4
PL/SQL Data Types

Every PL/SQL constant, variable, parameter, and function return value has a **data type** that determines its storage format and its valid values and operations.

This chapter explains **scalar data types**, which store values with no internal components.

A scalar data type can have subtypes. A **subtype** is a data type that is a subset of another data type, which is its **base type**. A subtype has the same valid operations as its base type. A data type and its subtypes comprise a **data type family**.

PL/SQL predefines many types and subtypes in the package `STANDARD` and lets you define your own subtypes.

The PL/SQL scalar data types are:

- The SQL data types
- `PLS_INTEGER`
- `BINARY_INTEGER`
- `REF CURSOR`
- User-defined subtypes

Topics

- SQL Data Types
- PLS_INTEGER and BINARY_INTEGER Data Types
- SIMPLE_INTEGER Subtype of PLS_INTEGER
- User-Defined PL/SQL Subtypes

> **See Also:**
>
> - "PL/SQL Collections and Records" for information about **composite data types**
> - "Cursor Variables" for information about `REF CURSOR`
> - "CREATE TYPE Statement" for information about creating schema-level user-defined data types
> - "PL/SQL Predefined Data Types" for the predefined PL/SQL data types and subtypes, grouped by data type family

SQL Data Types

The PL/SQL data types include the SQL data types.

For information about the SQL data types, see *Oracle Database SQL Language Reference*—all information there about data types and subtypes, data type comparison rules, data conversion, literals, and format models applies to both SQL and PL/SQL, except as noted here:

- Different Maximum Sizes
- Additional PL/SQL Constants for BINARY_FLOAT and BINARY_DOUBLE
- Additional PL/SQL Subtypes of BINARY_FLOAT and BINARY_DOUBLE

Unlike SQL, PL/SQL lets you declare variables, to which the following topics apply:

- BOOLEAN Data Type
- JSON Data Type
- VECTOR Data Type
- CHAR and VARCHAR2 Variables
- LONG and LONG RAW Variables
- ROWID and UROWID Variables

Different Maximum Sizes

The SQL data types listed in Table 4-1 have different maximum sizes in PL/SQL and SQL.

Table 4-1 Data Types with Different Maximum Sizes in PL/SQL and SQL

Data Type	Maximum Size in PL/SQL	Maximum Size in SQL
`CHAR`[1]	32,767 bytes	2,000 bytes
`NCHAR`[1]	32,767 bytes	2,000 bytes
`RAW`[1]	32,767 bytes	2,000 bytes[2]
`VARCHAR2`[1]	32,767 bytes	4,000 bytes[2]
`NVARCHAR2`[1]	32,767 bytes	4,000 bytes[2]
`LONG`[3]	32,760 bytes	2 gigabytes (GB) - 1
`LONG RAW`[3]	32,760 bytes	2 GB
`BLOB`	128 terabytes (TB)	(4 GB - 1) * *`database_block_size`*
`CLOB`	128 TB	(4 GB - 1) * *`database_block_size`*
`NCLOB`	128 TB	(4 GB - 1) * *`database_block_size`*

1 When specifying the maximum size of a value of this data type in PL/SQL, use an integer literal (not a constant or variable) whose value is in the range from 1 through 32,767.
2 To eliminate this size difference, follow the instructions in *Oracle Database SQL Language Reference*.
3 Supported only for backward compatibility with existing applications.

Additional PL/SQL Constants for BINARY_FLOAT and BINARY_DOUBLE

The SQL data types `BINARY_FLOAT` and `BINARY_DOUBLE` represent single-precision and double-precision IEEE 754-format floating-point numbers, respectively.

`BINARY_FLOAT` and `BINARY_DOUBLE` computations do not raise exceptions, so you must check the values that they produce for conditions such as overflow and underflow by comparing them to predefined constants (for examples, see *Oracle Database SQL Language Reference*). PL/SQL has more of these constants than SQL does.

Table 4-2 lists and describes the predefined PL/SQL constants for BINARY_FLOAT and BINARY_DOUBLE, and identifies those that SQL also defines.

Table 4-2 Predefined PL/SQL BINARY_FLOAT and BINARY_DOUBLE Constants

Constant	Description
BINARY_FLOAT_NAN (*)	BINARY_FLOAT value for which the condition IS NAN (not a number) is true
BINARY_FLOAT_INFINITY (*)	Single-precision positive infinity
BINARY_FLOAT_MAX_NORMAL	Maximum normal BINARY_FLOAT value
BINARY_FLOAT_MIN_NORMAL	Minimum normal BINARY_FLOAT value
BINARY_FLOAT_MAX_SUBNORMAL	Maximum subnormal BINARY_FLOAT value
BINARY_FLOAT_MIN_SUBNORMAL	Minimum subnormal BINARY_FLOAT value
BINARY_DOUBLE_NAN (*)	BINARY_DOUBLE value for which the condition IS NAN (not a number) is true
BINARY_DOUBLE_INFINITY (*)	Double-precision positive infinity
BINARY_DOUBLE_MAX_NORMAL	Maximum normal BINARY_DOUBLE value
BINARY_DOUBLE_MIN_NORMAL	Minimum normal BINARY_DOUBLE value
BINARY_DOUBLE_MAX_SUBNORMAL	Maximum subnormal BINARY_DOUBLE value
BINARY_DOUBLE_MIN_SUBNORMAL	Minimum subnormal BINARY_DOUBLE value

(*) SQL also predefines this constant.

Additional PL/SQL Subtypes of BINARY_FLOAT and BINARY_DOUBLE

PL/SQL predefines these subtypes:

- SIMPLE_FLOAT, a subtype of SQL data type BINARY_FLOAT
- SIMPLE_DOUBLE, a subtype of SQL data type BINARY_DOUBLE

Each subtype has the same range as its base type and has a NOT NULL constraint (explained in "NOT NULL Constraint").

If you know that a variable will never have the value NULL, declare it as SIMPLE_FLOAT or SIMPLE_DOUBLE, rather than BINARY_FLOAT or BINARY_DOUBLE. Without the overhead of checking for nullness, the subtypes provide significantly better performance than their base types. The performance improvement is greater with PLSQL_CODE_TYPE='NATIVE' than with PLSQL_CODE_TYPE='INTERPRETED' (for more information, see "Use Data Types that Use Hardware Arithmetic").

BOOLEAN Data Type

The data type BOOLEAN stores **logical values**, which are the boolean values TRUE and FALSE and the value NULL. NULL represents an unknown value.

The syntax for declaring a BOOLEAN variable is:

```
variable_name BOOLEAN
```

By default, you cannot pass a BOOLEAN value to any NUMBER or VARCHAR2 parameters for any procedures or functions, such as the DBMS_OUTPUT.PUT or DBMS_OUTPUT.PUT_LINE subprograms.

In order to pass a BOOLEAN value to these procedures, set the initialization parameter PLSQL_IMPLICIT_CONVERSION_BOOL to TRUE. Setting the parameter to TRUE also allows implicit conversions in the assignment of variables, for example, if you want to assign a NUMBER or VARCHAR2 value to a BOOLEAN variable. Additionally, a TRUE value makes it possible to use string literals in the assignment of BOOLEAN variables. The parameter has no effect on explicit conversions such as CAST or the functions TO_NUMBER, TO_CHAR, or TO_BOOLEAN.

If a subprogram is overloaded with BOOLEAN and numeric or character types, setting PLSQL_IMPLICIT_CONVERSION_BOOL to TRUE can cause compile-time errors. For more information about potential overload errors with the use of this parameter, see "Subprogram Overload Errors".

The PLSQL_IMPLICIT_CONVERSION_BOOL parameter is persistable, meaning any PL/SQL unit created with the parameter set uses the value specified at the time of unit creation when the unit is compiled with the REUSE SETTINGS clause.

It is also possible to assign a BOOLEAN expression to a BOOLEAN variable (regardless of the PLSQL_IMPLICIT_CONVERSION_BOOL parameter's value). For details about BOOLEAN expressions, see "BOOLEAN Expressions".

