

With Oracle Database Release 23ai, Release Update 23.8, PL/SQL natively supports the creation and declaration of `SPARSE` vectors.

Sparse vectors are vectors that typically have a large number of dimensions but only a few of those dimensions have non-zero values. In certain cases, using `SPARSE` vectors can be more space-efficient and can improve similarity search performance when compared with using `DENSE` vectors.

If you are using RAC instances, you must enable Patch 37535524 using Oracle RAC two-stage rolling updates to enable support for `SPARSE` in PL/SQL. For more information, see *Oracle Real Application Clusters Administration and Deployment Guide*.

For more information about `SPARSE` vectors in PL/SQL, see VECTOR Data Type.

For information about `SPARSE` vectors, see *Oracle Database AI Vector Search User's Guide*.

Deprecated Features

The following features are deprecated, and may be desupported in a future release.

The command `ALTER TYPE` ... `INVALIDATE` is deprecated. Use the `CASCADE` clause instead.

The `REPLACE` clause of `ALTER TYPE` is deprecated. Use the *`alter_method_spec`* clause instead. Alternatively, you can recreate the type using the `CREATE OR REPLACE TYPE` statement.

For the syntax and semantics, see ALTER TYPE Statement
Starting with Oracle Database 12c release 1 (12.1), the compilation parameter PLSQL_DEBUG is deprecated.

To compile PL/SQL units for debugging, specify PLSQL_OPTIMIZE_LEVEL=1.

For information about compilation parameters, see PL/SQL Units and Compilation Parameters.

Desupported Features

No features in PL/SQL Language Reference have been desupported.

See Also:

- *Oracle Database Upgrade Guide* for more information about desupported features in this release of Oracle Database