B
PL/SQL Name Resolution

This appendix explains PL/SQL **name resolution**; that is, how the PL/SQL compiler resolves ambiguous references to identifiers.

An unambiguous identifier reference can become ambiguous if you change identifiers in its compilation unit (that is, if you add, rename, or delete identifiers).

Note:

The `AUTHID` property of a stored PL/SQL unit affects the name resolution of SQL statements that the unit issues at run time. For more information, see "Invoker's Rights and Definer's Rights (AUTHID Property)".

Topics

- Qualified Names and Dot Notation
- Column Name Precedence
- Differences Between PL/SQL and SQL Name Resolution Rules
- Resolution of Names in Static SQL Statements
- What is Capture?
- Avoiding Inner Capture in SELECT and DML Statements

Qualified Names and Dot Notation

When one named item belongs to another named item, you can (and sometimes must) qualify the name of the "child" item with the name of the "parent" item, using dot notation. For example:

When referencing ...	You must qualify its name with ...	Using this syntax ...
Field of a record	Name of the record	*record_name.field_name*
Method of a collection	Name of the collection	*collection_name.method*
Pseudocolumn `CURRVAL`	Name of a sequence	*sequence_name*.`CURRVAL`
Pseudocolumn `NEXTVAL`	Name of a sequence	*sequence_name*.`NEXTVAL`

If an identifier is declared in a named PL/SQL unit, you can qualify its simple name (the name in its declaration) with the name of the unit (block, subprogram, or package), using this syntax:

```
unit_name.simple_identifier_name
```

If the identifier is not visible, then you *must* qualify its name (see "Scope and Visibility of Identifiers").

If an identifier belongs to another schema, then you must qualify its name with the name of the schema, using this syntax:

```
schema_name.package_name
```

A simple name can be qualified with multiple names, as Example B-1 shows.

Some examples of possibly ambiguous qualified names are:

- Field or attribute of a function return value, for example:

  ```
  func_name().field_name
  func_name().attribute_name
  ```

- Schema object owned by another schema, for example:

  ```
  schema_name.table_name
  schema_name.procedure_name()
  schema_name.type_name.member_name()
  ```

- Package object owned by another user, for example:

  ```
  schema_name.package_name.procedure_name()
  schema_name.package_name.record_name.field_name
  ```

- Record containing an ADT, for example:

  ```
  record_name.field_name.attribute_name
  record_name.field_name.member_name()
  ```

Example B-1 Qualified Names

```
CREATE OR REPLACE PACKAGE pkg1 AUTHID DEFINER AS
  m NUMBER;
  TYPE t1 IS RECORD (a NUMBER);
  v1 t1;
  TYPE t2 IS TABLE OF t1 INDEX BY PLS_INTEGER;
  v2 t2;
  FUNCTION f1 (p1 NUMBER) RETURN t1;
  FUNCTION f2 (q1 NUMBER) RETURN t2;
END pkg1;
/

CREATE OR REPLACE PACKAGE BODY pkg1 AS
  FUNCTION f1 (p1 NUMBER) RETURN t1 IS
    n NUMBER;
  BEGIN
     n := m;             -- Unqualified variable name
     n := pkg1.m;        -- Variable name qualified by package name
     n := pkg1.f1.p1;    -- Parameter name qualified by function name,
                         --  which is qualified by package name
     n := v1.a;          -- Variable name followed by component name
     n := pkg1.v1.a;     -- Variable name qualified by package name
                         --  and followed by component name
     n := v2(10).a;      -- Indexed name followed by component name
     n := f1(10).a;      -- Function invocation followed by component name
     n := f2(10)(10).a;  -- Function invocation followed by indexed name
                         --  and followed by component name
     n := hr.pkg1.f2(10)(10).a;  -- Schema name, package name,
                                 -- function invocation, index, component name
     v1.a := p1;
     RETURN v1;
   END f1;

   FUNCTION f2 (q1 NUMBER) RETURN t2 IS
```

```
    v_t1 t1;
    v_t2 t2;
  BEGIN
    v_t1.a := q1;
    v_t2(1) := v_t1;
    RETURN v_t2;
  END f2;
END pkg1;
/
```

Column Name Precedence

If a SQL statement references a name that belongs to both a column and either a local variable or formal parameter, then the column name takes precedence.

Caution:

When a variable or parameter name is interpreted as a column name, data can be deleted, changed, or inserted unintentionally.

In Example B-2, the name `last_name` belongs to both a local variable and a column (names are not case-sensitive). Therefore, in the `WHERE` clause, both references to `last_name` resolve to the column, and all rows are deleted.

Example B-3 solves the problem in Example B-2 by giving the variable a different name.

Example B-4 solves the problem in Example B-2 by labeling the block and qualifying the variable name with the block name.