See Also:

- *Oracle Database Reference* for more information about the PLSQL_IMPLICIT_CONVERSION_BOOL parameter
- *Oracle Database SQL Language Reference* for information about the SQL BOOLEAN data type and for a list of available string literals used to represent TRUE and FALSE

Example 4-1 Printing BOOLEAN Values

In this example, BOOLEAN values are printed by passing the values directly to the procedure DBMS_OUTPUT.PUT_LINE. Executing this code successfully depends on the initialization parameter PLSQL_IMPLICIT_CONVERSION_BOOL being set to TRUE.

```
DECLARE
  t_b boolean := TRUE;
  f_b boolean := FALSE;
BEGIN
  DBMS_OUTPUT.PUT_LINE('My bool is: ' || t_b);
  DBMS_OUTPUT.PUT_LINE('My bool is: ' || f_b);
END;
```

Result:

```
My bool is: TRUE
My bool is: FALSE
```

JSON Data Type

You can use `JSON` data type instances with PL/SQL subprograms. The PL/SQL `JSON` data type is stored in the database in a binary form for faster access to nested `JSON` values.

You can use JSON data type and its instances in most places where a SQL data type is allowed, including:

- As the column type for table or view DDL.
- As a parameter type for a PL/SQL subprogram.
- As an element or field type in records, PL/SQL collections, and `%ROWTYPE` attributes.
- In expressions wherever a SQL/JSON function or condition is allowed.

The JSON data type is not currently supported in SQL collections or objects.

Topics

- PL/SQL and JSON Type Conversions

> **See Also:**
>
> - json-schema.org for information about JSON Schema
> - *Oracle Database JSON Developer's Guide* for details about using PL/SQL with JSON data
> - *Oracle Database JSON Developer's Guide* for more information about PL/SQL object types for JSON

PL/SQL and JSON Type Conversions

The built-in function `json_value` supports scalar data type mappings as well as mappings from JSON objects to user-defined PL/SQL types. Given an instance of a user-defined PL/SQL or SQL aggregate type, the PL/SQL JSON constructor returns a corresponding JSON object or JSON array type instance.

The use of PL/SQL user-defined subtypes as the returning aggregate data type is supported by `json_value`. This includes support for any constraints or initializers employed by subtypes used as field or element data types in a returning aggregate data type.

All PL/SQL record field and collection data element data type constraints are honored by PL/SQL `json_value`. Constraints include character max length, number scale and precision, time/time stamp/interval constraints, integer range checks, and not null constraints.

These types can be declared in any program scope visible to the `json_value` call site, including top-level SQL (for SQL objects and collections), package level PL/SQL, or locally in a PL/SQL function, procedure, or anonymous call block.

PL/SQL specific user-defined aggregate types include:

- Records
- `INDEX BY PLS_INTEGER` collections
- Associative arrays

- Nested tables
- Varrays
- Objects

PL/SQL aggregate types can be used as the `IN` and `RETURN` data types of PL/SQL built-in functions. All PL/SQL `%ROWTYPE`s are supported in the `RETURNING` clause of `json_value`.

The `ON MISMATCH` clause can be used with `json_value` to handle type matching exceptions. It is used to specify the desired behavior when a targeted JSON value cannot be converted to the specified return type. Note that PL/SQL records, index by `PLS_INTEGER` collections, and index by `VARCHAR2` collections cannot be atomically null. Therefore, the `NULL ON MISMATCH` clause raises a compile time error when one of these types is specified as the return type. For more information about the `ON MISMATCH` clause, see *Oracle Database JSON Developer's Guide*.

Type Name Resolution and Scoping

A type name used in `json_value` is resolved using standard PL/SQL name resolution rules. PL/SQL begins looking for a name in the inner-most scope of the PL/SQL code where the name is referenced and expands the search to the outer scopes until the name is resolved.

The PL/SQL built-in function `json_value` resolves up to three part names, which include the following formats:

- `<schema name>.<package name>.<type name>`
- `<package name>.<type name>`
- `<schema name>.<type name>`
- `<type name>`

Note that this differs from the SQL `json_value` built-in function, which only resolves one or two part type names.

Synonyms may be used where appropriate in the full type name string and those synonyms are resolved during type name resolution.

Topics

- JSON Objects and PL/SQL Records
- JSON Objects and Index by PLS_INTEGER and Nested Table Collections
- JSON Arrays and Nested Tables, Index by PLS_INTEGER, and Varray Collections
- JSON Objects and Associative Arrays

> **See Also:**
>
> - *Oracle Database JSON Developer's Guide* for more information about the `json_value` built-in function

JSON Objects and PL/SQL Records

PL/SQL records hold data using name/value pairs and can be mapped to and from JSON objects via the JSON constructor and the built-in function `json_value`, respectively.

Topics

- JSON Objects to PL/SQL Records
- PL/SQL Records to JSON Objects

JSON Objects to PL/SQL Records

When a PL/SQL record name is specified in the `RETURNING` clause, `json_value` maps the input JSON object to the PL/SQL record and returns an instance of the PL/SQL record. If the input JSON is not a JSON object, the `ON MISMATCH` clause applies.

To accomplish the mapping, each JSON key name must map to a unique attribute in the PL/SQL record using a default case-insensitive comparison that disregards any double quotes surrounding the name, as well as the placement of the key or attribute name in either of the types being mapped.

Case sensitive mapping is supported using the case-sensitive mapping syntax, as shown below:

```
DECLARE
    TYPE personrecord IS RECORD(first VARCHAR2(10), last VARCHAR2(10));
    p personrecord;
BEGIN
    p := JSON_VALUE(JSON('{"FIRST":"Jane", "LAST":"Cooper"}'), '$'
    RETURNING personrecord USING CASE_SENSITIVE MAPPING);
    DBMS_OUTPUT.PUT_LINE(p.first ||' '|| p.last);
END;
/
```

Once the key name is mapped, the JSON value for the key name is copied into the PL/SQL record attribute. The JSON value must be convertible to the PL/SQL data type of the mapped field. If the value types are not convertible, a `MISMATCH` error is raised.

Record types that contain JSON fields are supported in calls to `json_value`, with the JSON fields mapped to any JSON type, including JSON objects and JSON arrays. In other words, if a JSON attribute name is mapped to a record field name and the record field is a JSON type, PL/SQL copies the JSON value of the JSON attribute into the record field JSON type.

The JSON value must be valid JSON. If the JSON document is textual, the JSON value is parsed when it is copied into the JSON field to verify that it is valid JSON. Once the copy is complete, no further recursive mapping takes place for the attribute.

Example 4-2 Convert a JSON Object to PL/SQL Records

This example demonstrates how the same JSON object can be mapped to two different PL/SQL records.

```
DECLARE
    TYPE theRec1 IS RECORD (field1 NUMBER, field2 VARCHAR2(10));
    TYPE theRec2 IS RECORD ("fIeLd2" VARCHAR2(20), "FieLD1" NUMBER);
```

```
    Rec1 theRec1;
    Rec2 theRec2;
BEGIN
    Rec1 := JSON_VALUE(JSON('{"FIELD1":10, "field2":"hello"}'), '$' RETURNING
theRec1);
    Rec2 := JSON_VALUE(JSON('{"FIELD1":10, "field2":"hello"}'), '$' RETURNING
theRec2);
END;
/
```

Running the PL/SQL block results in `Rec1` and `Rec2` containing the following values, respectively:

```
theRec1(field1=>10, field2=>'hello')
theRec2("fIeLd2"=>'hello', "FielD1"=>10)
```

PL/SQL Records to JSON Objects

SQL objects and PL/SQL record type instances, including implicit records created by the `<table | view | cursor>%ROWTYPE` attribute, are allowed as valid inputs to the JSON constructor.

The PL/SQL object attribute name becomes the JSON key name. Double quoted attribute names become case sensitive JSON key names while non-double quoted attribute names become uppercase JSON key names. In PL/SQL object attribute values are mapped to the closest JSON value type.

Example 4-3 Convert a PL/SQL Record to a JSON Object

```
DECLARE
    TYPE theRec IS RECORD(field1 NUMBER, "Field2" NUMBER);
    myRec theRec := theRec(10, 20);
    myJson JSON;
BEGIN
    myJson := JSON(myRec);
    DBMS_OUTPUT.PUT_LINE(JSON_SERIALIZE(myJson));
END;
/
```

Result:

```
{"FIELD1":10, "Field2":20}
```

JSON Objects and Index by PLS_INTEGER and Nested Table Collections

Index by `PLS_INTEGER` collections and nested table collections can be converted to and from JSON objects using the built-in `json_value` function and the JSON constructor, respectively.

Topics

- JSON Objects to Index by PLS_INTEGER and Nested Table Collections
- Index by PLS_INTEGER Collections and Nested Types to JSON Objects

JSON Objects to Index by PLS_INTEGER and Nested Table Collections

Index by `PLS_INTEGER` and nested table collections can both be sparse collection types that depend on integer indexed elements. These types map to JSON objects, where the string key attribute of the object is a string representation of the collection's integer index.

