Oracle® Warehouse Builder

Transformation Guide

10*g* Release 2 (10.2)

B28227-01

June 2006

Oracle Warehouse Builder Transformation Guide, 10*g* Release 2 (10.2)

B28227-01

Contents

Preface

This preface includes the following topics:

- Audience
- Documentation Accessibility
- Conventions
- Related Publications

Audience

This manual is written for Oracle database administrators and others who create warehouses using Oracle Warehouse Builder.

Documentation Accessibility

Our goal is to make Oracle products, services, and supporting documentation accessible, with good usability, to the disabled community. To that end, our documentation includes features that make information available to users of assistive technology. This documentation is available in HTML format, and contains markup to facilitate access by the disabled community. Accessibility standards will continue to evolve over time, and Oracle is actively engaged with other market-leading technology vendors to address technical obstacles so that our documentation can be accessible to all of our customers. For more information, visit the Oracle Accessibility Program Web site at

`http://www.oracle.com/accessibility/`

Accessibility of Code Examples in Documentation

Screen readers may not always correctly read the code examples in this document. The conventions for writing code require that closing braces should appear on an otherwise empty line; however, some screen readers may not always read a line of text that consists solely of a bracket or brace.

Accessibility of Links to External Web Sites in Documentation

This documentation may contain links to Web sites of other companies or organizations that Oracle does not own or control. Oracle neither evaluates nor makes any representations regarding the accessibility of these Web sites.

TTY Access to Oracle Support Services

Oracle provides dedicated Text Telephone (TTY) access to Oracle Support Services within the United States of America 24 hours a day, seven days a week. For TTY support, call 800.446.2398.

Conventions

In this manual, Windows refers to the Windows NT, Windows 2000, and Windows XP operating systems. The SQL*Plus interface to Oracle Database may be referred to as SQL.

In the examples, an implied carriage return occurs at the end of each line, unless otherwise noted. You must press the Return key at the end of a line of input.

The following table lists the conventions used in this manual:

Convention	Meaning
. . .	Vertical ellipsis points in an example mean that information not directly related to the example has been omitted.
...	Horizontal ellipsis points in statements or commands mean that parts of the statement or command not directly related to the example have been omitted.
boldface text	Boldface type in text refers to interface buttons and links. Boldface type also serves as emphasis to set apart main ideas.
italicized text	Italicized text applies to new terms introduced for the first time. Italicized text also serves as an emphasis on key concepts.
`unicode text`	Unicode text denotes exact code, file directories and names, and literal commands.
`italicized unicode text`	Italicized unicode text refers to parameters whose value is specified by the user.
[]	Brackets enclose optional clauses from which you can choose one or none.

Related Publications

The Warehouse Builder documentation set includes these manuals:

- *Oracle Warehouse Builder User's Guide*
- *Oracle Warehouse Builder Transformation Guide*
- *Oracle Warehouse Builder Installation and Configuration Guide*
- *Oracle Warehouse Builder API and Scripting Reference*
- *Oracle Business Intelligence Concepts Guide*

In addition to the Warehouse Builder documentation, you can reference *Oracle Database Data Warehousing Guide*.

1

Introducing Oracle Warehouse Builder Transformations

One of the main functions of an Extract, Transformation, and Loading (ETL) tool is to transform data. Oracle Warehouse Builder provides several methods of transforming data. This chapter discusses transformations and describes how to create custom transformation using Warehouse Builder. It also describes how to import transformation definitions.

This chapter contains the following topics:

About Transformations

Transformations are PL/SQL functions, procedures, and packages that enable you to transform data. You use transformations when designing mappings and process flows that define ETL processes.

Transformations are stored in the Warehouse Builder repository. Depending on where the transformation is defined, transformations can be shared either across the project in which they are defined or across all the projects.

Transformation packages are deployed at the package level but executed at the transformation level.

Types of Transformations

Transformations, in Warehouse Builder, can be categorized as follows:

- Predefined Transformations
- Custom Transformations

The following sections provide more details about these types of transformations.

Predefined Transformations

Warehouse Builder enables you to perform common transformations by providing a set of predefined transformations. These predefined transformations are part of the

Oracle Library and consist of built-in and seeded functions and procedures. You can directly use these predefined transformations to transform your data. For more information on the Oracle Library, see "Types of Transformation Libraries" on page 1-4.

Predefined transformations are organized into the following categories:

- Administration
- Bioinformatics
- Character
- Control Center
- Conversion
- Date
- Numeric
- OLAP
- Other
- Streams
- XML

For more information about the transformations that belong to each category, see Chapter 2, "Transformations".

Custom Transformations

A custom transformation is one that is created by the user. Custom transformations can use predefined transformations as part of their definition.

Custom transformations contains the following categories:

- **Functions:** The functions category contains any standalone functions. It is available under the Public Transformations node of the Global Explorer. It is also created automatically under the Transformations node of every Oracle module in the Project Explorer. Functions can be defined by the user or imported from a database. A function transformation takes 0-n input parameters and produces a result value.
- **Procedures:** The procedures category contains any standalone procedures used as transformations. It is available under the Public Transformations node of the Global Explorer. The Procedures category is also automatically created under the Transformations node of each Oracle module in the Global Explorer. Procedures can be defined by the user or imported from a database. A procedure transformation takes 0-n input parameters and produces 0-n output parameters.
- **Packages:** PL/SQL packages can be created or imported in Warehouse Builder. The package body may be modified. The package header, which is the signature for the function or procedure, cannot be modified. The package can be viewed in the transformation library property sheet.
- **PL/SQL Types:** PL/SQL types include PL/SQL record types, ref cursor types, and nested table types. The PL/SQL types category contains any standalone PL/SQL types. It is automatically created under the Package node of every transformation, both in the Project Explorer and Global Explorer.

In addition to these categories, you can also import PL/SQL packages. Although you can modify the package body of an imported package, you cannot modify the package header, which is the signature for the function or procedure. For more information on importing PL/SQL packages, see "Importing PL/SQL" on page 1-12.

For more information about creating custom transformations, see "Defining Custom Transformations" on page 1-5.

Transforming Data Using Warehouse Builder

Warehouse Builder provides an intuitive user interface that enables you to define transformations. You can either use the predefined transformations or define custom transformations that suit your requirements. Transformations are stored in the repository. Custom transformation can be deployed to the Oracle database just like any other data object that you define in an Oracle module.

The Mapping Editor includes a set of prebuilt transformation operators that enable you to define common transformations when you define how data will move from source to target. Transformation operators are prebuilt PL/SQL functions, procedures, package functions, and package procedures. They take input data, perform operations on it and produce output. For more information on these operators, refer to the chapter titled "Data Flow Operators" in the Oracle Warehouse Builder User's Guide.

Benefits of Using Warehouse Builder for Transforming Data

Warehouse Builder enables you to reuse PL/SQL as well as to write your own PL/SQL transformations. To enable faster development of warehousing solutions, Warehouse Builder provides custom procedures and functions written in PL/SQL.

Because SQL and PL/SQL are versatile and proven languages widely used by many information professionals, the time and expense of developing an alternative transformation language is eliminated by using Warehouse Builder. With Warehouse Builder, you can create solutions using existing knowledge and a proven, open, and standard technology.

All major relational database management systems support SQL and all programs written in SQL can be moved from one database to another with very little modification.

This means that all the SQL knowledge in your organization is fully portable to Warehouse Builder. Warehouse Builder enables you to import and maintain any existing complex custom code. You can later use these custom transformations in Warehouse Builder mappings.

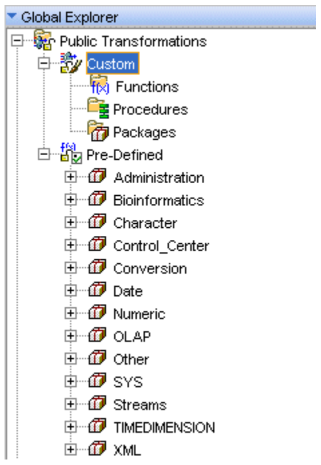
About Transformation Libraries

A transformation library consists of a set of reusable transformations. Each time you create a repository, Warehouse Builder creates a Transformation Library containing transformation operations for that project. This library contains the standard Oracle Library and an additional library for each Oracle module defined within the project.

Types of Transformation Libraries

Transformation libraries are available under the Public Transformations node of the Global Explorer in the Design Center. Figure 1–1 displays the Global Explorer with the Public Transformations node expanded.

Figure 1–1 Global Explorer with the Oracle Library



Transformation libraries can be categorized as follows:

- **Oracle Library**

  This is a collection of predefined functions from which you can define procedures for your Global Shared Library. The Oracle Library is contained in the Global Explorer. Expand the Pre-Defined node under the Public Transformation node. Each category of predefined transformations is represented by a separate node as shown in Figure 1–1. Expand the node for a category to view the predefined transformations in that category. For example, expand the Character node to view the predefined character transformations contained in the Oracle library.

- **Global Shared Library**

  This is a collection of reusable transformations created by the user. These transformations are categorized as functions, procedures, and packages defined within your repository.

  The transformations in the Global Shared Library are available under the Custom node of the Public Transformations node as shown in Figure 1–1. Any transformation that you create under this node is available across all projects in the repository. For information on creating transformations in the global shared library, see "Defining Custom Transformations" on page 1-5.

  When you deploy a transformation defined in the Global Shared Library, the transformation is deployed to the location that is associated with the default control center.

Accessing Transformation Libraries

Since transformations can be used at different points in the ETL process, Warehouse Builder enables you to access transformation libraries from different points in the Design Center.

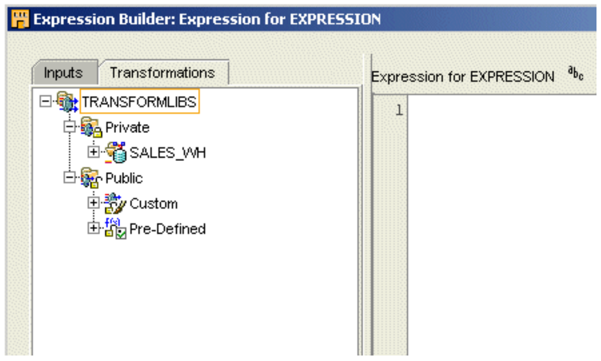
You can access the Transformation Libraries using the following:

- Expression Builder

  While creating mappings, you may need to create expressions to transform your source data. The Expression Builder interface enables you to create the expressions required to transform data. Since these expressions can include transformations, Warehouse Builder enables you to access transformation libraries from the Expression Builder.

Transformation libraries are available under the Transformations tab of the Expression Builder as shown in Figure 1–2. The Private node under TRANSFORMLIBS contains transformations that are available only in the current project. These transformations are created under the Transformation node of the Oracle module. The Public node contains the custom transformations from the Global shared Library and the predefined transformations from the Oracle Library.

Figure 1–2 Transformation Libraries in the Expression Builder



- Add Transformation Operator dialog

  The Transformation operator in the Mapping Editor enables you to add transformations, both from the Oracle library and the Global Shared Library, to a mapping. You can use this operator to transform data as part of the mapping.

- Create Transformation Wizard

  The Implementation page of the Create Transformation Wizard enables you to specify the PL/SQL code that is part of the function or procedure body. You can use transformations in the PL/SQL code.

Defining Custom Transformations

Custom transformations include procedures, functions, packages, and table functions. Warehouse Builder provides wizards to create each type of custom transformation. Custom transformations can belong to the Global Shared Library or to a particular project.

Custom Transformations in the Global Shared Library

Custom transformations that are part of the Global Shared Library can be used across all projects of the repository in which they are defined. For example, you create a function called ADD_EMPL in the Global Shared Library of the repository REP_OWNER. This procedure can be used across all the projects in REP_OWNER.

You use the Custom node of the Public Transformations node in the Global Explorer to define custom transformations that can used across all projects in the repository. Figure 1–1 displays the Global Explorer that you use to create such transformations.

To create a custom transformation in the Global Shared Library:

1. From the Global Explorer, expand the Public Transformations node and then the Custom node.

   Warehouse Builder displays the type of transformations that you can create. This includes functions, procedures, and packages. Note that PL/SQL types can be created only as part of a package.

2. Right-click the type of transformation you want to define and select **New**.

   For example, to create a function, right-click **Functions** and select **New**.

3. For functions and procedures, Warehouse Builder displays the Welcome page of the Create Function Wizard or the Create Procedure wizard respectively. Click **Next** to proceed. See "Defining Functions and Procedures" on page 1-7 for more information about the other pages in the wizard.

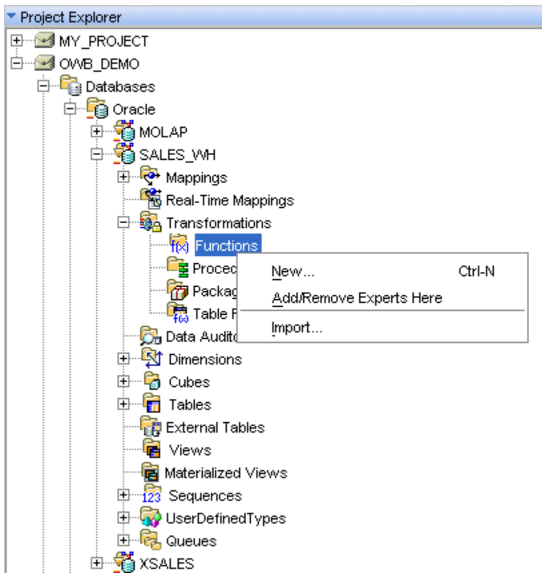
   For packages, Warehouse Builder displays the Create Transformation Library dialog. Provide a name and an optional description for the package and click **OK**. The new package is added to the Packages node. You can subsequently create procedures, functions, or PL/SQL types that belong to this package. For more information about creating PL/SQL types, see "Defining PL/SQL Types" on page 1-8.

Custom Transformations in a Project

Sometimes, you may need to define custom transformations that are required only in the current module or project. In this case, you can define custom transformations in an Oracle module of a project. Such custom transformations are accessible from all the projects in the current repository. For example, consider the repository owner called REP_OWNER that contains two projects PROJECT1 and PROJECT2. In the Oracle module called SALES of PROJECT1, you define a procedure called CALC_SAL. This procedure can be used in all modules belonging to PROJECT1, but is not accessible in PROJECT2.

Figure 1–3 displays the Project Explorer from which you can create custom transformations that are accessible within the project in which they are defined. Expand the Oracle module of the project in which you want to create a custom transformation. Expand the Transformations node under the module. There is a node for each type of custom transformation. Use these nodes to create your transformation.

Figure 1–3 Creating Custom Transformations in an Oracle Module



To define a custom transformation in an Oracle Module:

1. From the Project Explorer, expand the Oracle warehouse module node and then the Transformations node.

2. Right-click the type of transformation you want to create and select **New**. For example, to create a package, right-click **Packages** and select **New**.

   For functions or procedures, Warehouse Builder displays the Welcome page of the Create Function Wizard or the Create Procedure Wizard respectively. Click **Next** to proceed. See "Defining Functions and Procedures" on page 1-7 for information about the remaining wizard pages.

   For packages, Warehouse Builder opens the Create Transformation Library dialog. Provide a name and an optional description for the package and click **OK**. The package gets added under the Packages node. You can subsequently create procedures, functions, or PL/SQL types that belong to this package. For more information about creating PL/SQL types, see "Defining PL/SQL Types" on page 1-8.

Defining Functions and Procedures

Use the following pages of the Create Function Wizard or Create Procedure Wizard to define a function or procedure.

- Name and Description Page on page 1-7
- Parameters Page on page 1-7
- Implementation Page on page 1-8
- Summary Page on page 1-8

Name and Description Page

You use the Name and Description page to describe the custom transformation. Specify the following details on this page:

- **Name:** Represents the name of the custom transformation. For more information about naming conventions, refer to the section titled "Naming Conventions for Data Objects" in the chapter "Defining Oracle Data Objects" of the Oracle Warehouse Builder User's Guide.
- **Description:** Represents the description of the custom transformation. This is an optional field.
- **Return Type:** Represents the data type of the value returned by the function. You select a return type from the available options in the drop-down list. This field is applicable only for functions.

Parameters Page

Use the Parameters page to define the parameters, both input and output, of the transformation. Specify the following details for each parameter:

- **Name:** Enter the name of the parameter.
- **Type:** Select the data type of the parameter from the drop-down list.
- **I/O:** Select the type of parameter. The options available are Input, Output, and input/Output.
- **Required:** Select Yes to indicate that a parameter is mandatory and No to indicate that it is not mandatory.
- **Default Value:** Enter the default value for the parameter. The default value is used when you do not specify a value for the parameter at the time of executing the function or procedure.

Implementation Page

Use the Implementation page to specify the implementation details, such as the code, of the transformation. To specify the code used to implement the function or procedure, click **Code Editor**. Warehouse Builder displays the Code Editor window of the New Transformation wizard. This editor contains two panels. The upper panel displays the code and the lower panel displays the function signature and messages.

When you create a function, the following additional options are displayed:

- **Function is deterministic:** This hint helps to avoid redundant function calls. If a stored function was called previously with the same arguments, the previous result can be used. The function result should not depend on the state of session variables or schema objects. Otherwise, results might vary across calls. Only DETERMINISTIC functions can be called from a function-based index or a materialized view that has query-rewrite enabled.
- **Enable function for parallel execution:** This option declares that a stored function can be used safely in the child sessions of parallel DML evaluations. The state of a main (logon) session is never shared with child sessions. Each child session has its own state, which is initialized when the session begins. The function result should not depend on the state of session (static) variables. Otherwise, results might vary across sessions.
- **Pipelined:** Select this option to create a pipelined table function. This option is enabled only when you create a table function.

Summary Page

The Summary page provides a summary of the options that you chose on the previous pages of the wizard. Click **Finish** to complete defining the function or procedure. Warehouse Builder creates the function or procedure and displays it under the corresponding folder under the Public Transformations and Custom nodes in the Global Explorer.

Defining PL/SQL Types

Warehouse Builder enables you to create the following PL/SQL types:

- PL/SQL Record types
- Ref Cursor types
- Nested Table types

To create a PL/SQL Type, use the following steps:

1. From the Project Explorer, expand the Transformations node.

   To create a PL/SQL type in the Global Shared Library, from the Global Explorer, expand the Public Transformations node and then the Custom node.
2. Expand the package node under which you want to create the PL/SQL type.
3. Right-click **PL/SQL Types** and select **New**.

   The Welcome page of the Create PL/SQL Type Wizard is displayed. Click **Next** to proceed. The wizard guides you through the following pages:

   - Name and Description Page
   - Attributes Page
   - Return Type Page

- Summary Page

Name and Description Page

Use the Name and Description page to provide the name and an optional description for the PL/SQL type. Also use this page to select the type of PL/SQL type you want to create.

You can create any of the following PL/SQL types:

- PL/SQL Record Type

  Select this option to create a PL/SQL record type. A record type is a composite data structure whose attributes can have different data types. You can use a record type to hold related items and pass them to subprograms as a single parameter. For example, you can create an employee record whose attributes include employee ID, first name, last name, date of joining, and department ID.

- Ref Cursor Type

  Select this option to create a ref cursor. Ref cursors are like pointers to result sets. The advantage with a ref cursor is that it is not tied to any particular query.

- Nested Table Type

  Select this option to create a nested table. Nested tables represent sets of values. They are similar to one-dimensional arrays with no declared number of elements. Nested tables enable you to model multidimensional arrays by creating a nested table whose elements are also tables.

After specifying the name and selecting the type of PL/SQL type object to create, click **Next**.

Attributes Page

Use the Attributes page to define the attributes of the PL/SQL record type. You specify attributes only for PL/SQL record types. A PL/SQL record must have at least one attribute.

For each attribute, define the following:

- **Name:** The name of the attribute. The name should be unique within the record type.
- **Type:** The data type of the attribute. Select the data type from the drop-down list.
- **Length:** The length of the data type, for character data types.
- **Precision:** The total number of digits allowed for the attribute, for numeric data types.
- **Scale:** The total number of digits to the right of the decimal point, for numeric data types.
- **Seconds Precision:** The number of digits in the fractional part of the datetime field. It can be a number between 0 and 9. The Seconds Precision is used only for TIMESTAMP data types.

Click **Next** to proceed to the next step.

Return Type Page

Use the Return Type page to select the return type of the PL/SQL type. You must specify a return type when you create ref cursors and nested tables.

To define ref cursors:

The return type for a ref cursor can only be a PL/SQL record type. If you know the name of the PL/SQL record type, you can search for it by typing the name in the **Search For** field and clicking **Go**.

The area below the Search For field displays the available PL/SQL types. These PL/SQL types are grouped under the two nodes: Public and Private. Expand the Public node to view the PL/SQL types that are part of the Oracle shared library. The types are grouped by package name. The Private node contains PL/SQL types that are created as part of a package in an Oracle module. Only PL/SQL types that belong to the current project are displayed. Each Oracle module is represented by a node. Within the module, the PL/SQL types are grouped by the package to which they belong.