In Example B-5, the function `dept_name` has a formal parameter and a local variable whose names are those of columns of the table `DEPARTMENTS`. The parameter and variable name are qualified with the function name to distinguish them from the column names.

Example B-2 Variable Name Interpreted as Column Name Causes Unintended Result

```
DROP TABLE employees2;
CREATE TABLE employees2 AS
  SELECT LAST_NAME FROM employees;

DECLARE
  last_name  VARCHAR2(10) := 'King';
BEGIN
  DELETE FROM employees2 WHERE LAST_NAME = last_name;
  DBMS_OUTPUT.PUT_LINE('Deleted ' || SQL%ROWCOUNT || ' rows.');
END;
/
```

Result:

```
Deleted 107 rows.
```

Example B-3 Fixing Example B-2 with Different Variable Name

```
DROP TABLE employees2;
CREATE TABLE employees2 AS
  SELECT LAST_NAME FROM employees;

DECLARE
  v_last_name  VARCHAR2(10) := 'King';
BEGIN
  DELETE FROM employees2 WHERE LAST_NAME = v_last_name;
  DBMS_OUTPUT.PUT_LINE('Deleted ' || SQL%ROWCOUNT || ' rows.');
END;
/
```

Result:

```
Deleted 2 rows.
```

Example B-4 Fixing Example B-2 with Block Label

```
DROP TABLE employees2;
CREATE TABLE employees2 AS
  SELECT LAST_NAME FROM employees;

<<main>>
DECLARE
  last_name  VARCHAR2(10) := 'King';
BEGIN
  DELETE FROM employees2 WHERE last_name = main.last_name;
  DBMS_OUTPUT.PUT_LINE('Deleted ' || SQL%ROWCOUNT || ' rows.');
END;
/
```

Result:

```
Deleted 2 rows.
```

Example B-5 Subprogram Name for Name Resolution

```
DECLARE
  FUNCTION dept_name (department_id IN NUMBER)
    RETURN departments.department_name%TYPE
  IS
    department_name  departments.department_name%TYPE;
  BEGIN
    SELECT department_name INTO dept_name.department_name
      --    ^column                ^local variable
    FROM departments
    WHERE department_id = dept_name.department_id;
    --     ^column           ^formal parameter
    RETURN department_name;
  END dept_name;
BEGIN
```

```
  FOR item IN (
    SELECT department_id
    FROM departments
    ORDER BY department_name) LOOP

      DBMS_OUTPUT.PUT_LINE ('Department: ' || dept_name(item.department_id));
  END LOOP;
END;
/
```

Result:

```
Department: Accounting
Department: Administration
Department: Benefits
Department: Construction
Department: Contracting
Department: Control And Credit
Department: Corporate Tax
Department: Executive
Department: Finance
Department: Government Sales
Department: Human Resources
Department: IT
Department: IT Helpdesk
Department: IT Support
Department: Manufacturing
Department: Marketing
Department: NOC
Department: Operations
Department: Payroll
Department: Public Relations
Department: Purchasing
Department: Recruiting
Department: Retail Sales
Department: Sales
Department: Shareholder Services
Department: Shipping
Department: Treasury
```

Differences Between PL/SQL and SQL Name Resolution Rules

PL/SQL and SQL name resolution rules are very similar. However:

- PL/SQL rules are less permissive than SQL rules.

  Because most SQL rules are context-sensitive, they recognize as legal more situations than PL/SQL rules do.

- PL/SQL and SQL resolve qualified names differently.

  For example, when resolving the table name `HR.JOBS`:

  – PL/SQL searches first for packages, types, tables, and views named `HR` in the current schema, then for public synonyms, and finally for objects named `JOBS` in the `HR` schema.

  - SQL searches first for objects named `JOBS` in the `HR` schema, and then for packages, types, tables, and views named `HR` in the current schema.

To avoid problems caused by the few differences between PL/SQL and SQL name resolution rules, follow the recommendations in "Avoiding Inner Capture in SELECT and DML Statements".

> **Note:**
>
> When the PL/SQL compiler processes a static SQL statement, it sends that statement to the SQL subsystem, which uses SQL rules to resolve names in the statement. For details, see "Resolution of Names in Static SQL Statements".

Resolution of Names in Static SQL Statements

Static SQL is described in PL/SQL Static SQL.

When the PL/SQL compiler finds a static SQL statement:

1. If the statement is a `SELECT` statement, the PL/SQL compiler removes the `INTO` clause.
2. The PL/SQL compiler sends the statement to the SQL subsystem.
3. The SQL subsystem checks the syntax of the statement.