When converting from a JSON object to either collection type, an error is raised if the JSON object string key attribute does not cleanly convert into an integer value. With nested table collections, the key attribute must be a positive integer, otherwise an error is raised. Additionally, the maximum key value cannot exceed the number of elements in the JSON object. If a larger key value is required, an index by `PLS_INTEGER` collection can be used.

If there are any gaps between index values in the object, those gaps are recreated in both collection types. That is, if elements are missing between the lowest and highest number index in the JSON object, those elements will also be missing in the collection. Keep in mind that missing elements are not the same as `NULL` elements.

The JSON object index key attributes do not need to be in sorted order. They are sorted when they are inserted into the collection.

Example 4-4   Convert a JSON Object to an Index by PLS_INTEGER Collection

This example demonstrates the conversion of a JSON object to an Index by `PLS_INTEGER` collection using the built-in function `json_value`.

```
DECLARE
    TYPE theIBPLS IS TABLE OF NUMBER INDEX BY BINARY_INTEGER;
    myIBPLS theIBPLS;
BEGIN
    myIBPLS := JSON_VALUE(JSON('{"-10":10, "-1":1, "100":-100}'), '$'
RETURNING theIBPLS);
END;
/
```

Running the PL/SQL block results in the creation of an Index by `PLS_INTEGER` collection with the following element values:

```
theIBPLS(-10=>10, -1=>1, 100=>-100)
```

Example 4-5   Convert a JSON Object to a Nested Table Collection

This example demonstrates the conversion of a JSON object to a nested table collection using the built-in function `json_value`.

```
DECLARE
    TYPE theNSTTAB IS TABLE OF NUMBER;
    myNSTTAB theNSTTAB;
BEGIN
    myNSTTAB := JSON_VALUE(JSON('{"1":10, "2":20, "3":30, "4":40}'), '$'
RETURNING theNSTTAB);
END;
/
```

Running the PL/SQL block results in the creation of a nested table collection with the following values:

```
theNSTTAB(1=>10, 2=>20, 3=>30, 4=>40)
```

Index by PLS_INTEGER Collections and Nested Types to JSON Objects

Index by `PLS_INTEGER` collections are converted to a JSON object with index values preserved when passed to a JSON constructor. When represented as a JSON object, the collection's index appears as a JSON string representation of the index integer value.

In order to preserve sparseness on a round trip from PL/SQL to JSON and back to PL/SQL, a nested table collection is converted to a JSON object when it is passed to a JSON constructor. When represented as a JSON object, nested table indices appear as a JSON string representation of the index integer value.

Example 4-6 Convert an Index by PLS_INTEGER Collection to a JSON Object

This example demonstrates the conversion of an index by `PLS_INTEGER` collection to a JSON object using the JSON constructor.

```
DECLARE
    TYPE theIBPLS IS TABLE OF NUMBER INDEX BY PLS_INTEGER;
    myIBPLS theIBPLS := theIBPLS(-1=>1, 2=>2, -3=>3);
    myJSON JSON;
BEGIN
    myJSON := JSON(myIBPLS);
    DBMS_OUTPUT.PUT_LINE(JSON_SERIALIZE(myJSON));
END;
/
```

Result:

```
{ "-3":3, "-1":1, "2":2 }
```

Example 4-7 Convert a Nested Table to a JSON Object

This example demonstrates the conversion of a sparse nested table into a JSON object using the JSON constructor.

```
DECLARE
    TYPE theNSTTAB IS TABLE OF NUMBER;
    myNSTTAB theNSTTAB := theNSTTAB(1=>1, 2=>2, 3=>3);
    myJSON JSON;
BEGIN
    myNSTTAB.delete(2); --myNSTTAB becomes sparse when elements are deleted
    myJSON := JSON(myNSTTAB);
    DBMS_OUTPUT.PUT_LINE(JSON_SERIALIZE(myJSON));
END;
/
```

Result:

```
{ "1":1, "3":3 }
```

JSON Arrays and Nested Tables, Index by PLS_INTEGER, and Varray Collections

JSON arrays are converted to nested tables, Index by `PLS_INTEGER`, or Varray collections using the built-in `json_value` function. Varrays are converted to JSON arrays when passed through the JSON constructor while Index by `PLS_INTEGER` collections and nested tables are converted to JSON objects.

Topics

- JSON Arrays to Nested Tables, Index by PLS_INTEGER, and Varray Collections
- Varrays to JSON Arrays

JSON Arrays to Nested Tables, Index by PLS_INTEGER, and Varray Collections

When a nested table, index by `PLS_INTEGER`, or varray collection is specified in the `RETURNING` clause, `json_value` converts the input JSON array to the PL/SQL collection type and returns an instance of the PL/SQL collection. If the input JSON is not a JSON array, a `MISMATCH` error is raised.

To convert a JSON array into a PL/SQL collection, the JSON array elements are inserted one by one into the collection. Insertion begins with the first element in the JSON array inserted at index `1` of the PL/SQL collection and ends when the last JSON array element is inserted into the collection. The collection index is incremented by `1` for each inserted element.

- A JSON null element results in a PL/SQL `NULL` element being inserted into the collection.
- If the number of elements in a JSON array exceeds the size of its corresponding varray, a `MISMATCH` error is raised.
- If the JSON element types are not convertible to the PL/SQL collection element type, a `MISMATCH` error is raised.

Example 4-8 Convert a JSON Array to an Index by PLS_INTEGER Collection

This example converts a JSON array to an index by `PLS_INTEGER` collection using the built-in function `json_value`.

```
DECLARE
    TYPE theIBPLS IS TABLE OF NUMBER INDEX BY PLS_INTEGER;
    myIBPLS theIBPLS;
BEGIN
    myIBPLS := JSON_VALUE(JSON('[1, 2, 3, 4, 5]'), '$' RETURNING theIBPLS);
END;
/
```

Running this PL/SQL block results in `myIBPLS` having the following value:

```
theIBPLS(1=>1, 2=>2, 3=>3, 4=>4, 5=>5)
```

Example 4-9 Convert a JSON Array to a Varray

This example converts a JSON array to a varray using the built-in function `json_value`.

```
DECLARE
    TYPE theVARRAY IS VARRAY(5) OF NUMBER;
    myVARRAY theVARRAY;
```

```
BEGIN
    myVARRAY := JSON_VALUE(JSON('[1, 2, 3, 4, 5]'), '$' RETURNING theVARRAY);
END;
/
```

Running this PL/SQL block results in `myVARRAY` having the following value:

```
theVARRAY(1=>1, 2=>2, 3=>3, 4=>4, 5=>5)
```

Example 4-10 Convert a JSON Array to a Nested Table

This example converts a JSON array to a nested table using the built-in function `json_value`.

```
DECLARE
    TYPE theNESTEDTABLE IS TABLE OF NUMBER;
    myNESTEDTABLE theNESTEDTABLE;
BEGIN
    myNESTEDTABLE := JSON_VALUE(JSON('[1, 2, 3, 4, 5]'), '$' RETURNING
theNESTEDTABLE);
END;
/
```

Running this PL/SQL block results in `myNESTEDTABLE` having the following value:

```
theNESTEDTABLE(1=>1, 2=>2, 3=>3, 4=>4, 5=>5)
```

Varrays to JSON Arrays

Varrays are converted to JSON arrays when they are passed to a JSON constructor.

When varrays are converted to JSON arrays, each element of the collection is inserted into the JSON array beginning with the element at the smallest collection index and ending with the element at the largest collection index. The indices are not transferred into the JSON array, only the element value.

When passed to the JSON constructor, Index by `PLS_INTEGER` collections and nested types are converted to JSON objects rather than JSON arrays.

Example 4-11 Convert a Varray to a JSON Array

```
DECLARE
    TYPE theVarray IS VARRAY(4) OF NUMBER;
    myVarray theVarray := theVarray(1, 2, 3, null);
    myJSON JSON;
BEGIN
    myJSON := JSON(myVarray);
    DBMS_OUTPUT.PUT_LINE(JSON_SERIALIZE(myJSON));
END;
/
```

Result:

```
[1, 2, 3, null]
```

JSON Objects and Associative Arrays

Associative arrays can be converted to and from JSON objects using the JSON constructor and the built-in function `json_value`, respectively.

Topics

- JSON Objects to Associative Arrays
- Associative Arrays to JSON Objects

JSON Objects to Associative Arrays

When JSON objects are mapped into associative arrays, each JSON key name and value pair is inserted into the associative array based on the ordering and or collection of the associative array.