To define nested tables:

For nested tables, the return type can be a scalar data type or a PL/SQL record type. Select one of the following options on this page based on what the PL/SQL type returns:

- Select a scalar type as return type

  This option enables you to create a PL/SQL type that returns a scalar type. Use the drop-down list to select the data type.

- Select a PL/SQL record as return type

  This option enables you to create a PL/SQL type that returns a PL/SQL record type. If you know the name of the PL/SQL record type that is returned, type the name in the **Search For** field and click **Go**. The results of the search are displayed in the area below the option.

  You can also select the return type from the list of available types displayed. The area below this option contains two nodes: Public and Private. The Public node contains PL/SQL record types that are part of the Oracle Shared Library. The PL/SQL record types are grouped by the package to which they belong. The Private node contains the PL/SQL record types created as transformations in each Oracle module in the current project. These are grouped by module. Select the PL/SQL record type that the PL/SQL type returns.

Click **Next** to proceed with the creation of the PL/SQL type.

Summary Page

The Summary page displays the options that you have chosen on the wizard pages. Review the options. Click **Back** to modify any options. Click **Finish** to create the PL/SQL type.

Editing Transformation Properties

You can edit the definition of a transformation using the editors. Make sure you edit properties consistently. For example, if you change the name of a parameter, then you must also change its name in the implementation code.

Editing Function or Procedure Definitions

The Edit Function dialog enables you to edit function definitions. To edit a procedure definition, use the Edit Procedure dialog.

Use the following steps to edit functions or procedures:

1. From the Project Explorer, expand the Oracle module in which the transformation is created. Then expand the Transformations node.

   To edit a transformation that is part of the Global Shared Library, from the Global Explorer, expand the Public Transformations node, and then the Custom node.

2. Right-click the name of the function, procedure, or package you want to edit and select **Open Editor.**

   The Edit Function or Edit Procedure dialog is displayed. Use the following tabs to edit the function or procedure definition:

   - Name Tab on page 1-11
   - Parameters Tab on page 1-11
   - Implementation Tab on page 1-11

To edit a package, Warehouse Builder displays the Edit Transformation Library dialog. You can only edit the name and description of the package. You can edit the functions and procedures contained within the package using the steps used to edit functions or packages.

Name Tab

Use the Name tab to edit the name and description of the function or procedure. For functions, you can also edit the return data type.

Parameters Tab

Use the Parameters tab to edit, add, or delete new parameters for a function or procedure. You can also edit and define the attributes of the parameters. The contents of the Parameters tab are the same as that of the Parameters page of the Create Transformation Wizard. For more information about the contents of this page, see "Parameters Page" on page 1-7.

Implementation Tab

Use the Implementation tab to review the PL/SQL code for the function or procedure. Click **Code Editor** to edit the code. The contents of the Implementation tab are the same as that of the Implementation page of the Create Transformation Wizard. For more information on the contents of the Implementation page, see "Implementation Page" on page 1-8.

Editing PL/SQL Types

The Edit PL/SQL Type dialog enables you to edit the definition of a PL/SQL type. Use the following steps to edit a PL/SQL type:

1. From the Project Explorer, expand the Oracle module that contains the PL/SQL type. Then expand the Transformations node.

   To edit a PL/SQL type stored in the Global Shared Library, expand the Public Transformations node in the Global Explorer, and then the Custom node.

2. Expand the package that contains the PL/SQL type and then the PL/SQL Types node.

3. Right-click the name of the PL/SQL type that you want to edit and select **Open Editor**.

   The Edit PL/SLQ Type dialog is displayed. Use the following tabs to edit the PL/SQL type:

- Name Tab
- Attributes Tab
- Return Type Tab

Name Tab

The Name tab displays the name and the description of the PL/SQL type. Use this tab to edit the name or the description of the PL/SQL type.

To rename a PL/SQL type, select the name and enter the new name.

Attributes Tab

The Attributes tab displays details about the existing attributes of the PL/SQL record type. This tab is displayed for PL/SQL record types only. You can modify existing attributes, add new attributes, or delete attributes.

To add a new attribute, click the **Name** column of a blank row specify the details for the attribute. To delete an attribute, right-click the gray cell to the left the row that represents the attribute and select **Delete**.

Return Type Tab

Use the Return Type tab to modify the details of the return type of the PL/SQL type. For a ref cursor, the return type must be a PL/SQL record. For a nested table, the return type can be a PL/SQL record type or a scalar data type.

Importing PL/SQL

Use the Import Wizard to import PL/SQL functions, procedures, and packages into a Warehouse Builder project.

The following steps describe how to import PL/SQL packages from other sources into Warehouse Builder.

To import a PL/SQL function, procedure, or package:

1. From the Project Explorer, expand the project node and then Databases node.
2. Right-click an Oracle module node and select **Import**.

   Warehouse Builder displays the Import Metadata Wizard Welcome page.
3. Click **Next.**
4. Select **PL/SQL Transformation** in the Object Type field of the Filter Information page.
5. Click **Next.**

   The Import Metadata Wizard displays the Object Selection page.
6. Select a function, procedure, or package from the Available Objects list. Move the objects to the Selected Objects list by clicking the single arrow button to move a single object or the double arrow button to move multiple objects.
7. Click **Next.**

   The Import Metadata Wizard displays the Summary and Import page.
8. Verify the import information. Click **Back** to revise your selections.
9. Click **Finish** to import the selected PL/SQL transformations.

Warehouse Builder displays the Import Results page.

10. Click **OK** proceed with the import. Click **Undo** to cancel the import process.

    The imported PL/SQL information appears under the Transformations node of the Oracle node into which you imported the data.

When you use imported PL/SQL:

- You can edit, save, and deploy the imported PL/SQL functions and procedures.
- You cannot edit imported PL/SQL packages.
- Wrapped PL/SQL objects are not readable.
- Imported packages can be viewed and modified in the category property sheet.
- You can edit the imported package body but not the imported package specification.

2 Transformations

As you design mappings and process flows, you may want to use specialized transformations to transform data. This chapter describes all the predefined transformations provided by Warehouse Builder.

This chapter contains the following topics, each of which details all the predefined transformations in that category.

Administrative Transformations

Administrative transformations provide pre-built functionality to perform actions that are regularly performed in ETL processes. The main focus of these transformations is in the DBA related areas or to improve performance. For example, it is common to disable constraints when loading tables and then to re-enable them after loading has completed.

The administrative transformations in Warehouse Builder are custom functions. The Administrative transformation that Warehouse Builder provides are:

- WB_ABORT on page 2-2
- WB_COMPILE_PLSQL on page 2-2
- WB_DISABLE_ALL_CONSTRAINTS on page 2-2
- WB_DISABLE_ALL_TRIGGERS on page 2-3
- WB_DISABLE_CONSTRAINT on page 2-4
- WB_DISABLE_TRIGGER on page 2-5

- WB_ENABLE_ALL_CONSTRAINTS on page 2-6
- WB_ENABLE_ALL_TRIGGERS on page 2-6
- WB_ENABLE_CONSTRAINT on page 2-7
- WB_ENABLE_TRIGGER on page 2-8
- WB_TRUNCATE_TABLE on page 2-9

WB_ABORT

Syntax

```
WB_ABORT(p_code, p_message)
```

where *p_code* is the abort code, and must be between -20000 and -29999; and *p_message* is an abort message you specify.

Purpose

`WB_ABORT` enables you to abort the application from a Warehouse Builder component. You can run it from a post mapping process or as a transformation within a mapping.

Example

Use this administration function to abort an application. You can use this function in a post mapping process to abort deployment if there is an error in the mapping.

WB_COMPILE_PLSQL

Syntax

```
WB_COMPILE_PLSQL(p_name, p_type)
```

where *p_name* is the name of the object that is to be compiled; *p_type* is the type of object to be compiled. The legal types are:

```
'PACKAGE'
'PACKAGE BODY'
'PROCEDURE'
'FUNCTION'
'TRIGGER'
```

Purpose

This program unit compiles a stored object in the database.

Example

The following hypothetical example compiles the procedure called `add_employee_proc`:

```
EXECUTE WB_COMPILE_PLSQL('ADD_EMPLOYEE_PROC', 'PROCEDURE');
```

WB_DISABLE_ALL_CONSTRAINTS

Syntax

```
WB_DISABLE_ALL_CONSTRAINTS(p_name)
```

where *p_name* is the name of the table on which constraints are disabled.

Purpose

This program unit disables all constraints that are owned by the table as stated in the call to the program.

For faster loading of data sets, you can disable constraints on a table. The data is now loaded without validation. This is mainly done on relatively clean data sets.

Example

The following example shows the disabling of the constraints on the table OE.CUSTOMERS:

```
SELECT constraint_name
,      DECODE(constraint_type,'C','Check','P','Primary') Type
,      status
FROM user_constraints
WHERE table_name = 'CUSTOMERS';

CONSTRAINT_NAME                TYPE    STATUS
------------------------------ ------- --------
CUST_FNAME_NN                  Check   ENABLED
CUST_LNAME_NN                  Check   ENABLED
CUSTOMER_CREDIT_LIMIT_MAX      Check   ENABLED
CUSTOMER_ID_MIN                Check   ENABLED
CUSTOMERS_PK                   Primary ENABLED
```

Perform the following in SQL*Plus or Warehouse Builder to disable all constraints:

```
EXECUTE WB_DISABLE_ALL_CONSTRAINTS('CUSTOMERS');

CONSTRAINT_NAME                TYPE    STATUS
------------------------------ ------- --------
CUST_FNAME_NN                  Check   DISABLED
CUST_LNAME_NN                  Check   DISABLED
CUSTOMER_CREDIT_LIMIT_MAX      Check   DISABLED
CUSTOMER_ID_MIN                Check   DISABLED
CUSTOMERS_PK                   Primary DISABLED
```

Note: This statement uses a cascade option to allow dependencies to be broken by disabling the keys.

WB_DISABLE_ALL_TRIGGERS

Syntax

```
WB_DISABLE_ALL_TRIGGERS(p_name)
```

where *p_name* is the table name on which the triggers are disabled.

Purpose

This program unit disables all triggers owned by the table as stated in the call to the program. The owner of the table must be the current user (in variable USER). This action stops triggers and improves performance.

Example

The following example shows the disabling of all triggers on the table `OE.OC_ORDERS`:

```
SELECT trigger_name
,      status
FROM user_triggers
WHERE table_name = 'OC_ORDERS';

TRIGGER_NAME                   STATUS
------------------------------ --------
ORDERS_TRG                     ENABLED
ORDERS_ITEMS_TRG               ENABLED
```

Perform the following in SQL*Plus or Warehouse Builder to disable all triggers on the table `OC_ORDERS`.

```
EXECUTE WB_DISABLE_ALL_TRIGGERS ('OC_ORDERS');

TRIGGER_NAME                   STATUS
------------------------------ --------
ORDERS_TRG                     DISABLED
ORDERS_ITEMS_TRG               DISABLED
```

WB_DISABLE_CONSTRAINT

Syntax

```
WB_DISABLE_CONSTRAINT(p_constraintname, p_tablename)
```

where *`p_constraintname`* is the constraint name to be disabled; *`p_tablename`* is the table name on which the specified constraint is defined.

Purpose

This program unit disables the specified constraint that is owned by the table as stated in the call to the program. The user is the current user (in variable `USER`).

For faster loading of data sets, you can disable constraints on a table. The data is then loaded without validation. This reduces overhead and is mainly done on relatively clean data sets.

Example

The following example shows the disabling of the specified constraint on the table `OE.CUSTOMERS`:

```
SELECT constraint_name
, DECODE(constraint_type
, 'C', 'Check'
, 'P', 'Primary'
) Type
, status
FROM user_constraints
WHERE table_name = 'CUSTOMERS';

CONSTRAINT_NAME                TYPE    STATUS
------------------------------ ------- --------
CUST_FNAME_NN                  Check   ENABLED
CUST_LNAME_NN                  Check   ENABLED
```

```
CUSTOMER_CREDIT_LIMIT_MAX      Check   ENABLED
CUSTOMER_ID_MIN                Check   ENABLED
CUSTOMERS_PK                   Primary ENABLED
```

Perform the following in SQL*Plus or Warehouse Builder to disable the specified constraint.

```
EXECUTE WB_DISABLE_CONSTRAINT('CUSTOMERS_PK','CUSTOMERS');

CONSTRAINT_NAME                TYPE    STATUS
------------------------------ ------- --------
CUST_FNAME_NN                  Check   ENABLED
CUST_LNAME_NN                  Check   ENABLED
CUSTOMER_CREDIT_LIMIT_MAX      Check   ENABLED
CUSTOMER_ID_MIN                Check   ENABLED
CUSTOMERS_PK                   Primary DISABLED
```

Note: This statement uses a cascade option to allow dependencies to be broken by disabling the keys.

WB_DISABLE_TRIGGER

Syntax

```
WB_DISABLE_TRIGGER(p_name)
```

where *p_name* is the trigger name to be disabled.

Purpose

This program unit disables the specified trigger. The owner of the trigger must be the current user (in variable USER).

Example

The following example shows the disabling of a trigger on the table OE.OC_ORDERS:

```
SELECT trigger_name, status
FROM user_triggers
WHERE table_name = 'OC_ORDERS';

TRIGGER_NAME                   STATUS
------------------------------ --------
ORDERS_TRG                     ENABLED
ORDERS_ITEMS_TRG               ENABLED
```

Perform the following in SQL*Plus or Warehouse Builder to disable the specified constraint.

```
ECECUTE WB_DISABLE_TRIGGER ('ORDERS_TRG');

TRIGGER_NAME                   STATUS
------------------------------ --------
ORDERS_TRG                     DISABLED
ORDERS_ITEMS_TRG               ENABLED
```

WB_ENABLE_ALL_CONSTRAINTS

Syntax

```
WB_ENABLE_ALL_CONSTRAINTS(p_name)
```

where *p_name* is the name of the table for which all constraints should be enabled.

Purpose

This program unit enables all constraints that are owned by the table as stated in the call to the program.

For faster loading of data sets, you can disable constraints on a table. After the data is loaded, you must enable these constraints again using this program unit.

Example

The following example shows the enabling of the constraints on the table OE.CUSTOMERS:

```
SELECT constraint_name
, DECODE(constraint_type
, 'C', 'Check'
, 'P', 'Primary)
Type
, status
FROM user_constraints
WHERE table_name = 'CUSTOMERS';

CONSTRAINT_NAME                TYPE    STATUS
------------------------------ ------- --------
CUST_FNAME_NN                  Check   DISABLED
CUST_LNAME_NN                  Check   DISABLED
CUSTOMER_CREDIT_LIMIT_MAX      Check   DISABLED
CUSTOMER_ID_MIN                Check   DISABLED
CUSTOMERS_PK                   Primary DISABLED
```

Perform the following in SQL*Plus or Warehouse Builder to enable all constraints.

```
EXECUTE WB_ENABLE_ALL_CONSTRAINTS('CUSTOMERS');

CONSTRAINT_NAME                TYPE    STATUS
------------------------------ ------- --------
CUST_FNAME_NN                  Check   ENABLED
CUST_LNAME_NN                  Check   ENABLED
CUSTOMER_CREDIT_LIMIT_MAX      Check   ENABLED
CUSTOMER_ID_MIN                Check   ENABLED
CUSTOMERS_PK                   Primary ENABLED
```

WB_ENABLE_ALL_TRIGGERS

Syntax

```
WB_ENABLE_ALL_TRIGGERS(p_name)
```

where `p_name` is the table name on which the triggers are enabled.

Purpose

This program unit enables all triggers owned by the table as stated in the call to the program. The owner of the table must be the current user (in variable USER).

Example

The following example shows the enabling of all triggers on the table `OE.OC_ORDERS`:

```
SELECT trigger_name
,      status
FROM user_triggers
WHERE table_name = 'OC_ORDERS';

TRIGGER_NAME                   STATUS
------------------------------ --------
ORDERS_TRG                     DISABLED
ORDERS_ITEMS_TRG               DISABLED
```

Perform the following in SQL*Plus or Warehouse Builder to enable all triggers defined on the table `OE.OC_ORDERS`.

```
EXECUTE WB_ENABLE_ALL_TRIGGERS ('OC_ORDERS');

TRIGGER_NAME                   STATUS
------------------------------ --------
ORDERS_TRG                     ENABLED
ORDERS_ITEMS_TRG               ENABLED
```

WB_ENABLE_CONSTRAINT

Syntax

```
WB_ENABLE_CONSTRAINT(p_constraintname, p_tablename)
```

where *`p_constraintname`* is the constraint name to be disabled and *`p_tablename`* is the table name on which the specified constraint is defined.

Purpose

This program unit enables the specified constraint that is owned by the table as stated in the call to the program. The user is the current user (in variable `USER`). For faster loading of data sets, you can disable constraints on a table. After the loading is complete, you must re-enable these constraints. This program unit shows you how to enable the constraints one at a time.

Example

The following example shows the enabling of the specified constraint on the table `OE.CUSTOMERS`:

```
SELECT constraint_name
,      DECODE(constraint_type
       , 'C', 'Check'
       , 'P', 'Primary'
       ) Type
,      status
FROM user_constraints
WHERE table_name = 'CUSTOMERS';
```

```
CONSTRAINT_NAME                TYPE    STATUS
------------------------------ ------- --------
CUST_FNAME_NN                  Check   DISABLED
CUST_LNAME_NN                  Check   DISABLED
CUSTOMER_CREDIT_LIMIT_MAX      Check   DISABLED
CUSTOMER_ID_MIN                Check   DISABLED
CUSTOMERS_PK                   Primary DISABLED
```

Perform the following in SQL*Plus or Warehouse Builder to enable the specified constraint.

```
EXECUTE WB_ENABLE_CONSTRAINT('CUSTOMERS_PK', 'CUSTOMERS');

CONSTRAINT_NAME                TYPE    STATUS
------------------------------ ------- --------
CUST_FNAME_NN                  Check   DISABLED
CUST_LNAME_NN                  Check   DISABLED
CUSTOMER_CREDIT_LIMIT_MAX      Check   DISABLED
CUSTOMER_ID_MIN                Check   DISABLED
CUSTOMERS_PK                   Primary ENABLED
```

WB_ENABLE_TRIGGER

Syntax

```
WB_ENABLE_TRIGGER(p_name)
```

where `p_name` is the trigger name to be enabled.

Purpose

This program unit enables the specified trigger. The owner of the trigger must be the current user (in variable `USER`).

Example

The following example shows the enabling of a trigger on the table `OE.OC_ORDERS`:

```
SELECT trigger_name
,      status
FROM user_triggers
WHERE table_name = 'OC_ORDERS';

TRIGGER_NAME                   STATUS
------------------------------ --------
ORDERS_TRG                     DISABLED
ORDERS_ITEMS_TRG               ENABLED
```

Perform the following in SQL*Plus or Warehouse Builder to enable the specified constraint.

```
EXECUTE WB_ENABLE_TRIGGER ('ORDERS_TRG');

TRIGGER_NAME                   STATUS
------------------------------ --------
ORDERS_TRG                     ENABLED
ORDERS_ITEMS_TRG               ENABLED
```

WB_TRUNCATE_TABLE

Syntax

```
WB_TRUNCATE_TABLE(p_name)
```

where *p_name* is the table name to be truncated.

Purpose

This program unit truncates the table specified in the command call. The owner of the trigger must be the current user (in variable `USER`). The command disables and re-enables all referencing constraints to enable the truncate table command. Use this command in a pre-mapping process to explicitly truncate a staging table and ensure that all data in this staging table is newly loaded data.

Example

The following example shows the truncation of the table `OE.OC_ORDERS`:

```
SELECT COUNT(*) FROM oc_orders;

  COUNT(*)
----------
       105
```

Perform the following in SQL*Plus or Warehouse Builder to enable the specified constraint.

```
EXECUTE WB_TRUNCATE_TABLE ('OC_ORDERS');

  COUNT(*)
----------
         0
```

Character Transformations

Character transformations enable Warehouse Builder users to perform transformations on Character objects. The custom functions provided with Warehouse Builder are prefixed with `WB_`.