   If the syntax is incorrect, the compilation of the PL/SQL unit fails. If the syntax is correct, the SQL subsystem determines the names of the tables and tries to resolve the other names in the scope of the SQL statement.
4. If the SQL subsystem cannot resolve a name in the scope of the SQL statement, then it sends the name back to the PL/SQL compiler. The name is called an **escaped identifier**.
5. The PL/SQL compiler tries to resolve the escaped identifier.

   First, the compiler tries to resolve the identifier in the scope of the PL/SQL unit. If that fails, the compiler tries to resolve the identifier in the scope of the schema. If that fails, the compilation of the PL/SQL unit fails.
6. If the compilation of the PL/SQL unit succeeds, the PL/SQL compiler generates the text of the regular SQL statement that is equivalent to the static SQL statement and stores that text with the generated computer code.
7. At run time, the PL/SQL runtime system invokes routines that parse, bind, and run the regular SQL statement.

   The bind variables are the escaped identifiers (see step 4).
8. If the statement is a `SELECT` statement, the PL/SQL runtime system stores the results in the PL/SQL targets specified in the `INTO` clause that the PL/SQL compiler removed in step 1.

> **Note:**
>
> Bind variables can be evaluated in any order. If a program determines order of evaluation, then at the point where the program does so, its behavior is undefined.

What is Capture?

When a declaration or definition prevents the compiler from correctly resolving a reference in another scope, the declaration or definition is said to **capture** the reference. Capture is usually the result of migration or schema evolution.

Topics

- Outer Capture
- Same-Scope Capture
- Inner Capture

Note:

Same-scope and inner capture occur only in SQL scope.

Outer Capture

Outer capture occurs when a name in an inner scope, which had resolved to an item in an inner scope, now resolves to an item in an outer scope. Both PL/SQL and SQL are designed to prevent outer capture; you need not be careful to avoid it.

Same-Scope Capture

Same-scope capture occurs when a column is added to one of two tables used in a join, and the new column has the same name as a column in the other table. When only one table had a column with that name, the name could appear in the join unqualified. Now, to avoid same-scope capture, you must qualify the column name with the appropriate table name, everywhere that the column name appears in the join.

Inner Capture

Inner capture occurs when a name in an inner scope, which had resolved to an item in an outer scope, now either resolves to an item in an inner scope or cannot be resolved. In the first case, the result might change. In the second case, an error occurs.

In Example B-6, a new column captures a reference to an old column with the same name. Before new column `col2` is added to table `tab2`, `col2` resolves to `tab1.col2`; afterward, it resolves to `tab2.col2`.

To avoid inner capture, follow the rules in "Avoiding Inner Capture in SELECT and DML Statements".

Example B-6 Inner Capture of Column Reference

Table `tab1` has a column named `col2`, but table `tab2` does not:

```
DROP TABLE tab1;
CREATE TABLE tab1 (col1 NUMBER, col2 NUMBER);
INSERT INTO tab1 (col1, col2) VALUES (100, 10);

DROP TABLE tab2;
```

```
CREATE TABLE tab2 (col1 NUMBER);
INSERT INTO tab2 (col1) VALUES (100);
```

Therefore, in the inner `SELECT` statement, the reference to `col2` resolves to column `tab1.col2`:

```
CREATE OR REPLACE PROCEDURE proc AUTHID DEFINER AS
  CURSOR c1 IS
    SELECT * FROM tab1
    WHERE EXISTS (SELECT * FROM tab2 WHERE col2 = 10);
BEGIN
  OPEN c1;
  CLOSE c1;
END;
/
```

Add a column named `col2` to table `tab2`:

```
ALTER TABLE tab2 ADD (col2 NUMBER);
```

Now procedure `proc` is invalid. At its next invocation, the database automatically recompiles it, and the reference to `col2` in the inner `SELECT` statement resolves to column `tab2.col2`.

Avoiding Inner Capture in SELECT and DML Statements

Avoid inner capture of references in `SELECT`, `SELECT INTO`, and DML statements by following these recommendations:

- Specify a unique alias for each table in the statement.
- Do not specify a table alias that is the name of a schema that owns an item referenced in the statement.
- Qualify each column reference in the statement with the appropriate table alias.

In Example B-7, schema `hr` owns tables `tab1` and `tab2`. Table `tab1` has a column named `tab2`, whose Abstract Data Type (ADT) has attribute `a`. Table `tab2` does not have a column named `a`. Against recommendation, the query specifies alias `hr` for table `tab1` and references table `tab2`. Therefore, in the query, the reference `hr.tab2.a` resolves to table `tab1`, column `tab2`, attribute `a`. Then the example adds column `a` to table `tab2`. Now the reference `hr.tab2.a` in the query resolves to schema `hr`, table `tab2`, column `a`. Column `a` of table `tab2` captures the reference to attribute `a` in column `tab2` of table `tab1`.