Associative array key names are case sensitive and the insert preserves the case of the JSON key name. The JSON value for the key is converted as necessary to the associative array element type and the key name/value pair is then inserted into the associative array.

Similar to SQL objects and PL/SQL records, a JSON value can be a nested object or an array and must be convertible to the associative array element type. If the value types are not convertible, a `MISMATCH` error is raised.

Example 4-12 Convert a JSON Object to an Associative Array

This example converts a JSON object to an associative array using the built-in function `json_value`.

```
DECLARE
    TYPE theASCARRAY IS TABLE OF NUMBER INDEX BY VARCHAR2(10);
    myAscArray theASCARRAY;
BEGIN
    myAscArray := JSON_VALUE(JSON('{"Key1":10, "Key2":20}'), '$' RETURNING
theASCARRAY);
END;
/
```

Running this PL/SQL block will result in an associative with two elements:

```
theASCARRAY('Key1'=>10, 'Key2'=>20)
```

Associative Arrays to JSON Objects

The process of converting an associative array to a JSON object consists of inserting every associative array key and value into the JSON object as a name/value pair. The ordering of insertions may not matter because all key names in PL/SQL associative arrays are unique and the ordering of JSON attributes is not specified in the JSON standards. However, the key values will likely be inserted based on the internal sorted order or collation of the associative array.

Because associative arrays have `varchar2` keys, the key type inserted into the JSON object is a JSON string. The case of the key in the associative array is preserved in the copy to the JSON object.

The value of the associative array element is copied into the JSON object following the key. If the element type of the associative array is a nested aggregate type, a JSON object or array matching the aggregate type is created as the JSON value.

Example 4-13 Convert an Associative Array to a JSON Object

This example converts an associative array to a JSON object using the JSON constructor.

```
DECLARE
    TYPE AsscArray IS TABLE OF VARCHAR2(10) INDEX BY VARCHAR2(10);
    myAsscArray AsscArray := AsscArray('FIRST_NAME' => 'Bob', 'LAST_NAME' =>
'Jones');
    myJson JSON;
BEGIN
    myJson := JSON(myAsscArray);
    DBMS_OUTPUT.PUT_LINE(JSON_SERIALIZE(myJson));
END;
/
```

Running this PL/SQL block will result in a JSON object with the following values:

```
{"FIRST_NAME":"Bob", "LAST_NAME":"Jones"}
```

VECTOR Data Type

A vector value is an array of non-null numeric values, all of which are of the same numeric type. A vector in PL/SQL has two storage attributes; the number of values constituting the vector is its dimension and the numeric type of the values is its format.

A vector variable in PL/SQL holds a vector value and can be flexible, partially flexible, or fully inflexible in terms of dimension, dimension format, and storage format. Assignment of a vector value to a flexible PL/SQL vector variable always succeeds and no conversion occurs. An error occurs during assignment if the value and the variable differ in the inflexible attribute(s). In all cases, the elements stored in the vector must be of the same numeric type.

If specified, the format of a vector must be one of the following: `FLOAT64`, `FLOAT32`, `INT8`, or `BINARY`. The dimension of a `BINARY` vector must be a multiple of 8. For more information about using `BINARY` vectors, see *Oracle Database AI Vector Search User's Guide*.

A storage format attribute of either `SPARSE` or `DENSE` can optionally be included in vector declarations. If not specified, vectors are `DENSE` by default.

A sparse vector is a vector where the vast majority of dimension values are zero. Using sparse vectors in cases when you expect most dimension values to be zero can save storage space, as only the index values that are non-zero are stored. The string representation of a `SPARSE` vector is a JSON array composed of three parts; a dimension value, a JSON array of indexes (as non-negative integers) representing non-zero dimension values, and a JSON array of those non-zero values. For example, the following sparse representation describes a vector of 128 dimensions with the values 8 and 24 at the fourth and sixth dimensions. Every other dimension value is understood to be zero.

```
[128, [4, 6], [8, 24]]
```

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

Note:

Checks of the dimension, dimension format, and storage format are completed at runtime.

The PL/SQL VECTOR data type appears as its own distinct scalar type family and can be used with PL/SQL operators, passed to PL/SQL procedures and functions, set to NULL, and otherwise used in the same way as any other data type in PL/SQL. Note that although a vector variable can hold a NULL vector, the value(s) in the vector cannot be NULL.

Assignment semantics and handling of implicit conversion in PL/SQL differ from SQL. While SQL requires an exact match only for dimension, PL/SQL requires both format and dimension to match for a successful assignment. Additionally, SQL allows for implicit conversion between VECTOR and string types while PL/SQL does not support implicit conversion between vectors and any other type. Neither SQL nor PL/SQL support equality comparisons of vectors.

If a variable is declared using %TYPE on a vector variable or a vector column, the declared variable will be a vector that inherits the storage attributes of the referenced vector variable or column. The following example demonstrates this concept:

```
CREATE TABLE PLS_VEC_TAB(
    v1 vector,
    v2 vector(100),
    v3 vector(*, INT8),
    v4 vector(100, INT8),
    v5 vector(1024, BINARY),
    v6 vector(100, FLOAT32, DENSE),
    v7 vector(100, FLOAT32, SPARSE)
);

DECLARE
    vec0 vector;                   -- dimension and format are flexible, 
storage format is DENSE
    vec1 PLS_VEC_TAB.v1%TYPE;     -- dimension and format are flexible, 
storage format is DENSE
    vec2 PLS_VEC_TAB.v2%TYPE;     -- dimension is 100, format is flexible, 
storage format is DENSE
    vec3 PLS_VEC_TAB.v3%TYPE;     -- dimension is flexible, format is INT8, 
storage format is DENSE
    vec4 PLS_VEC_TAB.v4%TYPE;     -- dimension is 100, format is INT8, storage 
format is DENSE
    vec5 PLS_VEC_TAB.v5%TYPE;     -- dimension is 1024, format is BINARY, 
storage format is DENSE
    vec6 PLS_VEC_TAB.v6%TYPE;     -- dimension is 100, format is FLOAT32, 
storage format is DENSE
    vec7 PLS_VEC_TAB.v7%TYPE;     -- dimension is 100, format is FLOAT32, 
storage format is SPARSE

    vec_0 vec0%TYPE;     -- dimension and format are flexible, storage format 
is DENSE
    vec_1 vec1%TYPE;     -- dimension and format are flexible, storage format 
is DENSE
    vec_2 vec2%TYPE;     -- dimension is 100, format is flexible, storage 
format is DENSE
```

```
    vec_3 vec3%TYPE;    -- dimension is flexible, format is INT8, storage
format is DENSE
    vec_4 vec4%TYPE;    -- dimension is 100, format is INT8, storage format
is DENSE
    vec_5 vec5%TYPE;    -- dimension is 1024, format is BINARY, storage
format is DENSE
    vec_6 vec6%TYPE;    -- dimension is 100, format is FLOAT32, storage
format is DENSE
    vec_7 vec7%TYPE;    -- dimension is 100, format is FLOAT32, storage
format is SPARSE
BEGIN
    NULL;
END;
/
```

You can use the `VECTOR` data type and its instances in most places where a SQL data type is allowed, including:

- As an element or field type in records, PL/SQL collections, and `%ROWTYPE` attributes. Note that a `%ROWTYPE` attribute inherits both the dimension and format of an underlying vector column.
- In PL/SQL triggers, including with the pseudorecords `OLD` and `NEW` and in the `WHEN` clause of a conditional trigger.
- The `USING` clause supports vectors in all three bind directions; `IN`, `IN OUT`, and `OUT`.
- In the `FORALL` clause, the `RETURNING INTO` clause, and the `BULK COLLECT INTO` clause.
- As the arguments to an addition, subtraction, or multiplication operation.

The `VECTOR` data type is not currently supported in SQL collections or objects.

A variable of type `VECTOR` can come in the form of static SQL, dynamic SQL, or using `DBMS_SQL`. In all of these cases, a vector can appear as the column type, the bind type, or both. If the column and bind type are not both `VECTOR`, the remaining side must be a string type. Note that PL/SQL functions that use formal arguments of the `VECTOR` data type are not currently supported in the `WITH` clause of a SQL `SELECT` statement.