The character transformations available in Warehouse Builder are:

- ASCII on page 2-10
- CHR on page 2-10
- CONCAT on page 2-11
- INITCAP on page 2-12
- INSTR, INSTR2, INSTR4, INSTRB, INSTRC on page 2-12
- LENGTH, LENGTH2, LENGTH4, LENGTHB, LENGTHC on page 2-13
- LOWER on page 2-14
- LPAD on page 2-14
- LTRIM on page 2-15
- NLSSORT on page 2-15
- NLS_INITCAP on page 2-16

- NLS_LOWER on page 2-16
- NLS_UPPER on page 2-17
- REPLACE on page 2-17
- REGEXP_INSTR on page 2-18
- REGEXP_REPLACE on page 2-20
- REGEXP_SUBSTR on page 2-22
- RPAD on page 2-23
- RTRIM on page 2-24
- SOUNDEX on page 2-24
- SUBSTR, SUBSTR2, SUBSTR4, SUBSTRB, SUBSTRC on page 2-25
- TRANSLATE on page 2-26
- TRIM on page 2-27
- UPPER on page 2-27
- WB_LOOKUP_CHAR (number) on page 2-28
- WB_LOOKUP_CHAR (varchar2) on page 2-28
- WB_IS_SPACE on page 2-29

ASCII

Syntax

```
ascii::=ASCII(attribute)
```

Purpose

ASCII returns the decimal representation in the database character set of the first character of `attribute`. An `attribute` can be of data type CHAR, VARCHAR2, NCHAR, or NVARCHAR2. The value returned is of data type NUMBER. If your database character set is 7-bit ASCII, this function returns an ASCII value. If your database character set is EBCDIC Code, this function returns an EBCDIC value. There is no corresponding EBCDIC character function.

Example

The following example returns the ASCII decimal equivalent of the letter Q:

```
SELECT ASCII('Q') FROM DUAL;
ASCII('Q')
----------
81
```

CHR

Syntax

```
chr::=CHR(attribute)
```

Purpose

CHR returns the character with the binary equivalent to the number specified in the attribute in either the database character set or the national character set.

If USING NCHAR_CS is not specified, this function returns the character with the binary equivalent to attribute as a VARCHAR2 value in the database character set. If USING NCHAR_CS is specified in the expression builder, this function returns the character with the binary equivalent to attribute as a NVARCHAR2 value in the national character set.

Examples

The following example is run on an ASCII-based machine with the database character set defined as WE8ISO8859P1:

```
SELECT CHR(67)||CHR(65)||CHR(84) "Dog"
   FROM DUAL;

Dog
---
CAT
```

To produce the same results on an EBCDIC-based machine with the WE8EBCDIC1047 character set, modify the preceding example as follows:

```
SELECT CHR(195)||CHR(193)||CHR(227) "Dog"
   FROM DUAL;

Dog
---
CAT
```

The following example uses the UTF8 character set:

```
SELECT CHR (50052 USING NCHAR_CS)
   FROM DUAL;
CH
--
Ä
```

CONCAT

Syntax

```
concat::=CONCAT(attribute1, attribute2)
```

Purpose

CONCAT returns attribute1 concatenated with attribute2. Both attribute1 and attribute2 can be CHAR or VARCHAR2 data types. The returned string is of VARCHAR2 data type contained in the same character set as attribute1. This function is equivalent to the concatenation operator (||).

Example

This example uses nesting to concatenate three character strings:

```
SELECT CONCAT(CONCAT(last_name, '''s job category is '), job_id) "Job"
   FROM employees
   WHERE employee_id = 152;
```

```
Job
------------------------------------------------------
Hall's job category is SA_REP
```

INITCAP

Syntax

```
initcap::=INITCAP(attribute)
```

Purpose

`INITCAP` returns the content of the `attribute` with the first letter of each word in uppercase and all other letters in lowercase. Words are delimited by white space or by characters that are not alphanumeric. `Attribute` can be of the data types CHAR or VARCHAR2. The return value is the same data type as `attribute`.

Example

The following example capitalizes each word in the string:

```
SELECT INITCAP('the soap') "Capitals" FROM DUAL;

Capitals
---------
The Soap
```

INSTR, INSTR2, INSTR4, INSTRB, INSTRC

Syntax

```
instr::=INSTR(attribute1, attribute2, n, m)
instr2::=INSTR2(attribute1, attribute2, n, m)
instr4::=INSTR4(attribute1, attribute2, n, m)
instrb::=INSTRB(attribute1, attribute2, n, m)
instrc::=INSTRC(attribute1, attribute2, n, m)
```

Purpose

`INSTR` searches `attribute1` beginning with its nth character for the `m`th occurrence of `attribute2`. It returns the position of the character in `attribute1` that is the first character of this occurrence. `INSTRB` uses bytes instead of characters. `INSTRC` uses Unicode complete characters. `INSTR2` UCS2 code points. `INSTR4` uses UCS4 code points.

If `n` is negative, Oracle counts and searches backward from the end of `attribute1`. The value of `m` must be positive. The default values of both `n` and `m` are 1, which means that Oracle begins searching the first character of `attribute1` for the first occurrence of `attribute2`. The return value is relative to the beginning of `attribute1`, regardless of the value of `n`, and is expressed in characters. If the search is unsuccessful (if `attribute2` does not appear `m` times after the `n`th character of `attribute1`), then the return value is 0.

Examples

The following example searches the string "CORPORATE FLOOR", beginning with the third character, for the string "OR". It returns the position in `CORPORATE FLOOR` at which the second occurrence of "OR" begins:

```
SELECT INSTR('CORPORATE FLOOR','OR', 3, 2) "Instring"
   FROM DUAL;

Instring
----------
14
```

The next example begins searching at the third character from the end:

```
SELECT INSTR('CORPORATE FLOOR','OR', -3, 2) "Reversed Instring"
   FROM DUAL;

Reversed Instring
-----------------
2
```

This example assumes a double-byte database character set.

```
SELECT INSTRB('CORPORATE FLOOR','OR',5,2) "Instring in bytes"
   FROM DUAL;

Instring in bytes
-----------------
27
```

LENGTH, LENGTH2, LENGTH4, LENGTHB, LENGTHC

Syntax

```
length::=LENGTH(attribute)
length2::=LENGTH2(attribute)
length4::=LENGTH4(attribute)
lengthb::=LENGTHB(attribute)
lengthC::=LENGTHC(attribute)
```

Purpose

The length functions return the length of `attribute`, which can be of the data types `CHAR` or `VARCHAR2`. `LENGTH` calculates the length using characters as defined by the input character set. `LENGTHB` uses bytes instead of characters. `LENGTHC` uses Unicode complete characters. `LENGTH2` uses UCS2 code points. `LENGTH4` uses US4 code points. The return value is of data type NUMBER. If `attribute` has data type CHAR, the length includes all trailing blanks. If `attribute` contains a null value, this function returns null.

Example

The following examples use the `LENGTH` function using single- and multibyte database character set.

```
SELECT LENGTH('CANDIDE') "Length in characters"
   FROM DUAL;

Length in characters
--------------------
7
```

This example assumes a double-byte database character set.

```
SELECT LENGTHB ('CANDIDE') "Length in bytes"
   FROM DUAL;

Length in bytes
---------------
14
```

LOWER

Syntax

`lower::=LOWER(attribute)`

Purpose

`LOWER` returns `attribute`, with all letters in lowercase. The `attribute` can be of the data types CHAR and VARCHAR2. The return value is the same data type as that of `attribute`.

Example

The following example returns a string in lowercase:

```
SELECT LOWER('MR. SCOTT MCMILLAN') "Lowercase"
   FROM DUAL;

Lowercase
--------------------
mr. scott mcmillan
```

LPAD

Syntax

`lpad::=LPAD(attribute1, n, attribute2)`

Purpose

`LPAD` returns `attribute1`, left-padded to length `n` with the sequence of characters in `attribute2`. `Attribute2` defaults to a single blank. If `attribute1` is longer than `n`, this function returns the portion of `attribute1` that fits in `n`.

Both `attribute1` and `attribute2` can be of the data types CHAR and VARCHAR2. The string returned is of VARCHAR2 data type and is in the same character set as `attribute1`. The argument `n` is the total length of the return value as it is displayed on your screen. In most character sets, this is also the number of characters in the return value. However, in some multibyte character sets, the display length of a character string can differ from the number of characters in the string.

Example

The following example left-pads a string with the characters "*.":

```
SELECT LPAD('Page 1',15,'*.') "LPAD example"
   FROM DUAL;

LPAD example
---------------
*.*.*.*.*Page 1
```

LTRIM

Syntax

```
ltrim::=LTRIM(attribute, set)
```

Purpose

`LTRIM` removes characters from the left of `attribute`, with all the left most characters that appear in `set` removed. `Set` defaults to a single blank. If `attribute` is a character literal, you must enclose it in single quotes. Warehouse Builder begins scanning `attribute` from its first character and removes all characters that appear in `set` until it reaches a character absent in `set`. Then it returns the result.

Both `attribute` and `set` can be any of the data types CHAR and VARCHAR2. The string returned is of VARCHAR2 data type and is in the same character set as `attribute`.

Example

The following example trims all of the left-most x's and y's from a string:

```
SELECT LTRIM('xyxXxyLAST WORD','xy') "LTRIM example"
   FROM DUAL;

LTRIM example
------------
XxyLAST WORD
```

NLSSORT

Syntax

```
nlssort::=NLSSORT(attribute, nlsparam)
```

Purpose

`NLSSORT` returns the string of bytes used to sort `attribute`. The parameter `attribute` is of type VARCHAR2. Use this function to compare based on a linguistic sort of sequence rather than on the binary value of a string.

The value of `nlsparam` can have the form '`NLS_SORT = sort`' where sort is a linguistic sort sequence or `BINARY`. If you omit `nlsparam`, this function uses the default sort sequence for your session.

Example

The following example creates a table containing two values and shows how the values returned can be ordered by the `NLSSORT` function:

```
CREATE TABLE test (name VARCHAR2(15));
INSERT INTO TEST VALUES ('Gaardiner');
INSERT INTO TEST VALUES ('Gaberd');

SELECT * FROM test ORDER BY name;

NAME
------
Gaardiner
Gaberd
```

```
SELECT *
   FROM test
   ORDER BY NLSSORT(name, 'NLSSORT = XDanish');

Name
------
Gaberd
Gaardiner
```

NLS_INITCAP

Syntax

`nls_initcap::=NLS_INITCAP(attribute, nlsparam)`

Purpose

`NLS_INITCAP` returns `attribute`, with the first letter of each word in uppercase, all other letters in lowercase. Words are delimited by white space or characters that are not alphanumeric.

The value of `nlsparam` can have the form `'NLS_SORT = sort'`, where sort is either a linguistic sort sequence or BINARY. The linguistic sort sequence handles special linguistic requirements for case conversions. These requirements can result in a return value of a different length than the char. If you omit 'nlsparam', this function uses the default sort sequence for your session.

Example

The following examples show how the linguistic sort sequence results in a different return value from the function:

```
SELECT NLS_INITCAP('ijsland') "InitCap"
   FROM dual;

InitCap
---------
Ijsland

SELECT NLS_INITCAP('ijsland','NLS_SORT=XDutch) "InitCap"
   FROM dual;

InitCap
---------
IJsland
```

NLS_LOWER

Syntax

`nls_lower::=NLS_LOWER(attribute, nlsparam)`

Purpose

`NLS_LOWER` returns `attribute`, with all letters lowercase. Both `attribute` and `nlsparam` can be any of the data types CHAR, VARCHAR2, NCHAR, NVARCHAR2, CLOB, or NCLOB. The string returned is of data type VARCHAR2 and is in the same character set as `attribute`. The value of `nlsparam` can have the form `'NLS_SORT = sort'`, where sort is either a linguistic sort sequence or BINARY.

Example

The following example returns the character string `'citta''` using the XGerman linguistic sort sequence:

```
SELECT NLS_LOWER('CITTA''','NLS_SORT=XGerman) "Lowercase"
   FROM DUAL;

Lowercase
------------
citta'
```

NLS_UPPER

Syntax

```
nls_upper::=NLS_UPPER(attribute, nlsparam)
```

Purpose

`NLS_UPPER` returns `attribute`, with all letters uppercase. Both `attribute` and `nlsparam` can be any of the data types `CHAR`, `VARCHAR2`, `NCHAR`, `NVARCHAR2`, `CLOB`, or `NCLOB`. The string returned is of VARCHAR2 data type and is in the same character set as `attribute`. The value of `nlsparam` can have the form `'NLS_SORT = sort'`, where sort is either a linguistic sort sequence or BINARY.

Example

The following example returns a string with all letters converted to uppercase:

```
SELECT NLS_UPPER('große') "Uppercase"
   FROM DUAL;

Uppercase
------------
GROßE

SELECT NLS_UPPER('große', 'NLS_SORT=XGerman) "Uppercase"
   FROM DUAL;

Uppercase
------------
GROSSE
```

REPLACE

Syntax

```
replace::=REPLACE(attribute, 'search_string', 'replacement_string')
```

Purpose

`REPLACE` returns an `attribute` with every occurrence of `search_string` replaced with `replacement_string`. If `replacement_string` is omitted or null, all occurrences of `search_string` are removed. If `search_string` is null, `attribute` is returned.

Both `search_string` and `replacement_string`, as well as `attribute`, can be of the data types CHAR or VARCHAR2. The string returned is of VARCHAR2 data type and is in the same character set as `attribute`.

This function provides a superset of the functionality provided by the `TRANSLATE` function. `TRANSLATE` provides single-character, one-to-one substitution. `REPLACE` enables you to substitute one string for another, as well as to remove character strings.

Example

The following example replaces occurrences of "J" with "BL":

```
SELECT REPLACE('JACK and JUE','J','BL') "Changes"
   FROM DUAL;

Changes
--------------
BLACK and BLUE
```

REGEXP_INSTR

Syntax

```
regexp_instr:=REGEXP_INSTR(source_string, pattern, position, occurance,
                           return_option, match_parameter)
```

Purpose

`REGEXP_INSTR` extends the functionality of the `INSTR` function by letting you search a string for a regular expression pattern. The function evaluates strings using characters as defined by the input character set. It returns an integer indicating the beginning or ending position of the matched substring, depending on the value of the `return_option` argument. If no match is found, the function returns 0.

This function complies with the POSIX regular expression standard and the Unicode Regular Expression Guidelines.

- `source_string` is a character expression that serves as the search value. It is commonly a character column and can be of any of the datatypes CHAR, VARCHAR2, NCHAR, NVARCHAR2, CLOB, or NCLOB.
- `pattern` is the regular expression. It is usually a text literal and can be of any of the datatypes CHAR, VARCHAR2, NCHAR, or NVARCHAR2. It can contain up to 512 bytes. If the datatype of pattern is different from the datatype of source_ string, Oracle Database converts pattern to the datatype of source_ string. For a listing of the operators you can specify in pattern, please refer to Appendix C, "Oracle Regular Expression Support".
- `position` is a positive integer indicating the character of source_string where Oracle should begin the search. The default is 1, meaning that Oracle begins the search at the first character of source_string.
- `occurrence` is a positive integer indicating which occurrence of pattern in source_string Oracle should search for. The default is 1, meaning that Oracle searches for the first occurrence of pattern.
- `return_option` lets you specify what Oracle should return in relation to the occurrence:

  - If you specify 0, then Oracle returns the position of the first character of the occurrence. This is the default.
  - If you specify 1, then Oracle returns the position of the character following the occurrence.

- `match_parameter` is a text literal that lets you change the default matching behavior of the function. You can specify one or more of the following values for match_parameter:
  - 'i' specifies case-insensitive matching.
  - 'c' specifies case-sensitive matching.
  - 'n' allows the period (.), which is the match-any-character character, to match the newline character. If you omit this parameter, the period does not match the newline character.
  - 'm' treats the source string as multiple lines. Oracle interprets ^ and $ as the start and end, respectively, of any line anywhere in the source string, rather than only at the start or end of the entire source string. If you omit this parameter, Oracle treats the source string as a single line.

  If you specify multiple contradictory values, Oracle uses the last value. For example, if you specify 'ic', then Oracle uses case-sensitive matching. If you specify a character other than those shown above, then Oracle returns an error.

  If you omit `match_parameter`, then:
  - The default case sensitivity is determined by the value of the NLS_SORT parameter.
  - A period (.) does not match the newline character.
  - The source string is treated as a single line.

Example

The following example examines the string, looking for occurrences of one or more non-blank characters. Oracle begins searching at the first character in the string and returns the starting position (default) of the sixth occurrence of one or more non-blank characters.

```
SELECT
     REGEXP_INSTR('500 Oracle Parkway, Redwood Shores, CA', '[^ ]+', 1, 6) FROM
DUAL;

REGEXP_INSTR
------------
37
```

The following example examines the string, looking for occurrences of words beginning with s, r, or p, regardless of case, followed by any six alphabetic characters. Oracle begins searching at the third character in the string and returns the position in the string of the character following the second occurrence of a seven letter word beginning with s, r, or p, regardless of case.

```
SELECT
     REGEXP_INSTR('500 Oracle Parkway, Redwood Shores, CA',
     '[s|r|p][[:alpha:]]{6}', 3, 2, 1, 'i')
   FROM DUAL;

REGEXP_INSTR
------------
```

```
28
```

REGEXP_REPLACE

Syntax

```
regexp_replace:=REGEXP_REPLACE(source_string, pattern, replace_string,
                               position, occurance, match_parameter)
```

Purpose

REGEXP_REPLACE extends the functionality of the REPLACE function by letting you search a string for a regular expression pattern. By default, the function returns source_string with every occurrence of the regular expression pattern replaced with `replace_string`. The string returned is n the same character set as source_ string. The function returns VARCHAR2 if the first argument is not a LOB and returns CLOB if the first argument is a LOB.

This function complies with the POSIX regular expression standard and the Unicode Regular Expression Guidelines.

- `source_string` is a character expression that serves as the search value. It is commonly a character column and can be of any of the datatypes CHAR, VARCHAR2, NCHAR, NVARCHAR2, CLOB or NCLOB.
- `pattern` is the regular expression. It is usually a text literal and can be of any of the datatypes CHAR, VARCHAR2, NCHAR, or NVARCHAR2. It can contain up to 512 bytes. If the datatype of pattern is different from the datatype of source_ string, Oracle Database converts pattern to the datatype of source_ string.
- `replace_string` can be of any of the datatypes CHAR, VARCHAR2, NCHAR, NVARCHAR2, CLOB, or NCLOB. If replace_string is a CLOB or NCLOB, then Oracle truncates replace_string to 32K. The replace_string can contain up to 500 backreferences to subexpressions in the form `\n`, where n is a number from 1 to 9. If n is the blackslash character in replace_string, then you must precede it with the escape character (`\\`).
- `position` is a positive integer indicating the character of source_string where Oracle should begin the search. The default is 1, meaning that Oracle begins the search at the first character of source_string.
- `occurrence` is a nonnegative integer indicating the occurrence of the replace operation:
  - If you specify 0, then Oracle replaces all occurrences of the match.
  - If you specify a positive integer n, then Oracle replaces the nth occurrence.
- `match_parameter` is a text literal that lets you change the default matching behavior of the function. This argument affects only the matching process and has no effect on `replace_string`. You can specify one or more of the following values for `match_parameter`:
  - 'i' specifies case-insensitive matching.
  - 'c' specifies case-sensitive matching.
  - 'n' allows the period (.), which is the match-any-character character, to match the newline character. If you omit this parameter, the period does not match the newline character.

- 'm' treats the source string as multiple lines. Oracle interprets ^ and $ as the start and end, respectively, of any line anywhere in the source string, rather than only at the start or end of the entire source string. If you omit this parameter, Oracle treats the source string as a single line.

If you specify multiple contradictory values, Oracle uses the last value. For example, if you specify 'ic', then Oracle uses case-sensitive matching. If you specify a character other than those shown above, then Oracle returns an error. If you omit `match_parameter`, then:

- The default case sensitivity is determined by the value of the NLS_SORT parameter.
- A period (.) does not match the newline character.
- The source string is treated as a single line.

Example

The following example examines `phone_number`, looking for the pattern `xxx.xxx.xxxx`. Oracle reformats this pattern with `(xxx) xxx-xxxx`.

```
SELECT
     REGEXP_REPLACE(phone_number,
       '([[:digit:]]{3})\.([[:digit:]]{3})\.([[:digit:]]{4})',
       '(\1) \2-\3')
FROM employees;

REGEXP_REPLACE
--------------------------------------------------------------------------------
(515) 123-4567
(515) 123-4568
(515) 123-4569
(590) 423-4567
. . .
```

The following example examines `country_name`. Oracle puts a space after each non-null character in the string.

```
SELECT
     REGEXP_REPLACE(country_name, '(.)', '\1 ') "REGEXP_REPLACE"
FROM countries;

REGEXP_REPLACE
--------------------------------------------------------------------------------
A r g e n t i n a
A u s t r a l i a
B e l g i u m
B r a z i l
C a n a d a
. . .
```

The following example examines the string, looking for two or more spaces. Oracle replaces each occurrence of two or more spaces with a single space.

```
SELECT
     REGEXP_REPLACE('500 Oracle Parkway, Redwood Shores, CA','( ){2,}', ' ')
FROM DUAL;

REGEXP_REPLACE
--------------------------------------
500 Oracle Parkway, Redwood Shores, CA
```

REGEXP_SUBSTR

Syntax

```
regexp_substr:=REGEXP_SUBSTR(source_string, pattern, position,
                             occurance, match_parameter)
```

Purpose

REGEXP_SUBSTR extends the functionality of the SUBSTR function by letting you search a string for a regular expression pattern. It is also similar to REGEXP_INSTR, but instead of returning the position of the substring, it returns the substring itself. This function is useful if you need the contents of a match string but not its position in the source string. The function returns the string as VARCHAR2 or CLOB data in the same character set as source_string.