Topics

- Qualifying References to Attributes and Methods
- Qualifying References to Row Expressions

Example B-7 Inner Capture of Attribute Reference

```
CREATE OR REPLACE TYPE type1 AS OBJECT (a NUMBER);
/
DROP TABLE tab1;
CREATE TABLE tab1 (tab2 type1);
INSERT INTO tab1 (tab2) VALUES (type1(10));

DROP TABLE tab2;
CREATE TABLE tab2 (x NUMBER);
INSERT INTO tab2 (x) VALUES (10);

/* Alias tab1 with same name as schema name,
   a bad practice used here for illustration purpose.
   Note lack of alias in second SELECT statement. */

SELECT * FROM tab1 hr
WHERE EXISTS (SELECT * FROM hr.tab2 WHERE x = hr.tab2.a);
```

Result:

```
TAB2(A)
---------------

TYPE1(10)

1 row selected.
```

Add a column named `a` to table `tab2` (which belongs to schema `hr`):

```
ALTER TABLE tab2 ADD (a NUMBER);
```

Now, when the query runs, `hr.tab2.a` resolves to schema `hr`, table `tab2`, column `a`. To avoid this inner capture, apply the recommendations to the query:

```
SELECT * FROM hr.tab1 p1
WHERE EXISTS (SELECT * FROM hr.tab2 p2 WHERE p2.x = p1.tab2.a);
```

Qualifying References to Attributes and Methods

To reference an attribute or method of a table element, you must give the table an alias and use the alias to qualify the reference to the attribute or method.

In Example B-8, table `tbl1` has column `col1` of data type `t1`, an ADT with attribute `x`. The example shows several correct and incorrect references to `tbl1.col1.x`.

Example B-8 Qualifying ADT Attribute References

```
CREATE OR REPLACE TYPE t1 AS OBJECT (x NUMBER);
/
DROP TABLE tb1;
CREATE TABLE tb1 (col1 t1);
```

The references in the following `INSERT` statements do not need aliases, because they have no column lists:

```
BEGIN
  INSERT INTO tb1 VALUES ( t1(10) );
  INSERT INTO tb1 VALUES ( t1(20) );
  INSERT INTO tb1 VALUES ( t1(30) );
END;
/
```

The following references to the attribute `x` cause error ORA-00904:

```
UPDATE tb1 SET col1.x = 10 WHERE col1.x = 20;

UPDATE tb1 SET tb1.col1.x = 10 WHERE tb1.col1.x = 20;

UPDATE hr.tb1 SET hr.tb1.col1.x = 10 WHERE hr.tb1.col1.x = 20;

DELETE FROM tb1 WHERE tb1.col1.x = 10;
```

The following references to the attribute `x`, with table aliases, are correct:

```
UPDATE hr.tbl1 t SET t.col1.x = 10 WHERE t.col1.x = 20;

DECLARE
  y NUMBER;
BEGIN
  SELECT t.col1.x INTO y FROM tb1 t WHERE t.col1.x = 30;
END;
/

DELETE FROM tb1 t WHERE t.col1.x = 10;
```

Qualifying References to Row Expressions

Row expressions must resolve as references to table aliases. A row expression can appear in the `SET` clause of an `UPDATE` statement or be the parameter of the SQL function `REF` or `VALUE`.

In Example B-9, table `ot1` is a standalone nested table of elements of data type `t1`, an ADT with attribute `x`. The example shows several correct and incorrect references to row expressions.

Example B-9 Qualifying References to Row Expressions

```
CREATE OR REPLACE TYPE t1 AS OBJECT (x number);
/
DROP TABLE ot1;
CREATE TABLE ot1 OF t1;

BEGIN
  INSERT INTO ot1 VALUES (t1(10));
  INSERT INTO ot1 VALUES (20);
  INSERT INTO ot1 VALUES (30);
END;
/
```

The following references cause error ORA-00904:

```
UPDATE ot1 SET VALUE(ot1.x) = t1(20) WHERE VALUE(ot1.x) = t1(10);

DELETE FROM ot1 WHERE VALUE(ot1) = (t1(10));
```

The following references, with table aliases, are correct:

```
UPDATE ot1 o SET o = (t1(20)) WHERE o.x = 10;

DECLARE
  n_ref  REF t1;
BEGIN
  SELECT REF(o) INTO n_ref FROM ot1 o WHERE VALUE(o) = t1(30);
END;
/

DECLARE
  n t1;
BEGIN
  SELECT VALUE(o) INTO n FROM ot1 o WHERE VALUE(o) = t1(30);
END;
/

DECLARE
  n NUMBER;
BEGIN
```

```
  SELECT o.x INTO n FROM ot1 o WHERE o.x = 30;
END;
/

DELETE FROM ot1 o WHERE VALUE(o) = (t1(20));
```