PL/SQL will use the definition of the identifier `VECTOR` in the innermost scope in which it appears. If a user-defined type is created with the name `VECTOR` and then is referenced without a name prefix, PL/SQL interprets the variable using the local definition. If no local definition exists, PL/SQL expands its search to outer scopes until the name is resolved, eventually to the Package STANDARD definition.

Topics

- VECTOR Operations Supported by PL/SQL
- VECTOR Data Type PL/SQL Code Examples

See Also:

- *Oracle Database AI Vector Search User's Guide* for information about Oracle AI Vector Search
- *Oracle Database SQL Language Reference* for more information about the `VECTOR` data type

VECTOR Operations Supported by PL/SQL

PL/SQL natively supports the following base operations for use with the `VECTOR` data type:

- `VECTOR`
- `TO_VECTOR`
- `FROM_VECTOR` (and `VECTOR_SERIALIZE`)[1]
- `TO_CHAR`
- `TO_CLOB`
- `VECTOR_DIMENSION_COUNT` (and `VECTOR_DIMS`)
- `VECTOR_DIMENSION_FORMAT`
- `VECTOR_NORM`
- `VECTOR_DISTANCE` is supported with the following metric options (cosine distance is the default if no metric is specified):
  - `COSINE`
  - `MANHATTAN`
  - `EUCLIDEAN`
  - `EUCLIDEAN_SQUARED`
  - `DOT`
  - `HAMMING`
  - `JACCARD`
- The following vector distance functions are also natively supported as standalone functions in PL/SQL:
  - `COSINE_DISTANCE`
  - `L1_DISTANCE` (Manhattan distance)
  - `L2_DISTANCE` (Euclidean distance)
  - `INNER_PRODUCT`
- Additionally, the following shorthand distance operators are available for the corresponding vector distances:
  - Cosine distance: `<=>`
  - Euclidean distance: `<->`

[1] The optional `RETURNING` clause is not supported with `TO_VECTOR` and `VECTOR_SERIALIZE`.

  – Dot product: <#>

Note that the corresponding vector distance metrics, standalone functions, and shorthand distance operators will have equivalent results. For example, VECTOR_DISTANCE(v1, v2, COSINE) is equal to COSINE_DISTANCE(v1, v2) is equal to v1 <=> v2.

To construct a vector in PL/SQL, use VECTOR or TO_VECTOR. For example, see the following variable assignments:

```
v VECTOR := VECTOR('[1, 2, 3]');
v VECTOR := TO_VECTOR('[1, 2, 3]');
```

The ON CONVERSION ERROR clause used by SQL in explicit conversions to determine a default value if conversion fails is not supported by PL/SQL. Instead, a default value can be set using a code block in the exception handler.

You can use the VECTOR_DISTANCE function with metric keyword natively in PL/SQL, use the previously listed distance functions, or call VECTOR_DISTANCE from static SQL. The distance is returned as a BINARY_DOUBLE. Consider the following valid assignments:

```
dist := COSINE_DISTANCE(v1, v2);
dist := VECTOR_DISTANCE(v1, v2, COSINE);
dist := v1 <=> v2;
SELECT VECTOR_DISTANCE(v1, v2, COSINE) INTO dist;
SELECT v1 <=> v2 INTO dist;
```

Arithmetic operators for addition, subtraction, and multiplication can be applied to vectors dimension-wise. Both sides of the operation must evaluate to vectors with matching dimensions and must not be BINARY or SPARSE vectors. The resulting vector has the same number of dimensions as the operands and the format is determined based on the formats of the inputs. If one side of the operation is not a vector, an attempt is made automatically to convert the value to a vector. If the conversion fails, an error is raised.

The format used for the result is ranked in the following order: flexible, FLOAT64, FLOAT32, then INT8. As in, if either side of the operation has a flexible format, the result will be flexible, otherwise, if either side has the format FLOAT64, the result will be FLOAT64, and so on.

The syntax for arithmetic operators is as follows:

- Addition: expr1 + expr2
- Subtraction: expr1 - expr2
- Multiplication: expr1 * expr2

If either side of the arithmetic operation is NULL, the result is NULL. In the case of dimension overflow, an error is raised. For example, adding VECTOR('[1, 127]', 2, INT8) to VECTOR('[1, 1]', 2, INT8) results in an error because 127+1=128, which overflows the INT8 format.

See Also:

- *Oracle Database SQL Language Reference* for syntax and semantic information about vector SQL functions
- *Oracle Database AI Vector Search User's Guide* for more information about performing arithmetic operations with vectors

VECTOR Data Type PL/SQL Code Examples

The PL/SQL code examples provided here show how to use the `VECTOR` data type.

Example 4-14 Use the VECTOR Data Type with PL/SQL

The first part of this example demonstrates how to select a vector into a PL/SQL vector variable, in this case using `%TYPE` on a vector column.

```
DROP TABLE theVectorTable;
CREATE TABLE theVectorTable (embedding VECTOR(3, float32), id NUMBER);
INSERT INTO theVectorTable VALUES ('[1.11, 2.22, 3.33]', 1);
INSERT INTO theVectorTable VALUES ('[4.44, 5.55, 6.66]', 2);
INSERT INTO theVectorTable VALUES ('[7.77, 8.88, 9.99]', 3);

SET SERVEROUTPUT ON;

DECLARE
  v_embedding theVectorTable.embedding%TYPE;
BEGIN
  SELECT embedding INTO v_embedding FROM theVectorTable WHERE id=3;
  DBMS_OUTPUT.PUT_LINE('Embedding is ' || FROM_VECTOR(v_embedding));
END;
/
```

Result:

```
Embedding is [7.76999998E+000,8.88000011E+000,9.98999977E+000]
```

The following anonymous block uses a cursor with bulk fetch to capture `theVectorTable`'s vector and id data into a `table%ROWTYPE` index table.

```
DECLARE
  TYPE vecTabT IS TABLE OF theVectorTable%ROWTYPE INDEX BY BINARY_INTEGER;
  v_vecTabT vecTabT;
  CURSOR c IS SELECT * FROM theVectorTable;
BEGIN
  OPEN c;
  FETCH c BULK COLLECT INTO v_vecTabT;
  CLOSE c;

  -- display the contents of the vector index table
  FOR i IN 1..v_vecTabT.LAST LOOP
    DBMS_OUTPUT.PUT_LINE('Embedding ID ' || v_vecTabT(i).id || ': ' ||
```

```
            FROM_VECTOR(v_vecTabT(i).embedding));
  END LOOP;
END;
/
```

Result:

```
Embedding ID 1: [1.11000001E+000,2.22000003E+000,3.32999992E+000]
Embedding ID 2: [4.44000006E+000,5.55000019E+000,6.65999985E+000]
Embedding ID 3: [7.76999998E+000,8.88000011E+000,9.98999977E+000]
```

Example 4-15 Use the VECTOR Data Type with a PL/SQL Trigger

This example creates a `BEFORE UPDATE` trigger on `theVectorTable` that inserts vector values into `vecLogTable`:

```
DROP TABLE vecLogTable;
DROP SEQUENCE vecTrgSeq;
CREATE TABLE vecLogTable (embedding VECTOR(3, float32),
        describe VARCHAR2(25), seq NUMBER);
CREATE SEQUENCE vecTrgSeq;

CREATE OR REPLACE TRIGGER vecTrg
BEFORE UPDATE ON theVectorTable
FOR EACH ROW
BEGIN
  INSERT INTO vecLogTable VALUES (:old.embedding, 'OLD.VECTRG',
          vecTrgSeq.NEXTVAL);
  INSERT INTO vecLogTable VALUES (:new.embedding, 'NEW.VECTRG',
          vecTrgSeq.NEXTVAL);
END;
/

UPDATE theVectorTable SET embedding='[2.22, 4.44, 6.66]' WHERE id=2;
SELECT * FROM vecLogTable ORDER BY seq;
```

Result:

```
EMBEDDING                                          DESCRIBE         SEQ
-------------------------------------------------- ---------------- ---
[4.44000006E+000,5.55000019E+000,6.65999985E+000]  OLD.VECTRG         1
[2.22000003E+000,4.44000006E+000,6.65999985E+000]  NEW.VECTRG         2
```