This function complies with the POSIX regular expression standard and the Unicode Regular Expression Guidelines.

- `source_string` is a character expression that serves as the search value. It is commonly a character column and can be of any of the datatypes CHAR, VARCHAR2, NCHAR, NVARCHAR2, CLOB or NCLOB.
- `pattern` is the regular expression. It is usually a text literal and can be of any of the datatypes CHAR, VARCHAR2, NCHAR, or NVARCHAR2. It can contain up to 512 bytes. If the datatype of pattern is different from the datatype of source_ string, Oracle Database converts pattern to the datatype of source_ string.
- `replace_string` can be of any of the datatypes CHAR, VARCHAR2, NCHAR, NVARCHAR2, CLOB, or NCLOB. If replace_string is a CLOB or NCLOB, then Oracle truncates replace_string to 32K. The replace_string can contain up to 500 backreferences to subexpressions in the form \n, where n is a number from 1 to 9. If n is the blackslash character in replace_string, then you must precede it with the escape character (\\).
- `position` is a positive integer indicating the character of source_string where Oracle should begin the search. The default is 1, meaning that Oracle begins the search at the first character of source_string.
- `occurrence` is a nonnegative integer indicating the occurrence of the replace operation:
  - If you specify 0, then Oracle replaces all occurrences of the match.
  - If you specify a positive integer n, then Oracle replaces the nth occurrence.
- `match_parameter` is a text literal that lets you change the default matching behavior of the function. This argument affects only the matching process and has no effect on replace_string. You can specify one or more of the following values for match_parameter:
  - 'i' specifies case-insensitive matching.
  - 'c' specifies case-sensitive matching.
  - 'n' allows the period (.), which is the match-any-character character, to match the newline character. If you omit this parameter, the period does not match the newline character.

- 'm' treats the source string as multiple lines. Oracle interprets ^ and $ as the start and end, respectively, of any line anywhere in the source string, rather than only at the start or end of the entire source string. If you omit this parameter, Oracle treats the source string as a single line.

If you specify multiple contradictory values, Oracle uses the last value. For example, if you specify 'ic', then Oracle uses case-sensitive matching. If you specify a character other than those shown above, then Oracle returns an error. If you omit match_parameter, then:

- The default case sensitivity is determined by the value of the NLS_SORT parameter.
- A period (.) does not match the newline character.
- The source string is treated as a single line.

Example

The following example examines the string, looking for the first substring bounded by commas. Oracle Database searches for a comma followed by one or more occurrences of non-comma characters followed by a comma. Oracle returns the substring, including the leading and trailing commas.

```
SELECT
     REGEXP_SUBSTR('500 Oracle Parkway, Redwood Shores, CA',',[^,]+,')
FROM DUAL;

REGEXPR_SUBSTR
-----------------
, Redwood Shores,
```

The following example examines the string, looking for http:// followed by a substring of one or more alphanumeric characters and optionally, a period (.). Oracle searches for a minimum of three and a maximum of four occurrences of this substring between http:// and either a slash (/) or the end of the string.

```
SELECT
     REGEXP_SUBSTR('http://www.oracle.com/products',
                   'http://([[:alnum:]]+\.?){3,4}/?')
FROM DUAL;

REGEXP_SUBSTR
----------------------
http://www.oracle.com/
```

RPAD

Syntax

```
rpad::=RPAD(attribute1, n, attribute2)
```

Purpose

RPAD returns `attribute1`, right-padded to length `n` with `attribute2`, replicated as many times as necessary. `Attribute2` defaults to a single blank. If `attribute1` is longer than `n`, this function returns the portion of `attribute1` that fits in `n`.

Both `attribute1` and `attribute2` can be of the data types CHAR or VARCHAR2. The string returned is of VARCHAR2 data type and is in the same character set as `attribute1`.

The argument `n` is the total length of the return value as it is displayed on your screen. In most character sets, this is also the number of characters in the return value. However, in some multibyte character sets, the display length of a character string can differ from the number of characters in the string.

Example

The following example rights-pads a name with the letters "ab" until it is 12 characters long:

```
SELECT RPAD('MORRISON',12,'ab') "RPAD example"
   FROM DUAL;

RPAD example
-----------------
MORRISONabab
```

RTRIM

Syntax

```
rtrim::=RTRIM(attribute, set)
```

Purpose

`RTRIM` returns `attribute`, with all the right most characters that appear in `set` removed; `set` defaults to a single blank. If `attribute` is a character literal, you must enclose it in single quotes. `RTRIM` works similarly to `LTRIM`. Both `attribute` and `set` can be any of the data types CHAR or VARCHAR2. The string returned is of VARCHAR2 data type and is in the same character set as `attribute`.

Example

The following example trims the letters "xy" from the right side of a string:

```
SELECT RTRIM('BROWNINGyxXxy','xy') "RTRIM e.g."
   FROM DUAL;

RTRIM e.g
-------------
BROWNINGyxX
```

SOUNDEX

Syntax

```
soundex::=SOUNDEX(attribute)
```

Purpose

`SOUNDEX` returns a character string containing the phonetic representation of `attribute`. This function enables you to compare words that are spelled differently, but sound similar in English.

The phonetic representation is defined in *The Art of Computer Programming, Volume 3: Sorting and Searching*, by Donald E. Knuth, as follows:

- Retain the first letter of the string and remove all other occurrences of the following letters: a, e, h, i, o, u, w, y.
- Assign numbers to the remaining letters (after the first) as follows:
  - b, f, p, v = 1
  - c, g, j, k, q, s, x, z = 2
  - d, t = 3
  - l = 4
  - m, n = 5
  - r = 6
- If two or more letters with the same number were adjacent in the original name (before step 1), or adjacent except for any intervening h and w, omit all but the first.
- Return the first four bytes padded with 0.

Data types for `attribute` can be CHAR and VARCHAR2. The return value is the same data type as `attribute`.

Example

The following example returns the employees whose last names are a phonetic representation of "Smyth":

```
SELECT last_name, first_name
   FROM hr.employees
   WHERE SOUNDEX(last_name) = SOUNDEX('SMYTHE');


LAST_NAME  FIRST_NAME
---------- ----------
Smith      Lindsey
```

SUBSTR, SUBSTR2, SUBSTR4, SUBSTRB, SUBSTRC

Syntax

```
substr::=SUBSTR(attribute, position, substring_length)
substr2::=SUBSTR2(attribute, position, substring_length)
substr4::=SUBSTR4(attribute, position, substring_length)
substrb::=SUBSTRB(attribute, position, substring_length)
substrc::=SUBSTRC(attribute, position, substring_length)
```

Purpose

The substring functions return a portion of `attribute`, beginning at character `position`, `substring_length` characters long. `SUBSTR` calculates lengths using characters as defined by the input character set. `SUBSTRB` uses bytes instead of characters. `SUBSTRC` uses Unicode complete characters. `SUBSTR2` uses UCS2 code points. `SUBSTR4` uses UCS4 code points.

- If `position` is 0, it is treated as 1.

- If `position` is positive, Warehouse Builder counts from the beginning of `attribute` to find the first character.
- If `position` is negative, Warehouse Builder counts backward from the end of `attribute`.
- If `substring_length` is omitted, Warehouse Builder returns all characters to the end of `attribute`. If `substring_length` is less than 1, a `null` is returned.

Data types for `attribute` can be CHAR and VARCHAR2. The return value is the same data type as `attribute`. Floating-point numbers passed as arguments to `SUBSTR` are automatically converted to integers.

Examples

The following example returns several specified substrings of "ABCDEFG":

```
SELECT SUBSTR('ABCDEFG',3,4) "Substring"
   FROM DUAL;

Substring
---------
CDEF

SELECT SUBSTR('ABCDEFG',-5,4) "Substring"
   FROM DUAL;

Substring
---------
CDEF
```

Assume a double-byte database character set:

```
SELECT SUBSTRB('ABCDEFG',5,4.2) "Substring with bytes"
   FROM DUAL;

Substring with bytes
--------------------
CD
```

TRANSLATE

Syntax

```
translate::=TRANSLATE(attribute, from_string, to_string)
```

Purpose

`TRANSLATE` returns `attribute` with all occurrences of each character in `from_string` replaced by its corresponding character in `to_string`. Characters in `attribute` that are not in `from_string` are not replaced. The argument `from_string` can contain more characters than `to_string`. In this case, the extra characters at the end of `from_string` have no corresponding characters in `to_string`. If these extra characters appear in `attribute`, they are removed from the return value.

You cannot use an empty string for `to_string` to remove all characters in `from_string` from the return value. Warehouse Builder interprets the empty string as null, and if this function has a null argument, it returns `null`.

Examples

The following statement translates a license number. All letters 'ABC...Z' are translated to 'X' and all digits '012 . . . 9' are translated to '9':

```
SELECT TRANSLATE('2KRW229','0123456789ABCDEFGHIJKLMNOPQRSTUVWXYZ',
'9999999999XXXXXXXXXXXXXXXXXXXXXXXXXX') "License"
FROM DUAL;

License
--------
9XXX999
```

The following statement returns a license number with the characters removed and the digits remaining:

```
SELECT TRANSLATE('2KRW229','0123456789ABCDEFGHIJKLMNOPQRSTUVWXYZ', '0123456789')
"Translate example"
FROM DUAL;

Translate example
-----------------
2229
```

TRIM

Syntax

```
trim::=TRIM(attribute)
```

Purpose

`TRIM` enables you to trim leading or trailing spaces (or both) from a character string. The function returns a value with data type VARCHAR2. The maximum length of the value is the length of `attribute`.

Example

This example trims leading and trailing spaces from a string:

```
SELECT TRIM ('   Warehouse    ') "TRIM Example"
   FROM DUAL;

TRIM example
------------
Warehouse
```

UPPER

Syntax

```
upper::=UPPER(attribute)
```

Purpose

`UPPER` returns `attribute`, with all letters in uppercase; `attribute` can be of the data types CHAR and VARCHAR2. The return value is the same data type as `attribute`.

Example

The following example returns a string in uppercase:

```
SELECT UPPER('Large') "Uppercase"
   FROM DUAL;

Upper
-----
LARGE
```

WB_LOOKUP_CHAR (number)

Syntax

```
WB.LOOKUP_CHAR (table_name
, column_name
, key_column_name
, key_value
)
```

where `table_name` is the name of the table to perform the lookup on and `column_name` is the name of the VARCHAR2 column that will be returned. For example, the result of the lookup `key_column_name` is the name of the NUMBER column used as the key to match on in the lookup table, `key_value` is the value of the key column mapped into the `key_column_name` with which the match will be done.

Purpose

To perform a key lookup on a number that returns a VARCHAR2 value from a database table using a NUMBER column as the matching key.

Example

Consider the following table as a lookup table LKP1:

```
KEY_COLUMN    TYPE    COLOR
10            Car     Red
20            Bike    Green
```

Using this package with the following call:

```
WB.LOOKUP_CHAR ('LKP1'
, 'TYPE'
, 'KEYCOLUMN'
, 20
)
```

returns the value of 'Bike' as output of this transform. This output would then be processed in the mapping as the result of an inline function call.

Note: This function is a row-based key lookup. Set-based lookups are supported when you use the lookup operator.

WB_LOOKUP_CHAR (varchar2)

Syntax

```
WB.LOOKUP_CHAR (table_name
, column_name
, key_column_name
, key_value
)
```

where `table_name` is the name of the table to perform the lookup on; `column_name` is the name of the VARCHAR2 column that will be returned, for instance, the result of the lookup; `key_column_name` is the name of the VARCHAR2 column used as the key to match on in the lookup table; `key_value` is the value of the key column, for instance, the value mapped into the `key_column_name` with which the match will be done.

Purpose

To perform a key lookup on a VARCHAR2 character that returns a VARCHAR2 value from a database table using a VARCHAR2 column as the matching key.

Example

Consider the following table as a lookup table LKP1:

```
KEYCOLUMN  TYPE  COLOR
ACV        Car   Red
ACP        Bike  Green
```

Using this package with the following call:

```
WB.LOOKUP_CHAR ('LKP1'
, 'TYPE'
, 'KEYCOLUMN'
, 'ACP'
)
```

returns the value of 'Bike' as output of this transformation. This output is then processed in the mapping as the result of an inline function call.

> **Note:** This function is a row-based key lookup. Set-based lookups are supported when you use the lookup operator.

WB_IS_SPACE

Syntax

```
WB_IS_SPACE(attibute)
```

Purpose

Checks whether a string value only contains spaces. This function returns a Boolean value. In mainframe sources, some fields contain many spaces to make a file adhere to the fixed length format. This function provides a way to check for these spaces.

Example

`WB_IS_SPACE` returns TRUE if attribute contains only spaces.

Control Center Transformations

Control Center transformations are used in a process flow or in custom transformations to enable you to access information about the Control Center at execution time. For example, you can use a Control Center transformation in the expression on a transition to help control the flow through a process flow at execution time. You can also use Control Center transformations within custom functions. These custom functions can in turn be used in the design of your process flow.

All Control Center transformations require an audit ID that provides a handle to the audit data stored in the Control Center repository. The audit ID is a key into the public view `ALL_RT_AUDIT_EXECUTIONS`. The transformations can be used to obtain data specific to that audit ID at execution time. When run in the context of a process flow, you can obtain the audit ID at execution time using the pseudo variable `audit_id` in a process flow expression. This variable is evaluated as the audit ID of the currently executing job. For example, for a map input parameter, this represents the map execution and for a transition this represents the job at the source of the transition.

The Control Center transformations are:

- WB_RT_GET_ELAPSED_TIME on page 2-30
- WB_RT_GET_JOB_METRICS on page 2-31
- WB_RT_GET_LAST_EXECUTION_TIME on page 2-31
- WB_RT_GET_MAP_RUN_AUDIT on page 2-32
- WB_RT_GET_NUMBER_OF_ERRORS on page 2-33
- WB_RT_GET_NUMBER_OF_WARNINGS on page 2-33
- WB_RT_GET_PARENT_AUDIT_ID on page 2-34
- WB_RT_GET_RETURN_CODE on page 2-34
- WB_RT_GET_START_TIME on page 2-35

WB_RT_GET_ELAPSED_TIME

Syntax

```
WB_RT_GET_ELAPSED_TIME(audit_id)
```

Purpose

This function returns the elapsed time, in seconds, for the job execution given by the specified `audit_id`. It returns null if the specified audit ID does not exist. For example, you can use this function on a transition if you want to make a choice dependent on the time taken by the previous activity.

Example

The following example returns the time elapsed since the activity represented by `audit_id` was started:

```
declare
   audit_id NUMBER := 1812;
   l_time NUMBER;
begin
   l_time:= WB_RT_GET_ELAPSED_TIME(audit_id);
end;
```

WB_RT_GET_JOB_METRICS

Syntax

```
WB_RT_GET_JOB_METRICS(audit_id, no_selected, no_deleted, no_updated, no_inserted,
no_discarded, no_merged, no_corrected)
```

where `no_selected` represents the number of rows selected, `no_deleted` represents the number of rows deleted, `no_updated` represents the number of rows updated, `no_inserted` represents the number of rows inserted, `no_discarded` represents the number of rows discarded, `no_merged` represents the number of rows merged, and `no_corrected` represents the number of rows corrected during the job execution.

Purpose

This procedure returns the metrics of the job execution represented by the specified `audit_id`. The metrics include the number of rows selected, deleted, updated, inserted, merged, and corrected.

Example

The following example retrieves the job metrics for the audit ID represented by `audit_id`.

```
declare
   audit_id NUMBER := 16547;
   l_nselected NUMBER;
   l_ndeleted NUMBER;
   l_nupdated NUMBER;
   l_ninserted NUMBER;
   l_ndiscarded NUMBER;
   l_nmerged NUMBER;
   l_ncorrected NUMBER;
begin
   WB_RT_GET_JOB_METRICS(audit_id, l_nselected, l_ndeleted, l_nupdated,
                         l_ninserted, l_ndiscarded, l_nmerged, l_ncorrected);
   dbms_output.put_line('sel=' || l_nselected || ', del=' l_ndeleted ||
                        ', upd=' || l_nupdated);
   dbms_output.put_line('ins='|| l_ninserted || ' , dis=' || l_ndiscarded );
   dbms_output.put_line('mer=' || l_nmerged || ', cor=' ||l_ncorrected);
 end;
```

WB_RT_GET_LAST_EXECUTION_TIME

Syntax

```
WB_RT_GET_LAST_EXECUTION_TIME(objectName, objectType, objectLocationName)
```

where `objectName` represents the name of the object, `objectType` represents the type of the object (for example MAPPING, DATA_AUDITOR, PROCESS_FLOW, SCHEDULABLE), and `objectLocationName` represents the location to which the object is deployed.

Purpose

This transformation gives you access to time-based data. Typically, you can use this in a Process Flow to model some design aspect that is relevant to "time". For example you can design a path that may execute different maps if the time since the last execution is more than 1 day.

You can also use this transformation to determine time-synchronization across process flows that are running concurrently. For example, you can choose a path in a process flow according to whether another Process Flow has completed.

Example

The following example retrieves the time when the mapping TIMES_MAP was last executed and the if condition determines whether this time was within 1 day of the current time. Based on this time, it can perform different actions.

```
declare
    last_exec_time DATE;
begin
    last_exec_time:=WB_RT_GET_LAST_EXECUTION_TIME('TIMES_MAP','MAPPING','WH_
LOCATION');
    if last_exec_time < sysdate - 1 then
--       last-execution was more than one day ago
--       provide details of action here
          NULL;
    Else
--       provide details of action here
          NULL;
    end if;
end;
```

WB_RT_GET_MAP_RUN_AUDIT

Syntax

```
WB_RT_GET_MAP_RUN_AUDIT(audit_id)
```

Purpose

This function returns the map run ID for a job execution that represents a map activity. It returns null if `audit_id` does not represent the job execution for a map. For example, you can use the returned ID as a key to access the ALL_RT_MAP_RUN_ <name> views for more information.

Example

The following example retrieves the map run ID for a job execution whose audit ID is 67265. It then uses this map run ID to obtain the name of the source from the ALL_RT_ MAP_RUN_EXECUTIONS public view.

```
declare
  audit_id NUMBER := 67265;
  l_sources VARCHAR2(256);
  l_run_id NUMBER;
begin
  l_run_id := WB_RT_GET_MAP_RUN_AUDIT_ID(audit_id);
  SELECT source_name INTO l_sources FROM all_rt_map_run_sources
         WHERE map_run_id = l_run_id;
end;
```

WB_RT_GET_NUMBER_OF_ERRORS

Syntax

```
WB_RT_GET_NUMBER_OF_ERRORS(audit_id)
```

Purpose

This function returns the number of errors recorded for the job execution given by the specified `audit_id`. It returns null if the specific `audit_id` is not found.

Example

The following example retrieves the number of errors generated by the job execution whose audit ID is 8769. You can then perform different actions based on the number of errors.

```
declare
   audit_id NUMBER := 8769;
   l_errors NUMBER;
begin
   l_errors := WB_RT_GET_NUMBER_OF_ERRORS(audit_id);
   if l_errors < 5 then
      .....
   else
      .....
   end if;
end;
```

WB_RT_GET_NUMBER_OF_WARNINGS

Syntax

```
WB_RT_GET_NUMBER_OF_WARNINGS(audit_id)
```

Purpose

This function returns the number of warnings recorded for the job executions represented by `audit_id`. It returns null if `audit_id` does not exist.

Example

The following example returns the number of warnings generated by the job execution whose audit ID is 54632. You can then perform different actions based on the number of warnings.

```
declare
   audit_is NUMBER := 54632;
   l_warnings NUMBER;
begin
   l_ warnings:= WB_RT_GET_NUMBER_OF_WARNINGS (audit_id);
   if l_warnings < 5 then
      .....
   else
      .....
   end if;
end;
```

WB_RT_GET_PARENT_AUDIT_ID

Syntax

```
WB_RT_GET_PARENT_AUDIT_ID(audit_id)
```

Purpose

This function returns the audit id for the process that owns the job execution represented by audit_id. It returns null if audit_id does not exist. You can then use the returned audit id as a key into other public views such as ALL_RT_AUDIT_EXECUTIONS, or other Control Center transformations if further information is required.

Example

The following example retrieves the parent audit ID for a job execution whose audit ID is 76859. It then uses this audit ID to determine the elapsed time for the parent activity. You can perform different actions based on the elapsed time of the parent activity.

```
declare
   audit_id NUMBER := 76859;
   l_elapsed_time NUMBER;
   l_parent_id NUMBER;
begin
   l_parent_id := WB_RT_GET_PARENT_AUDIT_ID(audit_id);
   l_elapsed_time := WB_RT_GET_ELAPSED_TIME(l_parent_id);
   if l_elpased_time < 100 then
      .....
   else
      .....
   end if;
end;
```

WB_RT_GET_RETURN_CODE

Syntax

```
WB_RT_GET_RETURN_CODE(audit_id)
```

Purpose

This function returns the return code recorded for the job execution represented by `audit_id`. It returns null if `audit_id` does not exist. For a successful job execution, the return code is greater than or equal to 0. A return code of less than 0 signifies that the job execution has failed.

Example

The following example retrieves the return code for the job execution whose audit ID is represented by `audit_id`.

```
declare
   audit_id NUMBER:=69;
   l_code NUMBER;
begin
   l_code:= WB_RT_GET_RETURN_CODE(audit_id);
end;
```

WB_RT_GET_START_TIME

Syntax

```
WB_RT_GET_START_TIME(audit_id)
```

Purpose

This function returns the start time for the job execution represented by `audit_id`. It returns null if `audit_id` does not exist. For example, you can use this in a transition if you wanted to make a choice dependent on when the previous activity started.

Example

The following example determines the start time of the job execution whose audit ID is 354.

```
declare
   audit_id NUMBER:=354;
   l_date TIMESTAMP WITH TIMEZONE;
begin
   l_date := WB_RT_GET_START_TIME(audit_id);
end;
```

Conversion Transformations

The conversion transformations enable Warehouse Builder users to perform functions that allow conditional conversion of values. These functions achieve "if -then" constructions within SQL.

The conversion transformations available in Warehouse Builder are:

- ASCIISTR on page 2-36
- COMPOSE on page 2-36
- CONVERT on page 2-37
- HEXTORAW on page 2-37
- NUMTODSINTERVAL on page 2-38
- NUMTOYMINTERVAL on page 2-39
- RAWTOHEX on page 2-39
- RAWTONHEX on page 2-40
- SCN_TO_TIMESTAMP on page 2-40
- TIMESTAMP_TO_SCN on page 2-41
- TO_BINARY_DOUBLE on page 2-42
- TO_BINARY_FLOAT on page 2-43
- TO_CHAR on page 2-43
- TO_CLOB on page 2-45
- TO_DATE on page 2-45
- TO_DSINTERVAL on page 2-45
- TO_MULTI_BYTE on page 2-46

- TO_MULTI_BYTE on page 2-46
- TO_NCHAR on page 2-46
- TO_NCLOB on page 2-47
- TO_NUMBER on page 2-47
- TO_SINGLE_BYTE on page 2-48
- TO_TIMESTAMP on page 2-48
- TO_TIMESTAMP_TZ on page 2-49
- TO_YMINTERVAL on page 2-50
- UNISTR on page 2-50

ASCIISTR

Syntax

```
asciistr::=ASCII(attribute)
```

Purpose

`ASCIISTR` takes as its argument a string of data type VARCHAR2 and returns an ASCII version of the string. Non-ASCII characters are converted to the form \xxxx, where xxxx represents a UTF-16 code unit.

Example

The following example returns the ASCII string equivalent of the text string 'ABÄDE':

```
SELECT ASCIISTR('ABÄDE') FROM DUAL;

ASCIISTR('
----------
AB\00C4CDE
```

COMPOSE

Syntax

```
compose::=COMPOSE(attribute)
```

Purpose

`COMPOSE` returns a Unicode string in its fully normalized form in the same character set as the input. The parameter `attribute` can be any of the data types CHAR, VARCHAR2, NCHAR, NVARCHAR2, CLOB, or NCLOB. For example, an `o` code point qualified by an umlaut code point will be returned as the o-umlaut code point.

Example

The following example returns the o-umlaut code point:

```
SELECT COMPOSE('o' || UNISTR('\038')) FROM DUAL;

CO
---
ö
```

CONVERT

Syntax

```
convert::=CONVERT(attribute, dest_char_set, source_char_set)
```

Purpose

`CONVERT` converts a character string specified in an operator `attribute` from one character set to another. The data type of the returned value is VARCHAR2.

- The `attribute` argument is the value to be converted. It can of the data types CHAR and VARCHAR2.
- The *`dest_char_set`* argument is the name of the character set to which *`attribute`* is converted.
- The `source_char_set` argument is the name of the character set in which `attribute` is stored in the database. The default value is the database character set.

Both the destination and source character set arguments can be either literals or columns containing the name of the character set. For complete correspondence in character conversion, the destination character set must contain a representation of all the characters defined in the source character set. When a character does not exist in the destination character set, it is substituted with a replacement character. Replacement characters can be defined as part of a character set definition.

Example

The following example illustrates character set conversion by converting a Latin-1 string to ASCII. The result is the same as importing the same string from a WE8ISO8859P1 database to a US7ASCII database.

```
SELECT CONVERT('Ä Ê Í Õ Ø A B C D E ', 'US7ASCII', 'WE8ISO8859P1')
   FROM DUAL;

CONVERT('ÄÊÍÕØABCDE'
---------------------
A E I ? ? A B C D E ?
```

Common character sets include:

- US7ASCII: US 7-bit ASCII character set
- WE8DEC: West European 8-bit character set
- WE8HP: HP West European Laserjet 8-bit character set
- F7DEC: DEC French 7-bit character set
- WE8EBCDIC500: IBM West European EBCDIC Code Page 500
- WE8PC850: IBM PC Code Page 850
- WE8ISO8859P1: ISO 8859-1 West European 8-bit character set

HEXTORAW

Syntax

```
hextoraw::=HEXTORAW(attribute)
```

Purpose

`HEXTORAW` converts `attribute` containing hexadecimal digits in the CHAR, VARCHAR2, NCHAR, or NVARCHAR2 character set to a raw value. This function does not support CLOB data directly. However, CLOBs can be passed in as arguments through implicit data conversion.

Example

The following example creates a simple table with a raw column, and inserts a hexadecimal value that has been converted to RAW:

```
CREATE TABLE test (raw_col RAW(10));

INSERT INTO test VALUES (HEXTORAW('7D'));
```

NUMTODSINTERVAL

Syntax

```
numtodsinterval::=NUMTODSINTERVAL(n,interval_unit)
```

Purpose

`NUMTODSINTERVAL` converts `n` to an INTERVAL DAY TO SECOND literal. The argument `n` can be any NUMBER value or an expression that can be implicitly converted to a NUMBER value. The argument `interval_unit` can be of CHAR, VARCHAR2, NCHAR, or NVARCHAR2 data type. The value for `interval_unit` specifies the unit of `n` and must resolve to one of the following string values:

- 'DAY'
- 'HOUR'
- 'MINUTE'
- 'SECOND'

The parameter `interval_unit` is case insensitive. Leading and trailing values within the parentheses are ignored. By default, the precision of the return is 9.

Example

The following example calculates, for each employee, the number of employees hired by the same manager within the past 100 days from his or her hire date:

```
SELECT manager_id, last_name, hire_date,
       COUNT(*) OVER (PARTITION BY manager_id ORDER BY hire_date
       RANGE NUMTODSINTERVAL(100, 'day') PRECEDING) AS t_count
   FROM employees;

MANAGER_ID LAST_NAME                 HIRE_DATE    T_COUNT
---------- ---------                 ---------    -------
      100  Kochhar                   21-SEP-89          1
      100  De Haan                   13-JAN-93          1
      100  Raphaely                  07-DEC-94          1
      100  Kaufling                  01-MAY-95          1
      100  Hartstein                 17-FEB-96          1
. . .
      149  Grant                     24-MAY-99          1
      149  Johnson                   04-JUN-00          1
      210  Goyal                     17-AUG-97          1
      205  Gietz                     07-JUN-94          1
           King                      17-JUN-87          1
```

NUMTOYMINTERVAL

Syntax

```
numtoyminterval::=NUMTOYMINTERVAL(n,interval_unit)
```

Purpose

`NUMTOYMINTERVAL` converts `n` to an INTERVAL YEAR TO MONTH literal. The argument `n` can be any NUMBER value or an expression that can be implicitly converted to a NUMBER value. The argument `interval_unit` can be of CHAR, VARCHAR2, NCHAR, or NVARCHAR2 data type. The value for `interval_unit` specifies the unit of `n` and must resolve to one of the following string values:

- 'DAY'
- 'HOUR'
- 'MINUTE'
- 'SECOND'

The parameter `interval_unit` is case insensitive. Leading and trailing values within the parentheses are ignored. By default, the precision of the return is 9.

Example

The following example calculates, for each employee, the total salary of employees hired in the past one year from his or her hire date.

```
SELECT last_name, hire_date, salary, SUM(salary)
       OVER (ORDER BY hire_date
       RANGE NUMTOYMINTERVAL(1,'year') PRECEDING) AS t_sal
   FROM employees;

LAST_NAME                  HIRE_DATE     SALARY      T_SAL
---------                  ---------     ------      -----
King                       17-JUN-87      24000      24000
Whalen                     17-SEP-87       4400      28400
Kochhar                    21-SEP-89      17000      17000
. . .
Markle                     08-MAR-00       2200     112400
Ande                       24-MAR-00       6400     106500
Banda                      21-APR-00       6200     109400
Kumar                      21-APR-00       6100     109400
```

RAWTOHEX

Syntax

```
rawtohex::=RAWTOHEX(raw)
```

Purpose

`RAWTOHEX` converts `raw` to a character value containing its hexadecimal equivalent. The argument must be RAW data type. You can specify a BLOB argument for this function if it is called from within a PL/SQL block.

Example
The following hypothetical example returns the hexadecimal equivalent of a RAW column value:

```
SELECT RAWTOHEX(raw_column) "Graphics"
   FROM grpahics;

Graphics
--------
7D
```

RAWTONHEX

Syntax
```
rawtonhex::=RAWTONHEX(raw)
```

Purpose
`RAWTONHEX` converts `raw` to an NVARCHAR2 character value containing its hexadecimal equivalent.

Example
The following hypothetical example returns the hexadecimal equivalent of a RAW column value:

```
SELECT RAWTONHEX(raw_column),
     DUMP ( RAWTONHEX (raw_column) ) "DUMP"
   FROM graphics;

RAWTONHEX(RA)           DUMP
----------------------- ------------------------------
7D                      Typ=1 Len=4: 0,55,0,68
```

SCN_TO_TIMESTAMP

Syntax
```
scn_to_timestamp::=SCN_TO_TIMESTAMP(number)
```

Purpose
`SCN_TO_TIMESTAMP` takes as an argument a number that evaluates to a system change number (SCN), and returns the approximate timestamp associated with that SCN. The returned value is of TIMESTAMP data type. This function is useful when you want to know the timestamp associated with an SCN. For example, it can be used in conjunction with the ORA_ROWSCN pseudocolumn to associate a timestamp with the most recent change to a row.

Example
The following example uses the ORA_ROWSCN pseudocolumn to determine the system change number of the last update to a row and uses SCN_TO_TIMESTAMP to convert that SCN to a timestamp:

```
SELECT SCN_TO_TIMESTAMP(ORA_ROWSCN) FROM employees
       WHERE employee_id=188;
```

You could use such a query to convert a system change number to a timestamp for use in an Oracle Flashback Query:

```
SELECT salary FROM employees WHERE employee_id = 188;

    SALARY
----------
      3800

UPDATE employees SET salary = salary*10 WHERE employee_id = 188;
COMMIT;

SELECT salary FROM employees WHERE employee_id = 188;

    SALARY
----------
      3800

SELECT SCN_TO_TIMESTAMP(ORA_ROWSCN) FROM employees
       WHERE employee_id=188;

SCN_TO_TIMESTAMP(ORA_ROWSCN)
---------------------------------------------------------
28-AUG-03 01.58.01.000000000 PM

FLASHBACK TABLE employees TO TIMESTAMP
   TO_TIMESTAMP('28-AUG-03 01.00.00.000000000 PM');

SELECT salary FROM employees WHERE employee_id = 188;
    SALARY
----------
      3800
```

TIMESTAMP_TO_SCN

Syntax

```
timestamp_to_scn::=TIMESTAMP_TO_SCN(timestamp)
```

Purpose

`TIMESTAMP_TO_SCN` takes as an argument a timestamp value and returns the approximate system change number (SCN) associated with that timestamp. The returned value is of data type NUMBER. This function is useful any time you want to know the SCN associated with a particular timestamp.

Example

The following example inserts a row into the oe.orders table and then uses TIMESTAMP_TO_SCN to determine the system change number of the insert operation. (The actual SCN returned will differ on each system.)

```
INSERT INTO orders (order_id, order_date, customer_id, order_total)
   VALUES (5000, SYSTIMESTAMP, 188, 2345);

COMMIT;

SELECT TIMESTAMP_TO_SCN(order_date) FROM orders
   WHERE order_id = 5000;
```

```
TIMESTAMP_TO_SCN(ORDER_DATE)
----------------------------
                      574100
```

TO_BINARY_DOUBLE

Syntax

```
to_binary_double::=TO_BINARY_DOUBLE(expr, fmt, nlsparam)
```

Purpose

`TO_BINARY_DOUBLE` returns a double-precision floating-point number. The parameter `expr` can be a character string or a numeric value of type NUMBER, BINARY_FLOAT, or BINARY_DOUBLE. If `expr` is BINARY_DOUBLE, then the function returns `expr`.

The arguments `fmt` and `nlsparam` are optional and are valid only if `expr` is a character string. They serve the same purpose as for the TO_CHAR (number) function. The case-insensitive string '`INF`' is converted to positive infinity. The case-insensitive string '`-INF`' is converted to negative identity. The case-insensitive string '`NaN`' is converted to `NaN` (not a number).

You cannot use a floating-point number format element (F, f, D, or d) in a character string expr. Also, conversions from character strings or NUMBER to BINARY_ DOUBLE can be inexact, because the NUMBER and character types use decimal precision to represent the numeric value, and BINARY_DOUBLE uses binary precision. Conversions from BINARY_FLOAT to BINARY_DOUBLE are exact.

Example

The examples that follow are based on a table with three columns, each with a different numeric data type:

```
CREATE TABLE float_point_demo
  (dec_num NUMBER(10,2), bin_double BINARY_DOUBLE, bin_float BINARY_FLOAT);

INSERT INTO float_point_demo VALUES (1234.56,1234.56,1234.56);

SELECT * FROM float_point_demo;

   DEC_NUM BIN_DOUBLE  BIN_FLOAT
---------- ---------- ----------
   1234.56 1.235E+003 1.235E+003
```

The following example converts a value of data type NUMBER to a value of data type BINARY_DOUBLE:

```
SELECT dec_num, TO_BINARY_DOUBLE(dec_num)
  FROM float_point_demo;

   DEC_NUM TO_BINARY_DOUBLE(DEC_NUM)
---------- -------------------------
   1234.56                1.235E+003
```

The following example compares extracted dump information from the `dec_num` and `bin_double` columns:

```
SELECT DUMP(dec_num) "Decimal",
     DUMP(bin_double) "Double"
```

```
   FROM float_point_demo;

Decimal                     Double
--------------------------- --------------------------------------------
Typ=2 Len=4: 194,13,35,57   Typ=101 Len=8: 192,147,74,61,112,163,215,10
```

TO_BINARY_FLOAT

Syntax

```
to_binary_float::=TO_BINARY_FLOAT(expr, fmt, nlsparam)
```

Purpose

`TO_BINARY_FLOAT` returns a single-precision floating-point number. The parameter `expr` can be a character string or a numeric value of type NUMBER, BINARY_FLOAT, or BINARY_DOUBLE. If `expr` is BINARY_FLOAT, then the function returns `expr`.

The arguments `fmt` and `nlsparam` are optional and are valid only if `expr` is a character string. They serve the same purpose as for the TO_CHAR (number) function. The case-insensitive string '`INF`' is converted to positive infinity. The case-insensitive string '`-INF`' is converted to negative identity. The case-insensitive string '`NaN`' is converted to `NaN` (not a number).

You cannot use a floating-point number format element (F, f, D, or d) in a character string expr. Also, conversions from character strings or NUMBER to BINARY_FLOAT can be inexact, because the NUMBER and character types use decimal precision to represent the numeric value, and BINARY_FLOAT uses binary precision. Conversions from BINARY_DOUBLE to BINARY_FLOAT are inexact if the BINARY_DOUBLE value uses more bits of precision than supported by the BINARY_FLOAT.

Example

Using table `float_point_demo` created for TO_BINARY_DOUBLE, the following example converts a value of data type NUMBER to a value of data type BINARY_FLOAT:

```
SELECT dec_num, TO_BINARY_FLOAT(dec_num)
  FROM float_point_demo;

   DEC_NUM TO_BINARY_FLOAT(DEC_NUM)
---------- ------------------------
   1234.56              1.235E+003
```

TO_CHAR

Syntax

```
to_char_date::=TO_CHAR(attribute, fmt, nlsparam)
```

Purpose

`TO_CHAR` converts `attribute` of DATE or NUMBER data type to a value of VARCHAR2 data type in the format specified by the format `fmt`. If you omit `fmt`, a date is converted to a VARCHAR2 value in the default date format and a number is converted to a VARCHAR2 value exactly long enough to hold its significant digits.

If `attribute` is a date, the `nlsparam` specifies the language in which month and day names and abbreviations are returned. This argument can have this form: `'NLS_DATE_LANGUAGE = language'` If you omit `nlsparam`, this function uses the default date language for your session.

If `attribute` is a number, the `nlsparam` specifies these characters that are returned by number format elements:

- Decimal character
- Group separator
- Local currency symbol
- International currency symbol

This argument can have the following form:

```
'NLS_NUMERIC_CHARACTERS = ''dg''
 NLS_CURRENCY = ''text''
 NLS_ISO_CURRENCY = territory '
```

The characters `d` and `g` represent the decimal character and group separator, respectively. They must be different single-byte characters. Within the quoted string, you must use two single quotation marks around the parameter values. Ten characters are available for the currency symbol.

If you omit `nlsparam` or any one of the parameters, this function uses the default parameter values for your session.

Example

The following example applies various conversions on the system date in the database:

```
SELECT TO_CHAR(sysdate) no_fmt FROM DUAL;

NO_FMT
---------
26-MAR-02

SELECT TO_CHAR(sysdate, 'dd-mm-yyyy') fmted FROM DUAL;

FMTED
----------
26-03-2002
```

In this example, the output is blank padded to the left of the currency symbol.

```
SELECT TO_CHAR(-10000,'L99G999D99MI') "Amount" FROM DUAL;

Amount
--------------
$10,000.00-
SELECT TO_CHAR(-10000,'L99G999D99MI'
     'NLS_NUMERIC_CHARACTERS = '',.''
     NLS_CURRENCY = ''AusDollars'' ') "Amount"
   FROM DUAL;

Amount
-------------------
AusDollars10.000,00-
```

TO_CLOB

Syntax

```
to_clob::=TO_CLOB(attribute)
```

Purpose

`TO_CLOB` converts NCLOB values in a LOB column or other character strings to CLOB values. `char` can be any of the data types CHAR, VARCHAR2, NCHAR, NVARCHAR2, CLOB, or NCLOB. Oracle Database executes this function by converting the underlying LOB data from the national character set to the database character set.

Example

The following statement converts NCLOB data from the sample pm.print_media table to CLOB and inserts it into a CLOB column, replacing existing data in that column.

```
UPDATE PRINT_MEDIA SET AD_FINALTEXT = TO_CLOB (AD_FLTEXTN);
```

TO_DATE

Syntax

```
to_date::=TO_DATE(attribute, fmt, nlsparam)
```

Purpose

`TO_DATE` converts `attribute` of CHAR or VARCHAR2 data type to a value of data type DATE. The `fmt` is a date format specifying the format of `attribute`. If you omit `fmt`, `attribute` must be in the default date format. If `fmt` is 'J', for Julian, then `attribute` must be an integer. The `nlsparam` has the same purpose in this function as in the `TO_CHAR` function for date conversion.

Do not use the `TO_DATE` function with a DATE value for the `attribute` argument. The first two digits of the returned DATE value can differ from the original `attribute`, depending on `fmt` or the default date format.

Example

The following example converts character strings into dates:

```
SELECT TO_DATE('January 15, 1989, 11:00 A.M.','Month dd, YYYY, HH:MI A.M.',
     'NLS_DATE_LANGUAGE = American')
   FROM DUAL;

TO_DATE
---------
15-JAN-89
```

TO_DSINTERVAL

Syntax

```
to_dsinterval::=TO_DSINTERVAL(char, nlsparam)
```

Purpose

TO_DSINTERVAL converts a character string of CHAR, VARCHAR2, NCHAR, or NVARCHAR2 data type to an INTERVAL DAY TO SECOND value. The argument char represents the character string to be converted. The only valid nlsparam you can specify in this function is NLS_NUMERIC_CHARACTERS. nlsparam can have the form: NLS_NUMERIC_CHARACTERS = "dg", where d represents the decimal character and g represents the group separator.