Example 4-16 Use Vector Distance Functions with PL/SQL

This example demonstrates PL/SQL support for vector distance functions.

```
DECLARE
  v1 VECTOR := TO_VECTOR('[1, 2, 3]');
  v2 VECTOR := TO_VECTOR('[4, 5, 6]');
  v3 VECTOR := TO_VECTOR('[1, 2, 0, 6]', *, BINARY);
  v4 VECTOR := TO_VECTOR('[0, 6, 0, 3]', *, BINARY);
```

```
  man_dist NUMBER;
  euc_dist NUMBER;
  cos_dist NUMBER;
  inn_dist NUMBER;
  ham_dist NUMBER;
  dot_dist NUMBER;
  jac_dist NUMBER;
BEGIN
  man_dist := L1_DISTANCE(v1, v2); --Manhattan Distance
  euc_dist := L2_DISTANCE(v1, v2); --Euclidean Distance
  cos_dist := COSINE_DISTANCE(v1, v2); --Cosine Distance
  inn_dist := INNER_PRODUCT(v1, v2); --Inner Product

  --The Hamming Distance has no standalone function in PL/SQL
  ham_dist := VECTOR_DISTANCE(v1, v2, HAMMING);

  --The Negative Inner (Dot) Product has no standalone function in PL/SQL
  dot_dist := VECTOR_DISTANCE(v1, v2, DOT);

  --The Jaccard Distance has no standalone function in PL/SQL
  jac_dist := VECTOR_DISTANCE(v3, v4, JACCARD);

  DBMS_OUTPUT.PUT_LINE('The Manhattan distance is: ' || man_dist);
  DBMS_OUTPUT.PUT_LINE('The Euclidean distance is: ' || euc_dist);
  DBMS_OUTPUT.PUT_LINE('The Cosine distance is: ' || cos_dist);
  DBMS_OUTPUT.PUT_LINE('The Inner Product is: ' || inn_dist);
  DBMS_OUTPUT.PUT_LINE('The Hamming distance is: ' || ham_dist);
  DBMS_OUTPUT.PUT_LINE('The Dot Product is: ' || dot_dist);
  DBMS_OUTPUT.PUT_LINE('The Jaccard Distance between the BINARY vectors v3 
and v4 is: ' || jac_dist);
END;
/
```

Result:

```
The Manhattan distance is: 9
The Euclidean distance is: 5.1961524227066329
The Cosine distance is: .025368153802923787
The Inner Product is: 32
The Hamming distance is: 3
The Dot Product is: -32
The Jaccard Distance between the BINARY vectors v3 and v4 
is: .66666666666666674
```

Example 4-17 Use Shorthand Distance Operators

Note that because PL/SQL does not support implicit conversion with vectors, you must construct the vectors before the variable assignment or in the same line. This is the same behavior as the other distance functions in PL/SQL.

```
DECLARE
  v1 VECTOR := VECTOR('[1, 2, 3]');
  v2 VECTOR := VECTOR('[4, 5, 6]');
  cos_dist BINARY_DOUBLE;
  euc_dist BINARY_DOUBLE;
```

```
  dot_dist BINARY_DOUBLE;
BEGIN
  cos_dist := v1 <=> v2;
  euc_dist := v1 <-> v2;
  dot_dist := v1 <#> v2;

  DBMS_OUTPUT.PUT_LINE(cos_dist);
  DBMS_OUTPUT.PUT_LINE(euc_dist);
  DBMS_OUTPUT.PUT_LINE(dot_dist);
END;
/
```

Result:

```
2.5368153802923787E-002
5.1961524227066329E+000
-3.2E+001
```

Example 4-18 Perform Arithmetic Operations on Vectors

```
DECLARE
  v1 VECTOR := VECTOR('[10, 20, 30]', 3, INT8);
  v2 VECTOR := VECTOR('[6, 4, 2]', 3, INT8);
BEGIN
  DBMS_OUTPUT.PUT_LINE(TO_CHAR(v1 + v2));
  DBMS_OUTPUT.PUT_LINE(TO_CHAR(v1 - v2));
  DBMS_OUTPUT.PUT_LINE(TO_CHAR(v1 * v2));
END;
/
```

Result:

```
[16,24,32]
[4,16,28]
[60,80,60]
```

Example 4-19 Declare DENSE and SPARSE Vectors in PL/SQL

```
DECLARE
  vs1 VECTOR(*, *, SPARSE) := VECTOR('[10, [0, 3, 4, 6, 7], [1.9, 4, 7.2, 30,
60]]', *, *, SPARSE);
  vs2 VECTOR(*, *, SPARSE) := VECTOR('[10, [1, 3, 4, 8, 9], [4.5, 7.6, 4,
8.1, 5]]', *, *, SPARSE);

  vd1 VECTOR(*, *, DENSE) := VECTOR('[1.9, 0, 0, 4, 7.2, 0, 30, 60, 0, 0]',
*, *, DENSE);
  vd2 VECTOR(*, *, DENSE) := VECTOR('[0, 4.5, 0, 7.6, 4, 0, 0, 0, 8.1, 5]',
*, *, DENSE);
BEGIN
  DBMS_OUTPUT.PUT_LINE('Vector Distance Sparse: ' ||
TRUNC(VECTOR_DISTANCE(vs1, vs2), 5));
  DBMS_OUTPUT.PUT_LINE('Vector Distance Dense:  ' ||
TRUNC(VECTOR_DISTANCE(vd1, vd2), 5));
```

```
END;
/
```

Result:

```
Vector Distance Sparse: .93556
Vector Distance Dense:  .93556
```

CHAR and VARCHAR2 Variables

Topics

- Assigning or Inserting Too-Long Values
- Declaring Variables for Multibyte Characters
- Differences Between CHAR and VARCHAR2 Data Types

Assigning or Inserting Too-Long Values

If the value that you assign to a character variable is longer than the maximum size of the variable, an error occurs. For example:

```
DECLARE
  c VARCHAR2(3 CHAR);
BEGIN
  c := 'abc  ';
END;
/
```

Result:

```
DECLARE
*
ERROR at line 1:
ORA-06502: PL/SQL: value or conversion error: character string buffer too small
ORA-06512: at line 4
```

Similarly, if you insert a character variable into a column, and the value of the variable is longer than the defined width of the column, an error occurs. For example:

```
DROP TABLE t;
CREATE TABLE t (c CHAR(3 CHAR));

DECLARE
  s VARCHAR2(5 CHAR) := 'abc  ';
BEGIN
  INSERT INTO t(c) VALUES(s);
END;
/
```

Result:

```
BEGIN
*
ERROR at line 1:
ORA-12899: value too large for column "HR"."T"."C" (actual: 5, maximum: 3)
ORA-06512: at line 4
```

To strip trailing blanks from a character value before assigning it to a variable or inserting it into a column, use the `RTRIM` function, explained in *Oracle Database SQL Language Reference*. For example:

```
DECLARE
  c VARCHAR2(3 CHAR);
BEGIN
  c := RTRIM('abc  ');
  INSERT INTO t(c) VALUES(RTRIM('abc  '));
END;
/
```

Result:

```
PL/SQL procedure successfully completed.
```

Declaring Variables for Multibyte Characters

The maximum *size* of a `CHAR` or `VARCHAR2` variable is 32,767 bytes, whether you specify the maximum size in characters or bytes. The maximum *number of characters* in the variable depends on the character set type and sometimes on the characters themselves:

Character Set Type	Maximum Number of Characters
Single-byte character set	32,767
n-byte fixed-width multibyte character set (for example, AL16UTF16)	`FLOOR(32,767/`*n*`)`
n-byte variable-width multibyte character set with character widths between 1 and *n* bytes (for example, JA16SJIS or AL32UTF8)	Depends on characters themselves—can be anything from 32,767 (for a string containing only 1-byte characters) through `FLOOR(32,767/`*n*`)` (for a string containing only *n*-byte characters).

When declaring a `CHAR` or `VARCHAR2` variable, to ensure that it can always hold *n* characters in any multibyte character set, declare its length in characters—that is, `CHAR(`*n* `CHAR)` or `VARCHAR2(`*n* `CHAR)`, where *n* does not exceed `FLOOR(32767/4)` = 8191.

> **See Also:**
>
> *Oracle Database Globalization Support Guide* for information about Oracle Database character set support

Differences Between CHAR and VARCHAR2 Data Types

`CHAR` and `VARCHAR2` data types differ in:

- Predefined Subtypes
- How Blank-Padding Works
- Value Comparisons

Predefined Subtypes

The `CHAR` data type has one predefined subtype in both PL/SQL and SQL—`CHARACTER`.

The VARCHAR2 data type has one predefined subtype in both PL/SQL and SQL, VARCHAR, and an additional predefined subtype in PL/SQL, STRING.

Each subtype has the same range of values as its base type.

> **Note:**
>
> In a future PL/SQL release, to accommodate emerging SQL standards, VARCHAR might become a separate data type, no longer synonymous with VARCHAR2.