Example

The following example selects from the employees table the employees who had worked for the company for at least 100 days on January 1, 1990:

```
SELECT employee_id, last_name
   FROM employees
   WHERE hire_date + TO_DSINTERVAL('100 10:00:00') <= DATE '1990-01-01';

EMPLOYEE_ID LAST_NAME
----------- ---------------
        100 King
        101 Kochhar
        200 Whalen
```

TO_MULTI_BYTE

Syntax

```
to_multi_byte::=TO_MULTI_BYTE(attribute)
```

Purpose

TO_MULTI_BYTE returns attribute with all of its single-byte characters converted to their corresponding multibyte characters; attribute can be of data type CHAR or VARCHAR2. The value returned is in the same data type as attribute. Any single-byte characters in attribute that have no multibyte equivalents appear in the output string as single-byte characters.

This function is useful only if your database character set contains both single-byte and multibyte characters.

Example

The following example illustrates converting from a single byte 'A' to a multi byte:

```
'A' in UTF8:
SELECT dump(TO_MULTI_BYTE('A')) FROM DUAL;

DUMP(TO_MULTI_BYTE('A'))
------------------------
Typ=1 Len=3: 239,188,161
```

TO_NCHAR

Syntax

```
to_nchar::=TO_NCHAR(c, fmt, nlsparam)
```

Purpose

`TO_NCHAR` converts a character string, CLOB, or NCLOB value from the database character set to the national character set. This function is equivalent to the TRANSLATE ... USING function with a USING clause in the national character set.

Example

The following example converts NCLOB data from the pm.print_media table to the national character set:

```
SELECT TO_NCHAR(ad_fltextn) FROM print_media
   WHERE product_id = 3106;

TO_NCHAR(AD_FLTEXTN)
------------------------------------------------------------------------
TIGER2  Tastaturen...weltweit fuehrend in Computer-Ergonomie.
TIGER2 3106 Tastatur
Product Nummer: 3106
Nur 39 EURO!
Die Tastatur KB 101/CH-DE ist eine Standard PC/AT Tastatur mit 102 Tasten. Tasta
turbelegung: Schweizerdeutsch.
. NEU: Kommt mit ergonomischer Schaumstoffunterlage.
. Extraflache und ergonimisch-geknickte Versionen verfugbar auf Anfrage.
. Lieferbar in Elfenbein, Rot oder Schwarz.
```

TO_NCLOB

Syntax

```
to_nclob::=TO_NCLOB(char)
```

Purpose

`TO_NCLOB` converts CLOB values in a LOB column or other character strings to NCLOB values. `char` can be any of the data types CHAR, VARCHAR2, NCHAR, NVARCHAR2, CLOB, or NCLOB. Oracle Database implements this function by converting the character set of `char` from the database character set to the national character set.

Example

The following example inserts some character data into an NCLOB column of the pm.print_media table by first converting the data with the TO_NCLOB function:

```
INSERT INTO print_media (product_id, ad_id, ad_fltextn)
   VALUES (3502, 31001, TO_NCLOB('Placeholder for new product description'));
```

TO_NUMBER

Syntax

```
to_number::=TO_NUMBER(attribute, fmt, nlsparam)
```

Purpose

`TO_NUMBER` converts `attribute` to a value of CHAR or VARCHAR2 data type containing a number in the format specified by the optional format `fmt`, to a value of NUMBER data type.

Examples

The following example converts character string data into a number:

```
UPDATE employees
   SET salary = salary + TO_NUMBER('100.00', '9G999D99')
   WHERE last_name = 'Perkins';
```

The `nlsparam` string in this function has the same purpose as it does in the `TO_CHAR` function for number conversions.

```
SELECT TO_NUMBER('-AusDollars100','L9G999D99',
' NLS_NUMERIC_CHARACTERS = '',.''
NLS_CURRENCY = ''AusDollars''
') "Amount"
FROM DUAL;

Amount
----------
-100
```

TO_SINGLE_BYTE

Syntax

`to_single_byte::=TO_SINGLE_BYTE(attribute)`

Purpose

`TO_SINGLE_BYTE` returns `attribute` with all of its multibyte characters converted to their corresponding single-byte characters; `attribute` can be of data type CHAR or VARCHAR2. The value returned is in the same data type as `attribute`. Any multibyte characters in `attribute` that have no single-byte equivalents appear in the output as multibyte characters.

This function is useful only if your database character set contains both single-byte and multibyte characters.

Example

The following example illustrates going from a multibyte 'A' in UTF8 to a single byte ASCII 'A':

```
SELECT TO_SINGLE_BYTE( CHR(15711393)) FROM DUAL;

T
-
A
```

TO_TIMESTAMP

Syntax

`to_timestamp::=TO_TIMESTAMP(char, fnt, nlsparam)`

Purpose

`TO_TIMESTAMP` converts `char` of data type CHAR, VARCHAR2, NCHAR, or NVARCHAR2 to a value of TIMESTAMP data type. The optional `fmt` specifies the format of `char`. If you omit `fmt`, then `char` must be in the default format of the TIMESTAMP data type, which is determined by the NLS_TIMESTAMP_FORMAT initialization parameter. The optional `nlsparam` argument has the same purpose in this function as in the TO_CHAR function for date conversion.

This function does not support CLOB data directly. However, CLOBs can be passed in as arguments through implicit data conversion.

Example

The following example converts a character string to a timestamp. The character string is not in the default TIMESTAMP format, so the format mask must be specified:

```
SELECT TO_TIMESTAMP ('10-Sep-02 14:10:10.123000', 'DD-Mon-RR HH24:MI:SS.FF')
   FROM DUAL;

TO_TIMESTAMP('10-SEP-0214:10:10.123000','DD-MON-RRHH24:MI:SS.FF')
---------------------------------------------------------------------------
10-SEP-02 02.10.10.123000000 PM
```

TO_TIMESTAMP_TZ

Syntax

```
to_timestamp_tz::=TO_TIMESTAMP_TZ(char, fmt, nlsparam)
```

Purpose

`TO_TIMESTAMP_TZ` converts `char` of data type CHAR, VARCHAR2, NCHAR, or NVARCHAR2 to a value of TIMESTAMP WITH TIME ZONE data type. The optional `fmt` specifies the format of `char`. If you omit `fmt`, then `char` must be in the default format of the TIMESTAMP WITH TIME ZONE data type. The optional `nlsparam` has the same purpose in this function as in the TO_CHAR function for date conversion.

Note: This function does not convert character strings to TIMESTAMP WITH LOCAL TIME ZONE.

Example

The following example converts a character string to a value of TIMESTAMP WITH TIME ZONE:

```
SELECT TO_TIMESTAMP_TZ('1999-12-01 11:00:00 -8:00','YYYY-MM-DD HH:MI:SS TZH:TZM')
   FROM DUAL;

TO_TIMESTAMP_TZ('1999-12-0111:00:00-08:00','YYYY-MM-DDHH:MI:SSTZH:TZM')
--------------------------------------------------------------------
01-DEC-99 11.00.00.000000000 AM -08:00
```

The following example casts a null column in a UNION operation as TIMESTAMP WITH LOCAL TIME ZONE using the sample tables oe.order_items and `oe.orders`:

```
SELECT order_id, line_item_id, CAST(NULL AS TIMESTAMP WITH LOCAL TIME ZONE)
       order_date
   FROM order_items
   UNION
SELECT order_id, to_number(null), order_date
   FROM orders;

  ORDER_ID LINE_ITEM_ID ORDER_DATE
---------- ------------ -----------------------------------
```

```
      2354           1
      2354           2
      2354           3
      2354           4
      2354           5
      2354           6
      2354           7
      2354           8
      2354           9
      2354          10
      2354          11
      2354          12
      2354          13
      2354          14-JUL-00 05.18.23.234567 PM
      2355           1
      2355           2
...
```

TO_YMINTERVAL

Syntax

`to_yminterval::=TO_YMINTERVAL(char)`

Purpose

`TO_YMINTERVAL` converts a character string, represented by `char`, of data type CHAR, VARCHAR2, NCHAR, or NVARCHAR2 to an INTERVAL YEAR TO MONTH type.

Example

The following example calculates for each employee in the sample hr.employees table a date one year two months after the hire date:

```
SELECT hire_date, hire_date + TO_YMINTERVAL('01-02') "14 months"
   FROM employees;

HIRE_DATE 14 months
--------- ---------
17-JUN-87 17-AUG-88
21-SEP-89 21-NOV-90
13-JAN-93 13-MAR-94
03-JAN-90 03-MAR-91
21-MAY-91 21-JUL-92
. . .
```

UNISTR

Syntax

`unistr::=UNISTR(string)`

Purpose

`UNISTR` takes as its argument a text string, represented by `string`, and returns it in the national character set. The national character set of the database can be either AL16UTF16 or UTF8. UNISTR provides support for Unicode string literals by letting you specify the Unicode encoding value of characters in the string. This is useful, for example, for inserting data into NCHAR columns.

The Unicode encoding value has the form '\xxxx' where 'xxxx' is the hexadecimal value of a character in UCS-2 encoding format. To include the backslash in the string itself, precede it with another backslash (\\). For portability and data preservation, Oracle recommends that in the UNISTR string argument you specify only ASCII characters and the Unicode encoding values.

Example

The following example passes both ASCII characters and Unicode encoding values to the UNISTR function, which returns the string in the national character set:

```
SELECT UNISTR('abc\00e5\00f1\00f6') FROM DUAL;

UNISTR
------
abcåñö
```

Date Transformations

Date transformations provide Warehouse Builder users with functionality to perform transformations on date attributes. These transformations are ordered and the custom functions provided with Warehouse Builder are all in the format `WB_<function name>`.

The date transformations provided with Warehouse Builder are:

- ADD_MONTHS on page 2-52
- CURRENT_DATE on page 2-53
- DBTIMEZONE on page 2-53
- FROM_TZ on page 2-53
- LAST_DAY on page 2-54
- MONTHS_BETWEEN on page 2-54
- NEW_TIME on page 2-55
- NEXT_DAY on page 2-56
- ROUND (date) on page 2-56
- SESSIONTIMEZONE on page 2-56
- SYSDATE on page 2-57
- SYSTIMESTAMP on page 2-57
- SYS_EXTRACT_UTC on page 2-58
- TRUNC (date) on page 2-58
- WB_CAL_MONTH_NAME on page 2-58
- WB_CAL_MONTH_OF_YEAR on page 2-59
- WB_CAL_MONTH_SHORT_NAME on page 2-59
- WB_CAL_QTR on page 2-60
- WB_CAL_WEEK_OF_YEAR on page 2-60

- WB_CAL_YEAR on page 2-61
- WB_CAL_YEAR_NAME on page 2-61
- WB_DATE_FROM_JULIAN on page 2-62
- WB_DAY_NAME on page 2-62
- WB_DAY_OF_MONTH on page 2-63
- WB_DAY_OF_WEEK on page 2-63
- WB_DAY_OF_YEAR on page 2-64
- WB_DAY_SHORT_NAME on page 2-64
- WB_DECADE on page 2-65
- WB_HOUR12 on page 2-65
- WB_HOUR12MI_SS on page 2-66
- WB_HOUR24 on page 2-67
- WB_HOUR24MI_SS on page 2-67
- WB_IS_DATE on page 2-68
- WB_JULIAN_FROM_DATE on page 2-68
- WB_MI_SS on page 2-69
- WB_WEEK_OF_MONTH on page 2-69

ADD_MONTHS

Syntax

```
add_months::=ADD_MONTHS(attribute, n)
```

Purpose

`ADD_MONTHS` returns the date in the `attribute` plus `n` months. The argument `n` can be any integer. This will typically be added from an `attribute` or from a constant.

If the date in `attribute` is the last day of the month or if the resulting month has fewer days than the day component of `attribute`, then the result is the last day of the resulting month. Otherwise, the result has the same day component as `attribute`.

Example

The following example returns the month after the `hire_date` in the sample table employees:

```
SELECT TO_CHAR(ADD_MONTHS(hire_date,1), 'DD-MON-YYYY') "Next month"
   FROM employees
   WHERE last_name = 'Baer';

Next Month
-----------
07-JUL-1994
```

CURRENT_DATE

Syntax

```
current_date::=CURRENT_DATE()
```

Purpose

CURRENT_DATE returns the current date in the session time zone, in a value in the Gregorian calendar of data type DATE.

Example

The following example illustrates that CURRENT_DATE is sensitive to the session time zone:

```
ALTER SESSION SET TIME_ZONE = '-5:0';
ALTER SESSION SET NLS_DATE_FORMAT = 'DD-MON-YYYY HH24:MI:SS';
SELECT SESSIONTIMEZONE, CURRENT_DATE FROM DUAL;

SESSIONTIMEZONE CURRENT_DATE
--------------- --------------------
-05:00          29-MAY-2000 13:14:03

ALTER SESSION SET TIME_ZONE = '-8:0';
SELECT SESSIONTIMEZONE, CURRENT_DATE FROM DUAL;

SESSIONTIMEZONE CURRENT_DATE
--------------- --------------------
-08:00          29-MAY-2000 10:14:33
```

DBTIMEZONE

Syntax

```
dbtimezone::+DBTIMEZONE()
```

Purpose

DBTIMEZONE returns the value of the database time zone. The return type is a time zone offset (a character type in the format '[+|-]TZH:TZM') or a time zone region name, depending on how the user specified the database time zone value in the most recent CREATE DATABASE or ALTER DATABASE statement.

Example

The following example assumes that the database time zone is set to UTC time zone:

```
SELECT DBTIMEZONE FROM DUAL;

DBTIME
------
+00:00
```

FROM_TZ

Syntax

```
from_tz::=FROM_TZ(timestamp_value, time_zone_value)
```

Purpose

FROM_TZ converts a timestamp value, represented by timestamp_value, and a time zone, represented by time_zone_value, to a TIMESTAMP WITH TIME ZONE value. time_zone_value is a character string in the format 'TZH:TZM' or a character expression that returns a string in TZR with optional TZD format.

Example

The following example returns a timestamp value to TIMESTAMP WITH TIME ZONE:

```
SELECT FROM_TZ(TIMESTAMP '2000-03-28 08:00:00', '3:00')
   FROM DUAL;

FROM_TZ(TIMESTAMP'2000-03-2808:00:00','3:00')
---------------------------------------------------------------
28-MAR-00 08.00.00 AM +03:00
```

LAST_DAY

Syntax

```
last_day::=LAST_DAY(attribute)
```

Purpose

LAST_DAY returns the date of the last day of the month that contains the date in attribute.

Examples

The following statement determines how many days are left in the current month.

```
SELECT SYSDATE, LAST_DAY(SYSDATE) "Last", LAST_DAY(SYSDATE) - SYSDATE "Days Left"
   FROM DUAL;

SYSDATE   Last Days Left
--------- --------- ----------
23-OCT-97 31-OCT-97 8
```

MONTHS_BETWEEN

Syntax

```
months_between::=MONTHS_BETWEEN(attribute1, attribute2)
```

Purpose

MONTHS_BETWEEN returns the number of months between dates in attribute1 and attribute2. If attribute1 is later than attribute2, the result is positive; if earlier, then the result is negative.

If attribute1 and attribute2 are either the same day of the month or both last days of months, the result is always an integer. Otherwise, Oracle calculates the fractional portion of the result-based on a 31-day month and considers the difference in time components attribute1 and attribute2.

Example

The following example calculates the months between two dates:

```
SELECT MONTHS_BETWEEN(TO_DATE('02-02-1995','MM-DD-YYYY'),
     TO_DATE('01-01-1995','MM-DD-YYYY') ) "Months"
   FROM DUAL;

Months
----------
1.03225806
```

NEW_TIME

Syntax

```
new_time::=NEW_TIME(attribute, zone1, zone2)
```

Purpose

`NEW_TIME` returns the date and time in time zone `zone2` when date and time in time zone `zone1` are the value in `attribute`. Before using this function, you must set the `NLS_DATE_FORMAT` parameter to display 24-hour time.

The arguments `zone1` and `zone2` can be any of these text strings:

- AST, ADT: Atlantic Standard or Daylight Time
- BST, BDT: Bering Standard or Daylight Time
- CST, CDT: Central Standard or Daylight Time
- CST, EDT: Eastern Standard or Daylight Time
- GMT: Greenwich Mean Time
- HST, HDT: Alaska-Hawaii Standard Time or Daylight Time.
- MST, MDT: Mountain Standard or Daylight Time
- NST: Newfoundland Standard Time
- PST, PDT: Pacific Standard or Daylight Time
- YST, YDT: Yukon Standard or Daylight Time

Example

The following example returns an Atlantic Standard time, given the Pacific Standard time equivalent:

```
ALTER SESSION SET NLS_DATE_FORMAT = 'DD-MON-YYYY HH24:MI:SS';
SELECT NEW_TIME (TO_DATE ('11-10-99 01:23:45', 'MM-DD-YY HH24:MI:SS'),
      'AST', 'PST') "New Date and Time"
   FROM DUAL;

New Date and Time
--------------------
09-NOV-1999 21:23:45
```

NEXT_DAY

Syntax

```
next_day::=NEXT_DAY(attribute1, attribute2)
```

Purpose

`NEXT_DAY` returns the date of the first weekday named by the string in `attribute2` that is later than the date in `attribute1`. The argument `attribute2` must be a day of the week in the date language of your session, either the full name or the abbreviation. The minimum number of letters required is the number of letters in the abbreviated version. Any characters immediately following the valid abbreviation are ignored. The return value has the same hours, minutes, and seconds component as the argument `attribute1`.

Example

This example returns the date of the next Tuesday after February 2, 2001:

```
SELECT NEXT_DAY('02-FEB-2001','TUESDAY') "NEXT DAY"
   FROM DUAL;

NEXT DAY
-----------
06-FEB-2001
```

ROUND (date)

Syntax

```
round_date::=ROUND(attribute, fmt)
```

Purpose

`ROUND` returns the date in `attribute` rounded to the unit specified by the format model `fmt`. If you omit `fmt`, date is rounded to the nearest day.

Example

The following example rounds a date to the first day of the following year:

```
SELECT ROUND (TO_DATE ('27-OCT-00'),'YEAR') "New Year"
   FROM DUAL;

New Year
---------
01-JAN-01
```

SESSIONTIMEZONE

Syntax

```
sessiontimezone::=SESSIONTIMEZONE()
```

Purpose

`SESSIONTIMEZONE` returns the time zone of the current session. The return type is a time zone offset (a character type in the format '[+|]TZH:TZM') or a time zone region name, depending on how the user specified the session time zone value in the most recent ALTER SESSION statement. You can set the default client session time zone using the `ORA_SDTZ` environment variable.

Example

The following example returns the time zone of the current session:

```
SELECT SESSIONTIMEZONE FROM DUAL;

SESSION
-------
-08:00
```

SYSDATE

Syntax

```
sysdate::=SYSDATE
```

Purpose

`SYSDATE` returns the current date and time. The data type of the returned value is `DATE`. The function requires no arguments. In distributed SQL statements, this function returns the date and time on your local database. You cannot use this function in the condition of a `CHECK` constraint.

Example

The following example returns the current date and time:

```
SELECT TO_CHAR(SYSDATE, 'MM-DD-YYYY HH24:MI:SS')"NOW" FROM DUAL;

NOW
-------------------
04-13-2001 09:45:51
```

SYSTIMESTAMP

Syntax

```
systimestamp::=SYSTIMESTAMP()
```

Purpose

`SYSTIMESTAMP` returns the system date, including fractional seconds and time zone, of the system on which the database resides. The return type is TIMESTAMP WITH TIME ZONE.

Example

The following example returns the system timestamp:

```
SELECT SYSTIMESTAMP FROM DUAL;

SYSTIMESTAMP
------------------------------------------------------------------
28-MAR-00 12.38.55.538741 PM -08:00
```

The following example shows how to explicitly specify fractional seconds:

```
SELECT TO_CHAR(SYSTIMESTAMP, 'SSSSS.FF') FROM DUAL;

TO_CHAR(SYSTIME
---------------
55615.449255
```

SYS_EXTRACT_UTC

Syntax

`sys_extract_utc::=SYS_EXTRACT_UTC(datetime_with_timezone)`

Purpose

SYS_EXTRACT_UTC extracts the UTC (Coordinated Universal Time—formerly Greenwich Mean Time) from a datetime value with time zone offset or time zone region name.

Example

The following example extracts the UTC from a specified datetime:

```
SELECT SYS_EXTRACT_UTC(TIMESTAMP '2000-03-28 11:30:00.00 -08:00')
   FROM DUAL;

SYS_EXTRACT_UTC(TIMESTAMP'2000-03-2811:30:00.00-08:00')
-----------------------------------------------------------------
28-MAR-00 07.30.00 PM
```

TRUNC (date)

Syntax

`trunc_date::=TRUNC(attribute, fmt)`

Purpose

`TRUNC` returns `attribute` with the time portion of the day truncated to the unit specified by the format model `fmt`. If you omit `fmt`, date is truncated to the nearest day.

Example

The following example truncates a date:

```
SELECT TRUNC(TO_DATE('27-OCT-92','DD-MON-YY'), 'YEAR') "New Year"
   FROM DUAL;

New Year
---------
01-JAN-92
```

WB_CAL_MONTH_NAME

Syntax

`WB_CAL_MONTH_NAME(attribute)`

Purpose
The function call returns the full-length name of the month for the date specified in `attribute`.

Example
The following example shows the return value on the `sysdate` and on a specified date string:

```
SELECT WB_CAL_MONTH_NAME(sysdate)
   FROM DUAL;

WB_CAL_MONTH_NAME(SYSDATE)
----------------------------
March

SELECT WB_CAL_MONTH_NAME('26-MAR-2002')
   FROM DUAL;

WB_CAL_MONTH_NAME('26-MAR-2002')
----------------------------------
March
```

WB_CAL_MONTH_OF_YEAR

Syntax
```
WB_CAL_MONTH_OF_YEAR(attribute)
```

Purpose
`WB_CAL_MONTH_OF_YEAR` returns the month (1-12) of the year for date in `attribute`.

Example
The following example shows the return value on the `sysdate` and on a specified date string:

```
SELECT WB_CAL_MONTH_OF_YEAR(sysdate) month
   FROM DUAL;

     MONTH
----------
         3

SELECT WB_CAL_MONTH_OF_YEAR('26-MAR-2002') month
FROM DUAL;

     MONTH
----------
         3
```

WB_CAL_MONTH_SHORT_NAME

Syntax
```
WB_CAL_MONTH_SHORT_NAME(attribute)
```

Purpose

`WB_CAL_MONTH_SHORT_NAME` returns the short name of the month (for example 'Jan') for date in `attribute`.

Example

The following example shows the return value on the `sysdate` and on a specified date string:

```
SELECT WB_CAL_MONTH_SHORT_NAME (sysdate) month
FROM DUAL;

MONTH
---------
Mar

SELECT WB_CAL_MONTH_SHORT_NAME ('26-MAR-2002') month
FROM DUAL;

MONTH
---------
Mar
```

WB_CAL_QTR

Syntax

```
WB_CAL_QTR(attribute)
```

Purpose

`WB_CAL_QTR` returns the quarter of the Gregorian calendar year (for example Jan - March = 1) for the date in `attribute`.

Example

The following example shows the return value on the `sysdate` and on a specified date string:

```
SELECT WB_CAL_QTR (sysdate) quarter
FROM DUAL;

   QUARTER
----------
         1

SELECT WB_CAL_QTR ('26-MAR-2002') quarter
FROM DUAL;

   QUARTER
----------
         1
```

WB_CAL_WEEK_OF_YEAR

Syntax

```
WB_CAL_WEEK_OF_YEAR(attribute)
```

Purpose

WB_CAL_WEEK_OF_YEAR returns the week of the year (1-53) for the date in attribute.

Example

The following example shows the return value on the sysdate and on a specified date string:

```
SELECT WB_CAL_WEEK_OF_YEAR (sysdate) w_of_y
FROM DUAL;

    W_OF_Y
----------
        13

SELECT WB_CAL_WEEK_OF_YEAR ('26-MAR-2002') w_of_y
FROM DUAL;

    W_OF_Y
----------
        13
```

WB_CAL_YEAR

Syntax

```
WB_CAL_YEAR(attribute)
```

Purpose

WB_CAL_YEAR returns the numerical year component for the date in attribute.

Example

The following example shows the return value on the sysdate and on a specified date string:

```
SELECT WB_CAL_YEAR (sysdate) year
FROM DUAL;

      YEAR
----------
      2002

SELECT WB_CAL_YEAR ('26-MAR-2002') w_of_y
FROM DUAL;

      YEAR
----------
      2002
```

WB_CAL_YEAR_NAME

Syntax

```
WH_CAL_YEAR_NAME(attribute)
```

Purpose

`WB_CAL_YEAR_NAME` returns the spelled out name of the year for the date in `attribute`.

Example

The following example shows the return value on the `sysdate` and on a specified date string:

```
select WB_CAL_YEAR_NAME (sysdate) name
from dual;

NAME
----------------------------------------------
Two Thousand Two

select WB_CAL_YEAR_NAME ('26-MAR-2001') name
from dual;

NAME
----------------------------------------------
Two Thousand One
```

WB_DATE_FROM_JULIAN

Syntax

```
WB_DATE_FROM_JULIAN(attribute)
```

Purpose

`WB_DATE_FROM_JULIAN` converts Julian date `attribute` to a regular date.

Example

The following example shows the return value on a specified Julian date:

```
select to_char(WB_DATE_FROM_JULIAN(3217345),'dd-mon-yyyy') JDate
from dual;

JDATE
-----------
08-sep-4096
```

WB_DAY_NAME

Syntax

```
WB_DAY_NAME(attribute)
```

Purpose

`WB_DAY_NAME` returns the full name of the day for the date in `attribute`.

Example

The following example shows the return value on the `sysdate` and on a specified date string:

```
select WB_DAY_NAME (sysdate) name
from dual;
```

```
NAME
-------------------------------------------
Thursday

select WB_DAY_NAME ('26-MAR-2002') name
from dual;

NAME
-------------------------------------------
Tuesday
```

WB_DAY_OF_MONTH

Syntax

```
WB_DAY_OF_MONTH(attribute)
```

Purpose

`WB_DAY_OF_MONTH` returns the day number within the month for the date in `attribute`.

Example

The following example shows the return value on the `sysdate` and on a specified date string:

```
select WB_DAY_OF_MONTH (sysdate) num
from dual;

       NUM
----------
        28

select WB_DAY_OF_MONTH ('26-MAR-2002') num
from dual

       NUM
----------
        26
```

WB_DAY_OF_WEEK

Syntax

```
WB_DAY_OF_WEEK(attribute)
```

Purpose

`WB_DAY_OF_WEEK` returns the day number within the week for date `attribute` based on the database calendar.

Example

The following example shows the return value on the `sysdate` and on a specified date string:

```
select WB_DAY_OF_WEEK (sysdate) num
```

```
from dual;

       NUM
----------
         5

select WB_DAY_OF_WEEK ('26-MAR-2002') num
from dual;


       NUM
----------
         3
```

WB_DAY_OF_YEAR

Syntax

```
WB_DAY_OF_YEAR(attribute)
```

Purpose

`WB_DAY_OF_YEAR` returns the day number within the year for the date `attribute`.

Example

The following example shows the return value on the `sysdate` and on a specified date string:

```
select WB_DAY_OF_YEAR (sysdate) num
from dual;

       NUM
----------
        87

select WB_DAY_OF_YEAR ('26-MAR-2002') num
from dual;

       NUM
----------
        85
```

WB_DAY_SHORT_NAME

Syntax

```
WB_DAY_SHORT_NAME(attribute)
```

Purpose

`WB_DAY_SHORT_NAME` returns the three letter abbreviation or name for the date `attribute`.

Example

The following example shows the return value on the `sysdate` and on a specified date string:

```
select WB_DAY_SHORT_NAME  (sysdate) abbr
from dual;

ABBR
-------------------------------------
Thu

select WB_DAY_SHORT_NAME  ('26-MAR-2002') abbr
from dual;

NUM
-------------------------------------
Tue
```

WB_DECADE

Syntax

```
WB_DECADE(attribute)
```

Purpose

`WB_DECADE` returns the decade number within the century for the date `attribute`.

Example

The following example shows the return value on the `sysdate` and on a specified date string:

```
select WB_DECADE  (sysdate) dcd
from dual;

       DCD
----------
         2

select WB_DECADE  ('26-MAR-2002') DCD
from dual;

       DCD
----------
         2
```

WB_HOUR12

Syntax

```
WB_HOUR12(attribute)
```

Purpose

`WB_HOUR12` returns the hour (in a 12-hour setting) component of the date corresponding to `attribute`.

Example

The following example shows the return value on the `sysdate` and on a specified date string:

```
select WB_HOUR12 (sysdate) h12
from dual;

       H12
----------
         9

select WB_HOUR12 ('26-MAR-2002') h12
from dual;

       H12
----------
        12
```

Note: For a date not including the timestamp (in the second example), Oracle uses the 12:00 (midnight) timestamp and therefore returns 12 in this case.

WB_HOUR12MI_SS

Syntax

```
WB_HOUR12MI_SS(attribute)
```

Purpose

`WB_HOUR12MI_SS` returns the timestamp in `attribute` formatted to HH12:MI:SS.

Example

The following example shows the return value on the `sysdate` and on a specified date string:

```
select WB_HOUR12MI_SS (sysdate) h12miss
from dual;

H12MISS
-------------------------------------
09:08:52

select WB_HOUR12MI_SS ('26-MAR-2002') h12miss
from dual;

H12MISS
-------------------------------------
12:00:00
```

Note: For a date not including the timestamp (in the second example), Oracle uses the 12:00 (midnight) timestamp and therefore returns 12 in this case.

WB_HOUR24

Syntax

```
WB_HOUR24(attribute)
```

Purpose

`WB_HOUR24` returns the hour (in a 24-hour setting) component of date corresponding to `attribute`.

Example

The following example shows the return value on the `sysdate` and on a specified date string:

```
select WB_HOUR24 (sysdate) h24
from dual;

       H24
----------
         9

select WB_HOUR24 ('26-MAR-2002') h24
from dual;

       H24
----------
         0
```

> **Note:** For a date not including the timestamp (in the second example), Oracle uses the 00:00:00 timestamp and therefore returns the timestamp in this case.

WB_HOUR24MI_SS

Syntax

```
WB_HOUR24MI_SS(attribute)
```

Purpose

`WB_HOUR24MI_SS` returns the timestamp in `attribute` formatted to HH24:MI:SS.

Example

The following example shows the return value on the `sysdate` and on a specified date string:

```
select WB_HOUR24MI_SS (sysdate) h24miss
from dual;

H24MISS
------------------------------------
09:11:42

select WB_HOUR24MI_SS ('26-MAR-2002') h24miss
from dual;
```

```
H24MISS
------------------------------------
00:00:00
```

Note: For a date not including the timestamp (in the second example), Oracle uses the 00:00:00 timestamp and therefore returns the timestamp in this case.

WB_IS_DATE

Syntax

```
WB_IS_DATE(attribute, fmt)
```

Purpose

To check whether `attribute` contains a valid date. The function returns a Boolean value which is set to true if `attribute` contains a valid date. `Fmt` is an optional date format. If `fmt` is omitted, the date format of your database session is used.

You can use this function when you validate your data before loading it into a table. This way the value can be transformed before it reaches the table and causes an error.

Example

`WB_IS_DATE` returns true in PL/SQL if `attribute` contains a valid date.

WB_JULIAN_FROM_DATE

Syntax

```
WB_JULIAN_FROM_DATE(attribute)
```

Purpose

`WB_JULIAN_FROM_DATE` returns the Julian date of date corresponding to `attribute`.

Example

The following example shows the return value on the `sysdate` and on a specified date string:

```
select WB_JULIAN_FROM_DATE (sysdate) jdate
from dual;

     JDATE
----------
   2452362

select WB_JULIAN_FROM_DATE ('26-MAR-2002') jdate
from dual;

     JDATE
----------
   2452360
```

WB_MI_SS

Syntax

```
WB_MI_SS(attribute)
```

Purpose

`WB_MI_SS` returns the minutes and seconds of the time component in the date corresponding to `attribute`.

Example

The following example shows the return value on the `sysdate` and on a specified date string:

```
select WB_MI_SS (sysdate) mi_ss
from dual;

MI_SS
-------------------------------------------
33:23

select WB_MI_SS ('26-MAR-2002') mi_ss
from dual;

MI_SS
-------------------------------------------
00:00
```

Note: For a date not including the timestamp (in the second example), Oracle uses the 00:00:00 timestamp and therefore returns the timestamp in this case.

WB_WEEK_OF_MONTH

Syntax

```
WB_WEEK_OF_MONTH(attribute)
```

Purpose

`WB_WEEK_OF_MONTH` returns the week number within the calendar month for the date corresponding to `attribute`.

Example

The following example shows the return value on the `sysdate` and on a specified date string:

```
select WB_WEEK_OF_MONTH (sysdate) w_of_m
from dual;

    W_OF_M
----------
         4

select WB_WEEK_OF_MONTH ('26-MAR-2002') w_of_m
from dual;
```

```
    W_OF_M
----------
         4
```

Number Transformations

Number transformations provide Warehouse Builder users with functionality to perform transformations on numeric values. The custom functions provided with Warehouse Builder are prefixed with `WB_`.

All numerical transformations provided with Warehouse Builder are:

- ABS on page 2-71
- ACOS on page 2-71
- ASIN on page 2-71
- ATAN on page 2-72
- ATAN2 on page 2-72
- BITAND on page 2-72
- CEIL on page 2-73
- COS on page 2-74
- COSH on page 2-74
- EXP on page 2-74
- FLOOR on page 2-75
- LN on page 2-75
- LOG on page 2-75
- MOD on page 2-76
- NANVL on page 2-76
- POWER on page 2-77
- REMAINDER on page 2-77
- ROUND (number) on page 2-78
- SIGN on page 2-78
- SIN on page 2-79
- SINH on page 2-79
- SQRT on page 2-79
- TAN on page 2-80
- TANH on page 2-80
- TRUNC (number) on page 2-80
- WB_LOOKUP_NUM (on a number) on page 2-81
- WB_LOOKUP_NUM (on a varchar2) on page 2-82
- WB_IS_NUMBER on page 2-82

- WIDTH_BUCKET on page 2-83

ABS

Syntax

```
abs::=ABS(attribute)
```

Purpose

ABS returns the absolute value of `attribute`.

Example

The following example returns the absolute value of -15:

```
SELECT ABS(-15) "Absolute" FROM DUAL;
Absolute
----------
15
```

ACOS

Syntax

```
acos::= ACOS(attribute)
```

Purpose

`ACOS` returns the arc cosine of `attribute`. The argument `attribute` must be in the range of -1 to 1, and the function returns values in the range of 0 to pi, expressed in radians.

Example

The following example returns the arc cosine of .3:

```
SELECT ACOS(.3) "Arc_Cosine" FROM DUAL;

Arc_Cosine
----------
1.26610367
```

ASIN

Syntax

```
asin::=ASIN(attribute)
```

Purpose

`ASIN` returns the arc sine of `attribute`. The argument `attribute` must be in the range of -1 to 1, and the function returns values in the range of -pi/2 to pi/2, expressed in radians.

Example

The following example returns the arc cosine of .3:

```
SELECT ACOS(.3) "Arc_Sine" FROM DUAL;
```

```
Arc_Sine
----------
.304692654
```

ATAN

Syntax

```
atan::=ATAN(attribute)
```

Purpose

`ATAN` returns the arc tangent of `attribute`. The argument `attribute` can be in an unbounded range, and the function returns values in the range of -pi/2 to pi/2, expressed in radians.

Example

The following example returns the arc tangent of .3:

```
SELECT ATAN(.3) "Arc_Tangent" FROM DUAL;

Arc_Tangent
----------
.291456794
```

ATAN2

Syntax

```
atan2::=ATAN2(attribute1, attribute2)
```

Purpose

`ATAN2` returns the arc tangent of `attribute1` and `attribute2`. The argument attribute1 can be in an unbounded range, and the function returns values in the range of -pi to pi, depending on the signs of attribute1 and attribute2, and are expressed in radians. `ATAN2(attribute1,attribute2)` is the same as `ATAN2(attribute1/attribute2).`

Example

The following example returns the arc tangent of .3 and .2:

```
SELECT ATAN2(.3,.2) "Arc_Tangent2" FROM DUAL;

Arc_Tangent2
------------
.982793723
```

BITAND

Syntax

```
bitand::=BITAND(expr1,expr2)
```

Purpose

`BITAND` computes an AND operation on the bits of `expr1` and `expr2`, both of which must resolve to nonnegative integers, and returns an integer. This function is commonly used with the DECODE function, as illustrated in the example that follows.

Both arguments can be any numeric data type, or any nonnumeric data type that can be implicitly converted to NUMBER. The function returns NUMBER.

Example

The following represents each `order_status` in the sample table `oe.orders` by individual bits. (The example specifies options that can total only 7, so rows with order_status greater than 7 are eliminated.)

```
SELECT order_id, customer_id,
  DECODE(BITAND(order_status, 1), 1, 'Warehouse', 'PostOffice')
      Location,
  DECODE(BITAND(order_status, 2), 2, 'Ground', 'Air') Method,
  DECODE(BITAND(order_status, 4), 4, 'Insured', 'Certified') Receipt
  FROM orders
  WHERE order_status < 8;

  ORDER_ID CUSTOMER_ID LOCATION   MET RECEIPT
---------- ----------- ---------- --- ---------
      2458         101 Postoffice Air Certified
      2397         102 Warehouse  Air Certified
      2454         103 Warehouse  Air Certified
      2354         104 Postoffice Air Certified
      2358         105 Postoffice G   Certified
      2381         106 Warehouse  G   Certified
      2440         107 Warehouse  G   Certified
      2357         108 Warehouse  Air Insured
      2394         109 Warehouse  Air Insured
      2435         144 Postoffice G   Insured
      2455         145 Warehouse  G   Insured
. . .
```

CEIL

Syntax

```
ceil::=CEIL(attribute)
```

Purpose

`CEIL` returns smallest integer greater than or equal to `attribute`.

Example

The following example returns the smallest integer greater than or equal to 15.7:

```
 SELECT CEIL(15.7) "Ceiling" FROM DUAL;
 Ceiling
----------
16
```

COS

Syntax

```
cos::=COS(attribute)
```

Purpose

`COS` returns the cosine of `attribute` (an angle expressed in degrees).

Example

The following example returns the cosine of 180 degrees:

```
SELECT COS(180 * 3.14159265359/180) "Cosine" FROM DUAL;

Cosine
------
     -1
```

COSH

Syntax

```
cosh::=COSH(attribute)
```

Purpose

`COSH` returns the hyperbolic cosine of `attribute`.

Example

The following example returns the hyperbolic cosine of 0:

```
SELECT COSH(0) "Hyperbolic Cosine" FROM DUAL;

Hyperbolic Cosine
-----------------
                1
```

EXP

Syntax

```
exp::=EXP(attribute)
```

Purpose

`EXP` returns e raised to the nth power represented in `attribute`, where e = 2.71828183...

Example

The following example returns e to the 4th power:

```
SELECT EXP(4) "e to the 4th power" FROM DUAL;

e to the 4th power
------------------
54.59815
```

FLOOR

Syntax

```
floor::=FLOOR(attribute)
```

Purpose

`FLOOR` returns the largest integer equal to or less than the numerical value in `attribute`.

Example

The following example returns the largest integer equal to or less than 15.7:

```
SELECT FLOOR(15.7) "Floor" FROM DUAL;

Floor
----------
15
```

LN

Syntax

```
ln::=LN(attribute)
```

Purpose

`LN` returns the natural logarithm of `attribute`, where `attribute` is greater than 0.

Example

The following example returns the natural logarithm of 95:

```
SELECT LN(95) "Natural Logarithm" FROM DUAL;

Natural Logarithm
-----------------
4.55387689
```

LOG

Syntax

```
log::=LOG(attribute1, attribute2)
```

Purpose

`LOG` returns the logarithm, base `attribute1` of `attribute2`. The base `attribute1` can be any positive number other than 0 or 1 and attribute2 can be any positive number.

Example

The following example returns the logarithm of 100:

```
SELECT LOG(10,100) "Log base 10 of 100" FROM DUAL;
```

```
Log base 10 of 100
------------------
              2
```

MOD

Syntax

```
mod::=MOD(attribute1, attribute2)
```

Purpose

`MOD` returns the remainder of `attribute1` divided by `attribute2`. It returns `attribute1` if `attribute2` is 0.

Example

The following example returns the remainder of 11 divided by 4:

```
 SELECT MOD(11,4) "Modulus" FROM DUAL;
 Modulus
----------
3
```

NANVL

Syntax

```
nanvl::=NANVL(m,n)
```

Purpose

The NANVL function is useful only for floating-point numbers of type BINARY_FLOAT or BINARY_DOUBLE. It instructs Oracle Database to return an alternative value `n` if the input value `m` is NaN (not a number). If `m` is not NaN, then Oracle returns `m`. This function is useful for mapping NaN values to NULL.

This function takes as arguments any numeric data type or any nonnumeric data type that can be implicitly converted to a numeric data type. Oracle determines the argument with the highest numeric precedence, implicitly converts the remaining arguments to that data type, and returns that data type.