How Blank-Padding Works

This explains the differences and considerations of using blank-padding with CHAR and VARCHAR2.

Consider these situations:

- The value that you assign to a variable is shorter than the maximum size of the variable.
- The value that you insert into a column is shorter than the defined width of the column.
- The value that you retrieve from a column into a variable is shorter than the maximum size of the variable.

If the data type of the receiver is CHAR, PL/SQL blank-pads the value to the maximum size. Information about trailing blanks in the original value is lost.

If the data type of the receiver is VARCHAR2, PL/SQL neither blank-pads the value nor strips trailing blanks. Character values are assigned intact, and no information is lost.

Example 4-20 CHAR and VARCHAR2 Blank-Padding Difference

In this example, both the CHAR variable and the VARCHAR2 variable have the maximum size of 10 characters. Each variable receives a five-character value with one trailing blank. The value assigned to the CHAR variable is blank-padded to 10 characters, and you cannot tell that one of the six trailing blanks in the resulting value was in the original value. The value assigned to the VARCHAR2 variable is not changed, and you can see that it has one trailing blank.

```
DECLARE
  first_name  CHAR(10 CHAR);
  last_name   VARCHAR2(10 CHAR);
BEGIN
  first_name := 'John ';
  last_name  := 'Chen ';

  DBMS_OUTPUT.PUT_LINE('*' || first_name || '*');
  DBMS_OUTPUT.PUT_LINE('*' || last_name || '*');
END;
/
```

Result:

```
*John      *
*Chen *
```

Value Comparisons

The SQL rules for comparing character values apply to PL/SQL character variables.

Whenever one or both values in the comparison have the data type VARCHAR2 or NVARCHAR2, nonpadded comparison semantics apply; otherwise, blank-padded semantics apply. For more information, see *Oracle Database SQL Language Reference*.

LONG and LONG RAW Variables

> **Note:**
>
> Oracle supports the LONG and LONG RAW data types only for backward compatibility with existing applications. For new applications:
>
> - Instead of LONG, use VARCHAR2(32760), BLOB, CLOB or NCLOB.
> - Instead of LONG RAW, use RAW(32760) or BLOB.
>
> For information about how to migrate columns from LONG data types to LOB data types, see *Oracle Database SecureFiles and Large Objects Developer's Guide*.

You can insert any LONG value into a LONG column. You can insert any LONG RAW value into a LONG RAW column. You cannot retrieve a value longer than 32,760 bytes from a LONG or LONG RAW column into a LONG or LONG RAW variable.

You can insert any CHAR or VARCHAR2 value into a LONG column. You cannot retrieve a value longer than 32,767 bytes from a LONG column into a CHAR or VARCHAR2 variable.

You can insert any RAW value into a LONG RAW column. You cannot retrieve a value longer than 32,767 bytes from a LONG RAW column into a RAW variable.

> **See Also:**
>
> "Trigger LONG and LONG RAW Data Type Restrictions" for restrictions on LONG and LONG RAW data types in triggers

ROWID and UROWID Variables

When you retrieve a rowid into a ROWID variable, use the ROWIDTOCHAR function to convert the binary value to a character value. For information about this function, see *Oracle Database SQL Language Reference*.

To convert the value of a ROWID variable to a rowid, use the CHARTOROWID function, explained in *Oracle Database SQL Language Reference*. If the value does not represent a valid rowid, PL/SQL raises the predefined exception SYS_INVALID_ROWID.

To retrieve a rowid into a UROWID variable, or to convert the value of a UROWID variable to a rowid, use an assignment statement; conversion is implicit.

Note:

- `UROWID` is a more versatile data type than `ROWID`, because it is compatible with both logical and physical rowids.
- When you update a row in a table compressed with Hybrid Columnar Compression (HCC), the `ROWID` of the row changes. HCC, a feature of certain Oracle storage systems, is described in *Oracle Database Concepts*.

See Also:

Oracle Database PL/SQL Packages and Types Reference for information about the `DBMS_ROWID` package, whose subprograms let you create and return information about `ROWID` values (but not `UROWID` values)

PLS_INTEGER and BINARY_INTEGER Data Types

The PL/SQL data types `PLS_INTEGER` and `BINARY_INTEGER` are identical.

For simplicity, this document uses `PLS_INTEGER` to mean both `PLS_INTEGER` and `BINARY_INTEGER`.

The `PLS_INTEGER` data type stores signed integers in the range -2,147,483,648 through 2,147,483,647, represented in 32 bits.

The `PLS_INTEGER` data type has these advantages over the `NUMBER` data type and `NUMBER` subtypes:

- `PLS_INTEGER` values require less storage.
- `PLS_INTEGER` operations use hardware arithmetic, so they are faster than `NUMBER` operations, which use library arithmetic.

For efficiency, use `PLS_INTEGER` values for all calculations in its range.

Topics

- Preventing PLS_INTEGER Overflow
- Predefined PLS_INTEGER Subtypes
- SIMPLE_INTEGER Subtype of PLS_INTEGER

Preventing PLS_INTEGER Overflow

A calculation with two `PLS_INTEGER` values that overflows the `PLS_INTEGER` range raises an overflow exception.

For calculations outside the `PLS_INTEGER` range, use `INTEGER`, a predefined subtype of the `NUMBER` data type.

Example 4-21 PLS_INTEGER Calculation Raises Overflow Exception

This example shows that a calculation with two `PLS_INTEGER` values that overflows the `PLS_INTEGER` range raises an overflow exception, even if you assign the result to a `NUMBER` data type.

```
DECLARE
  p1 PLS_INTEGER := 2147483647;
  p2 PLS_INTEGER := 1;
  n NUMBER;
BEGIN
  n := p1 + p2;
END;
/
```

Result:

```
DECLARE
*
ERROR at line 1:
ORA-01426: numeric overflow
ORA-06512: at line 6
```

Example 4-22 Preventing Example 4-21 Overflow

This example shows the correct use of the `INTEGER` predefined subtype for calculations outside the `PLS_INTEGER` range.

```
DECLARE
  p1 PLS_INTEGER := 2147483647;
  p2 INTEGER := 1;
  n NUMBER;
BEGIN
  n := p1 + p2;
END;
/
```

Result:

```
PL/SQL procedure successfully completed.
```

Predefined PLS_INTEGER Subtypes

This summary lists the predefined subtypes of the `PLS_INTEGER` data type and describes the data they store.

Table 4-3 Predefined Subtypes of PLS_INTEGER Data Type

Data Type	Data Description
`NATURAL`	Nonnegative `PLS_INTEGER` value
`NATURALN`	Nonnegative `PLS_INTEGER` value with `NOT NULL` constraint
`POSITIVE`	Positive `PLS_INTEGER` value
`POSITIVEN`	Positive `PLS_INTEGER` value with `NOT NULL` constraint
`SIGNTYPE`	`PLS_INTEGER` value -1, 0, or 1 (useful for programming tri-state logic)
`SIMPLE_INTEGER`	`PLS_INTEGER` value with `NOT NULL` constraint.

`PLS_INTEGER` and its subtypes can be implicitly converted to these data types:

- CHAR
- VARCHAR2
- NUMBER
- LONG

All of the preceding data types except LONG, and all PLS_INTEGER subtypes, can be implicitly converted to PLS_INTEGER.

A PLS_INTEGER value can be implicitly converted to a PLS_INTEGER subtype only if the value does not violate a constraint of the subtype.

See Also:

- "NOT NULL Constraint"for information about the NOT NULL constraint
- "SIMPLE_INTEGER Subtype of PLS_INTEGER" for more information about SIMPLE_INTEGER

Example 4-23 Violating Constraint of SIMPLE_INTEGER Subtype

This example shows that casting the PLS_INTEGER value NULL to the SIMPLE_INTEGER subtype raises an exception.

```
DECLARE
  a SIMPLE_INTEGER := 1;
  b PLS_INTEGER := NULL;
BEGIN
  a := b;
END;
/
```

Result:

```
DECLARE
*
ERROR at line 1:
ORA-06502: PL/SQL: value or conversion error
ORA-06512: at line 5
```

SIMPLE_INTEGER Subtype of PLS_INTEGER

SIMPLE_INTEGER is a predefined subtype of the PLS_INTEGER data type.

SIMPLE_INTEGER has the same range as PLS_INTEGER and has a NOT NULL constraint. It differs significantly from PLS_INTEGER in its overflow semantics.

If you know that a variable will never have the value NULL or need overflow checking, declare it as SIMPLE_INTEGER rather than PLS_INTEGER. Without the overhead of checking for nullness and overflow, SIMPLE_INTEGER performs significantly better than PLS_INTEGER.

Topics

- SIMPLE_INTEGER Overflow Semantics
- Expressions with Both SIMPLE_INTEGER and Other Operands

- Integer Literals in SIMPLE_INTEGER Range

> **See Also:**
>
> "NOT NULL Constraint"

SIMPLE_INTEGER Overflow Semantics

If and only if all operands in an expression have the data type `SIMPLE_INTEGER`, PL/SQL uses two's complement arithmetic and ignores overflows.

Because overflows are ignored, values can wrap from positive to negative or from negative to positive; for example:

$2^{30} + 2^{30}$ = 0x40000000 + 0x40000000 = 0x80000000 = -2^{31}

$-2^{31} + -2^{31}$ = 0x80000000 + 0x80000000 = 0x00000000 = 0

For example, this block runs without errors:

```
DECLARE
  n SIMPLE_INTEGER := 2147483645;
BEGIN
  FOR j IN 1..4 LOOP
    n := n + 1;
    DBMS_OUTPUT.PUT_LINE(TO_CHAR(n, 'S9999999999'));
  END LOOP;
  FOR j IN 1..4 LOOP
   n := n - 1;
   DBMS_OUTPUT.PUT_LINE(TO_CHAR(n, 'S9999999999'));
  END LOOP;
END;
/
```

Result:

```
+2147483646
+2147483647
-2147483648
-2147483647
-2147483648
+2147483647
+2147483646
+2147483645

PL/SQL procedure successfully completed.
```

Expressions with Both SIMPLE_INTEGER and Other Operands

If an expression has both `SIMPLE_INTEGER` and other operands, PL/SQL implicitly converts the `SIMPLE_INTEGER` values to `PLS_INTEGER NOT NULL`.

The PL/SQL compiler issues a warning when `SIMPLE_INTEGER` and other values are mixed in a way that might negatively impact performance by inhibiting some optimizations.

Integer Literals in SIMPLE_INTEGER Range

Integer literals in the `SIMPLE_INTEGER` range have the data type `SIMPLE_INTEGER`.

However, to ensure backward compatibility, when all operands in an arithmetic expression are integer literals, PL/SQL treats the integer literals as if they were cast to `PLS_INTEGER`.

User-Defined PL/SQL Subtypes

PL/SQL lets you define your own subtypes.

The base type can be any scalar or user-defined PL/SQL data type specifier such as `CHAR`, `DATE`, or `RECORD` (including a previously defined user-defined subtype).

> **Note:**
>
> The information in this topic applies to both user-defined subtypes and the predefined subtypes listed in PL/SQL Predefined Data Types.

Subtypes can:

- Provide compatibility with ANSI/ISO data types
- Show the intended use of data items of that type
- Detect out-of-range values

Topics

- Unconstrained Subtypes
- Constrained Subtypes
- Subtypes with Base Types in Same Data Type Family

Unconstrained Subtypes

An **unconstrained subtype** has the same set of values as its base type, so it is only another name for the base type.

Therefore, unconstrained subtypes of the same base type are interchangeable with each other and with the base type. No data type conversion occurs.

To define an unconstrained subtype, use this syntax:

```
SUBTYPE subtype_name IS base_type
```

For information about *subtype_name* and *base_type*, see ***subtype***.

An example of an unconstrained subtype, which PL/SQL predefines for compatibility with ANSI, is:

```
SUBTYPE "DOUBLE PRECISION" IS FLOAT
```

Example 4-24 User-Defined Unconstrained Subtypes Show Intended Use

In this example, the unconstrained subtypes `Balance` and `Counter` show the intended uses of data items of their types.

```
DECLARE
  SUBTYPE Balance IS NUMBER;
```

```
  checking_account        Balance(6,2);
  savings_account         Balance(8,2);
  certificate_of_deposit  Balance(8,2);
  max_insured  CONSTANT   Balance(8,2) := 250000.00;

  SUBTYPE Counter IS NATURAL;

  accounts     Counter := 1;
  deposits     Counter := 0;
  withdrawals  Counter := 0;
  overdrafts   Counter := 0;

  PROCEDURE deposit (
    account  IN OUT Balance,
    amount   IN     Balance
  ) IS
  BEGIN
    account  := account + amount;
    deposits := deposits + 1;
  END;

BEGIN
  NULL;
END;
/
```

Constrained Subtypes

A **constrained subtype** has only a subset of the values of its base type.

If the base type lets you specify size, precision and scale, or a range of values, then you can specify them for its subtypes. The subtype definition syntax is:

```
SUBTYPE subtype_name IS base_type
  { precision [, scale ] | RANGE low_value .. high_value } [ NOT NULL ]
```

Otherwise, the only constraint that you can put on its subtypes is `NOT NULL`:

```
SUBTYPE subtype_name IS base_type [ NOT NULL ]
```

> **Note:**
>
> The only base types for which you can specify a range of values are `PLS_INTEGER` and its subtypes (both predefined and user-defined).

A constrained subtype can be implicitly converted to its base type, but the base type can be implicitly converted to the constrained subtype only if the value does not violate a constraint of the subtype.

A constrained subtype can be implicitly converted to another constrained subtype with the same base type only if the source value does not violate a constraint of the target subtype.

> **See Also:**
>
> - "*subtype_definition* ::=" syntax diagram
> - "*subtype*" semantic description
> - "Example 4-23", "Violating Constraint of SIMPLE_INTEGER Subtype"
> - "Formal Parameters of Constrained Subtypes"
> - "NOT NULL Constraint"

Example 4-25 User-Defined Constrained Subtype Detects Out-of-Range Values

In this example, the constrained subtype `Balance` detects out-of-range values.

```
DECLARE
  SUBTYPE Balance IS NUMBER(8,2);

  checking_account  Balance;
  savings_account   Balance;

BEGIN
  checking_account := 2000.00;
  savings_account  := 1000000.00;
END;
/
```

Result:

```
DECLARE
*
ERROR at line 1:
ORA-06502: PL/SQL: value or conversion error: number precision too large
ORA-06512: at line 9
```

Example 4-26 Implicit Conversion Between Constrained Subtypes with Same Base Type

In this example, the three constrained subtypes have the same base type. The first two subtypes can be implicitly converted to the third subtype, but not to each other.

```
DECLARE
  SUBTYPE Digit        IS PLS_INTEGER RANGE 0..9;
  SUBTYPE Double_digit IS PLS_INTEGER RANGE 10..99;
  SUBTYPE Under_100    IS PLS_INTEGER RANGE 0..99;

  d   Digit        :=  4;
  dd  Double_digit := 35;
  u   Under_100;
BEGIN
  u := d;   -- Succeeds; Under_100 range includes Digit range
  u := dd;  -- Succeeds; Under_100 range includes Double_digit range
  dd := d;  -- Raises error; Double_digit range does not include Digit range
END;
/
```

Result:

```
DECLARE
*
```

```
ERROR at line 1:
ORA-06502: PL/SQL: value or conversion error
ORA-06512: at line 12
```

Subtypes with Base Types in Same Data Type Family

If two subtypes have different base types in the same data type family, then one subtype can be implicitly converted to the other only if the source value does not violate a constraint of the target subtype.

For the predefined PL/SQL data types and subtypes, grouped by data type family, see PL/SQL Predefined Data Types.

Example 4-27 Implicit Conversion Between Subtypes with Base Types in Same Family

In this example, the subtypes `Word` and `Text` have different base types in the same data type family. The first assignment statement implicitly converts a `Word` value to `Text`. The second assignment statement implicitly converts a `Text` value to `Word`. The third assignment statement cannot implicitly convert the `Text` value to `Word`, because the value is too long.

```
DECLARE
  SUBTYPE Word IS CHAR(6);
  SUBTYPE Text IS VARCHAR2(15);

  verb       Word := 'run';
  sentence1  Text;
  sentence2  Text := 'Hurry!';
  sentence3  Text := 'See Tom run.';

BEGIN
  sentence1 := verb;  -- 3-character value, 15-character limit
  verb := sentence2;  -- 6-character value, 6-character limit
  verb := sentence3;  -- 12-character value, 6-character limit
END;
/
```

Result:

```
DECLARE
*
ERROR at line 1:
ORA-06502: PL/SQL: value or conversion error: character string buffer too small
ORA-06512: at line 13
```