Example

Using table `float_point_demo` created for TO_BINARY_DOUBLE, insert a second entry into the table:

```
INSERT INTO float_point_demo
  VALUES (0,'NaN','NaN');

SELECT * FROM float_point_demo;

   DEC_NUM BIN_DOUBLE  BIN_FLOAT
---------- ---------- ----------
   1234.56 1.235E+003 1.235E+003
         0        Nan        Nan
```

The following example returns bin_float if it is not a number. Otherwise, 0 is returned.

```
SELECT bin_float, NANVL(bin_float,0)
  FROM float_point_demo;

 BIN_FLOAT NANVL(BIN_FLOAT,0)
---------- ------------------
1.235E+003         1.235E+003
       Nan                  0
```

POWER

Syntax

```
power::=POWER(attribute1, attribute2)
```

Purpose

POWER returns attribute1 raised to the nth power represented in attribute2. The base attribute1 and the exponent in attribute2 can be any numbers, but if attribute1 is negative, then attribute2 must be an integer.

Example

The following example returns three squared:

```
SELECT POWER(3,2) "Raised" FROM DUAL;
Raised
----------
9
```

REMAINDER

Syntax

```
remainder::=REMAINDER(m,n)
```

Purpose

REMAINDER returns the remainder of m divided by n. This function takes as arguments any numeric data type or any nonnumeric data type that can be implicitly converted to a numeric data type. Oracle determines the argument with the highest numeric precedence, implicitly converts the remaining arguments to that data type, and returns that data type.

If n = 0 or m = infinity, then Oracle returns an error if the arguments are of type NUMBER and NaN if the arguments are BINARY_FLOAT or BINARY_DOUBLE. If n != 0, then the remainder is m - (n*N) where N is the integer nearest m/n. If m is a floating-point number, and if the remainder is 0, then the sign of the remainder is the sign of m. Remainders of 0 are unsigned for NUMBER values.

The MOD function is similar to REMAINDER except that it uses FLOOR in its formula, whereas REMAINDER uses ROUND.

Example

Using table `float_point_demo` created for TO_BINARY_DOUBLE, the following example divides two floating-point numbers and returns the remainder of that operation:

```
SELECT bin_float, bin_double, REMAINDER(bin_float, bin_double)
  FROM float_point_demo;

 BIN_FLOAT BIN_DOUBLE REMAINDER(BIN_FLOAT,BIN_DOUBLE)
---------- ---------- -------------------------------
1.235E+003 1.235E+003                      5.859E-005
```

ROUND (number)

Syntax

```
round_number::=ROUND(attribute1, attribute2)
```

Purpose

`ROUND` returns `attribute1` rounded to `attribute2` places right of the decimal point. If `attribute2` is omitted, `attribute1` is rounded to 0 places. Additionally, `attribute2` can be negative to round off digits left of the decimal point and `attribute2` must be an integer.

Examples

The following example rounds a number to one decimal point:

```
SELECT ROUND(15.193,1) "Round" FROM DUAL;

Round
----------
15.2
```

The following example rounds a number one digit to the left of the decimal point:

```
SELECT ROUND(15.193,-1) "Round" FROM DUAL;
Round
----------
20
```

SIGN

Syntax

```
sign::=SIGN(attribute)
```

Purpose

If `attribute` < 0, `SIGN` returns -1. If `attribute` = 0, the function returns 0. If `attribute` > 0, `SIGN` returns 1. This can be used in validation of measures where only positive numbers are expected.

Example

The following example indicates that the function's argument (-15) is <0:

```
SELECT SIGN(-15) "Sign" FROM DUAL;
 Sign
----------
-1
```

SIN

Syntax

```
sin::=SIN(attribute)
```

Purpose

`SIN` returns the sine of `attribute` (expressed as an angle)

Example

The following example returns the sine of 30 degrees:

```
SELECT SIN(30 * 3.14159265359/180) "Sine of 30 degrees" FROM DUAL;

Sine of 30 degrees
------------------
                .5
```

SINH

Syntax

```
sinh::=SINH(attribute)
```

Purpose

`SINH` returns the hyperbolic sine of attribute.

Example

The following example returns the hyperbolic sine of 1:

```
SELECT SINH(1) "Hyperbolic Sine of 1" FROM DUAL;

Hyperbolic Sine of 1
--------------------
          1.17520119
```

SQRT

Syntax

```
sqrt::=SQRT(attribute)
```

Purpose

`SQRT` returns square root of `attribute`. The value in `attribute` cannot be negative. `SQRT` returns a "real" result.

Example
The following example returns the square root of 26:

```
SELECT SQRT(26) "Square root" FROM DUAL;

Square root
-----------
5.09901951
```

TAN

Syntax
```
tan::=TAN(attrin=bute)
```

Purpose
`TAN` returns the tangent of `attribute` (an angle expressed in radians).

Example
The following example returns the tangent of 135 degrees:

```
SELECT TAN(135 * 3.14159265359/180) "Tangent of 135 degrees" FROM DUAL;

Tangent of 135 degrees
----------------------
                    -1
```

TANH

Syntax
```
tanh::=TANH(attribute)
```

Purpose
`TANH` returns the hyperbolic tangent of `attribute`.

Example
The following example returns the hyperbolic tangent of 5:

```
SELECT TANH(5) "Hyperbolic tangent of 5" FROM DUAL;

Hyperbolic tangent of 5
-----------------------
             .462117157
```

TRUNC (number)

Syntax
```
trunc_number::=TRUNC(attribute, m)
```

Purpose

TRUNC returns `attribute` truncated to `m` decimal places. If `m` is omitted, `attribute` is truncated to 0 places. `m` can be negative to truncate (make zero) `m` digits left of the decimal point.

Example

The following example truncates numbers:

```
SELECT TRUNC(15.79,1) "Truncate"
FROM DUAL;
Truncate
----------
15.7
 SELECT TRUNC(15.79,-1) "Truncate"
FROM DUAL;
 Truncate
----------
10
```

WB_LOOKUP_NUM (on a number)

Syntax

```
 WB_LOOKUP_NUM (table_name
, column_name
, key_column_name
, key_value
)
```

where `table_name` is the name of the table to perform the lookup on; `column_name` is the name of the NUMBER column that will be returned, for instance, the result of the lookup; `key_column_name` is the name of the NUMBER column used as the key to match on in the lookup table; `key_value` is the value of the key column, for example, the value mapped into the `key_column_name` with which the match will be done.

Purpose

To perform a key look up that returns a NUMBER value from a database table using a NUMBER column as the matching key.

Example

Consider the following table as a lookup table LKP1:

```
KEYCOLUMN  TYPE_NO  TYPE
10         100123   Car
20         100124   Bike
```

Using this package with the following call:

```
WB_LOOKUP_CHAR('LKP1'
, 'TYPE_NO'
, 'KEYCOLUMN'
, 20
)
```

returns the value of 100124 as output of this transformation. This output is then processed in the mapping as the result of an inline function call.

Note: This function is a row-based key lookup. Set-based lookups are supported when you use the lookup operator.

WB_LOOKUP_NUM (on a varchar2)

Syntax:

```
WB_LOOKUP_CHAR(table_name
, column_name
, key_column_name
, key_value
)
```

where `table_name` is the name of the table to perform the lookup on; `column_name` is the name of the NUMBER column that will be returned (such as the result of the lookup); `key_column_name` is the name of the NUMBER column used as the key to match on in the lookup table; `key_value` is the value of the key column, such as the value mapped into the `key_column_name` with which the match will be done.

Purpose:

To perform a key lookup which returns a NUMBER value from a database table using a VARCHAR2 column as the matching key.

Example

Consider the following table as a lookup table `LKP1`:

```
KEYCOLUMN  TYPE_NO  TYPE
ACV        100123   Car
ACP        100124   Bike
```

Using this package with the following call:

```
WB_LOOKUP_CHAR ('LKP1'
, 'TYPE'
, 'KEYCOLUMN'
, 'ACP'
)
```

returns the value of `100124` as output of this transformation. This output is then processed in the mapping as the result of an inline function call.

Note: This function is a row-based key lookup. Set-based lookups are supported when you use the Key lookup operator described in the Oracle Warehouse Builder User's Guide.

WB_IS_NUMBER

Syntax

```
 WB_IS_NUMBER(attibute, fmt)
```

Purpose

To check whether attribute contains a valid number. The function returns a Boolean value, which is set to true if attribute contains a valid number. Fmt is an optional number format. If fmt is omitted, the number format of your session is used.

You can use this function when you validate the data before loading it into a table. This way the value can be transformed before it reaches the table and causes an error.

Example

WB_IS_NUMBER returns true in PL/SQL if attribute contains a valid number.

WIDTH_BUCKET

Syntax

```
width_bucket::=WIDTH_BUCKET(expr,min_value,max_value,num_buckets)
```

Purpose

For a given expression, WIDTH_BUCKET returns the bucket number into which the value of this expression would fall after being evaluated. WIDTH_BUCKET lets you construct equiwidth histograms, in which the histogram range is divided into intervals that have identical size. Ideally each bucket is a closed-open interval of the real number line. For example, a bucket can be assigned to scores between 10.00 and 19.999... to indicate that 10 is included in the interval and 20 is excluded. This is sometimes denoted as (10, 20).

The argument expr represents the expression for which the histogram is being created. This expression must evaluate to a numeric or datetime value or to a value that can be implicitly converted to a numeric or datetime value. If expr evaluates to null, then the expression returns null. min_value and max_value are expressions that resolve to the end points of the acceptable range for expr. Both of these expressions must also evaluate to numeric or datetime values, and neither can evaluate to null. num_buckets is an expression that resolves to a constant indicating the number of buckets. This expression must evaluate to a positive integer.

When needed, Oracle Database creates an underflow bucket numbered 0 and an overflow bucket numbered num_buckets+1. These buckets handle values less than min_value and more than max_value and are helpful in checking the reasonableness of endpoints.

Example

The following example creates a ten-bucket histogram on the credit_limit column for customers in Switzerland in the sample table oe.customers and returns the bucket number ("Credit Group") for each customer. Customers with credit limits greater than the maximum value are assigned to the overflow bucket, 11:

```
SELECT customer_id, cust_last_name, credit_limit,
   WIDTH_BUCKET(credit_limit, 100, 5000, 10) "Credit Group"
   FROM customers WHERE nls_territory = 'SWITZERLAND'
   ORDER BY "Credit Group";

CUSTOMER_ID CUST_LAST_NAME       CREDIT_LIMIT Credit Group
----------- -------------------- ------------ ------------
        825 Dreyfuss                      500            1
        826 Barkin                        500            1
        853 Palin                         400            1
        827 Siegel                        500            1
```

```
843 Oates                         700           2
844 Julius                        700           2
835 Eastwood                     1200           3
840 Elliott                      1400           3
842 Stern                        1400           3
841 Boyer                        1400           3
837 Stanton                      1200           3
836 Berenger                     1200           3
848 Olmos                        1800           4
849 Kaurusmdki                   1800           4
828 Minnelli                     2300           5
829 Hunter                       2300           5
852 Tanner                       2300           5
851 Brown                        2300           5
850 Finney                       2300           5
830 Dutt                         3500           7
831 Bel Geddes                   3500           7
832 Spacek                       3500           7
838 Nicholson                    3500           7
839 Johnson                      3500           7
833 Moranis                      3500           7
834 Idle                         3500           7
845 Fawcett                      5000          11
846 Brando                       5000          11
847 Streep                       5000          11
```

OLAP Transformations

OLAP transformations enable Warehouse Builder users to load data stored in relational dimensions and cubes into an analytic workspace.

The OLAP transformations provided by Warehouse Builder are:

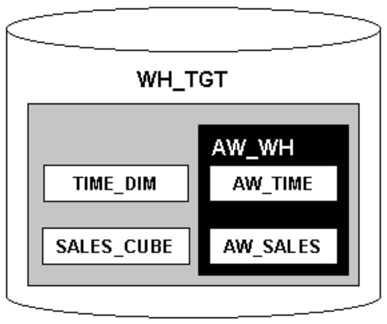
- WB_OLAP_AW_PRECOMPUTE on page 2-85
- WB_OLAP_LOAD_CUBE on page 2-86
- WB_OLAP_LOAD_DIMENSION on page 2-86
- WB_OLAP_LOAD_DIMENSION_GENUK on page 2-87

The `WB_OLAP_LOAD_CUBE`, `WB_OLAP_LOAD_DIMENSION`, and `WB_OLAP_LOAD_DIMENSION_GENUK` transformations are used for cube cloning in Warehouse Builder. Use these OLAP transformations only if your database version is Oracle Database 9*i* or Oracle Database 10*g* Release 1. Starting with Oracle 10*g* Release 2, you can directly deploy dimensions and cubes into an analytic workspace.

The `WB_OLAP_AW_PRECOMPUTE` only works with the Oracle Warehouse Builder 10*g* Release 2.

The examples used to explain these OLAP transformations are based on the scenario depicted in Figure 2–1.

Figure 2–1 Example of OLAP Transformations



The relational dimension `TIME_DIM` and the relational cube `SALES_CUBE` are stored in the schema `WH_TGT`. The analytic workspace `AW_WH`, into which the dimension and cube are loaded, is also created in the `WH_TGT` schema.

WB_OLAP_AW_PRECOMPUTE

Syntax

```
WBWB_OLAP_AW_PRECOMPUTE(p_aw_name, p_cube_name, p_measure_name, p_allow_parallel_
solve, p_max_job_queues_allocated)
```

where `p_aw_name` is the name of the AW where cube is deployed, `p_cube_name` is the name of the cube to solve, `p_measure_name` is the optional name of a specific measure to solve (if no measure is specified, then all measures will be solved), `p_allow_parallel_solve` is the boolean to indicate parallelization of solve based on partitioning (performance related parameter), `p_max_job_queues_allocated` is the number of DBMS jobs to execute in parallel (default value is 0). If 5 is defined and there are 20 partitions then a pool of 5 DBMS jobs will be used to perform the data load.

There is a subtle different between parallel and non-parallel solving. With non-parallel solve, the solve happens synchronously, so when the API call is completed the solve is complete. Parallel solve executes asynchronously, the API call will return with a job id of the job launched. The job will control parallel solving using the max job queues parameter to control its processing. The user may then use the job id to query the all_scheduler_* views to check on the status of the activity.

Purpose

WB_OLAP_AW_PRECOMPUTE is used for solving a non-compressed cube (compressed cubes are auto-solved). The load and solve steps can be done independently. By default, the cube map loads data, then solves (precomputes) the cube. You can load data using the map, then perform the solve at a different point of time (since the solve/build time is the costliest operation).

Example

The following example loads data from the relational cubes `MART` and `SALES_CUBE` into a cube called `SALES` and performs a simple solve execution working serially. This example has parameters for parallel solve and max number of job queues. If parallel solve is performed then an ASYNCHRONOUS solve job is launched and the master job ID is returned through the return function.

```
declare
  rslt varchar2(4000);
```

```
begin
…
  rslt :=wb_olap_aw_precompute('MART','SALES_CUBE','SALES');
…
end;
/
```

WB_OLAP_LOAD_CUBE

Syntax

```
wb_olap_load_cube::=WB_OLAP_LOAD_CUBE(olap_aw_owner, olap_aw_name, olap_cube_
owner, olap_cube_name, olap_tgt_cube_name)
```

where `olap_aw_owner` is the name of the database schema that owns the analytic workspace; `olap_aw_name` is the name of the analytic workspace that stores the cube data; `olap_cube_owner` is the name of the database schema that owns the related relational cube; `olap_cube_name` is the name of the relational cube; `olap_tgt_cube_name` is the name of the cube in the analytic workspace.

Purpose

`WB_OLAP_LOAD_CUBE` loads data from the relational cube into the analytic workspace. This allows further analysis of the cube data. This is for loading data in an AW cube from a relational cube which it was cloned from. This is a wrapper around some of the procedures in the DBMS_AWM package for loading a cube.

Example

The following example loads data from the relational cube `SALES_CUBE` into a cube called `AW_SALES` in the `AW_WH` analytic workspace:

```
WB_OLAP_LOAD_CUBE('WH_TGT', 'AW_WH', 'WH_TGT', 'SALES_CUBE', 'AW_SALES')
```

WB_OLAP_LOAD_DIMENSION

Syntax

```
wb_olap_load_dimension::=WB_OLAP_LOAD_DIMENSION(olap_aw_owner, olap_aw_name, olap_
dimension_owner, olap_dimension_name, olap_tgt_dimension_name)
```

where `olap_aw_owner` is the name of the database schema that owns the analytic workspace; `olap_aw_name` is the name of the analytic workspace that stores the dimension data; `olap_dimension_owner` is the name of the database schema in which the related relational dimension is stored; `olap_dimension_name` is the name of the relational dimension; `olap_tgt_dimension_name` is the name of the dimension in the analytic workspace.

Purpose

`WB_OLAP_LOAD_DIMENSION` loads data from the relational dimension into the analytic workspace. This allows further analysis of the dimension data. This is for loading data in an AW dimension from a relational dimension which it was cloned from. This is a wrapper around some of the procedures in the DBMS_AWM package for loading a dimension.

Example

The following example loads the data from the relational dimension `TIME_DIM` into a dimension called `AW_TIME` in the analytic workspace `AW_WH`:

```
WB_OLAP_LOAD_DIMENSION('WH_TGT', 'AW_WH', 'WH_TGT', 'TIME_DIM', 'AW_TIME')
```

WB_OLAP_LOAD_DIMENSION_GENUK

Syntax

```
wb_olap_load_dimension_genuk::=WB_OLAP_LOAD_DIMENSION_GENUK(olap_aw_owner, olap_
aw_name, olap_dimension_owner, olap_dimension_name, olap_tgt_dimension_name)
```

where `olap_aw_owner` is the name of the database schema that owns the analytic workspace; `olap_aw_name` is the name of the analytic workspace that stores the dimension data; `olap_dimension_owner` is the name of the database schema in which the related relational dimension is stored; `olap_dimension_name` is the name of the relational dimension; `olap_tgt_dimension_name` is the name of the dimension in the analytic workspace.

Purpose

`WB_OLAP_LOAD_DIMENSION_GENUK` loads data from the relational dimension into the analytic workspace. Unique dimension identifiers will be generated across all levels. This is for loading data in an AW dimension from a relational dimension which it was cloned from. This is a wrapper around some of the procedures in the DBMS_AWM package for loading a dimension.

If a cube has been cloned and if you select YES for the Generate Surrogate Keys for Dimensions option, then when you want to reload the dimensions, you should use the `WB_OLAP_LOAD_DIMENSION_GENUK` procedure. This procedure generates surrogate identifiers for all levels in the AW, because the AW requires all level identifiers to be unique across all levels of a dimension.

Example

Consider an example in which the dimension `TIME_DIM` has been deployed to the OLAP server by cloning the cube. The parameter generate surrogate keys for Dimension was set to true. To now reload data from the relational dimension `TIME_DIM` into the dimension `AW_TIME` in the analytic workspace `AW_WH`, use the following syntax.

```
WB_OLAP_LOAD_CUBE('WH_TGT', 'AW_WH', 'WH_TGT', 'TIME_DIM', 'AW_TIME')
```

Other Transformations

Other transformations included with Warehouse Builder enable you to perform various functions which are not restricted to certain data types. This section describes those types. Other transformations provided by Warehouse Builder are:

- DEPTH on page 2-88
- DUMP on page 2-88
- EMPTY_BLOB, EMPTY_CLOB on page 2-90
- NLS_CHARSET_DECL_LEN on page 2-90
- NLS_CHARSET_ID on page 2-90

- NLS_CHARSET_NAME on page 2-91
- NULLIF on page 2-91
- NVL on page 2-92
- NVL2 on page 2-93
- ORA_HASH on page 2-93
- PATH on page 2-94
- SYS_CONTEXT on page 2-95
- SYS_GUID on page 2-95
- SYS_TYPEID on page 2-96
- UID on page 2-96
- USER on page 2-97
- USERENV on page 2-97
- VSIZE on page 2-98

DEPTH

Syntax

```
depth::=DEPTH(correlation_integer)
```

Purpose

`DEPTH` is an ancillary function used only with the UNDER_PATH and EQUALS_PATH conditions. It returns the number of levels in the path specified by the UNDER_PATH condition with the same correlation variable. The `correlation_integer` can be any NUMBER integer. Use it to correlate this ancillary function with its primary condition if the statement contains multiple primary conditions. Values less than 1 are treated as 1.

Example

The EQUALS_PATH and UNDER_PATH conditions can take two ancillary functions, DEPTH and PATH. The following example shows the use of both ancillary functions. The example assumes the existence of the XMLSchema `warehouses.xsd`.

```
SELECT PATH(1), DEPTH(2)
   FROM RESOURCE_VIEW
   WHERE UNDER_PATH(res, '/sys/schemas/OE', 1)=1
     AND UNDER_PATH(res, '/sys/schemas/OE', 2)=1;

PATH(1)                             DEPTH(2)
-------------------------------- --------
/www.oracle.com                            1

/www.oracle.com/xwarehouses.xsd            2
```

DUMP

Syntax

```
dump::=DUMP(expr,return_fmt,start_position,length)
```

Purpose

`DUMP` returns a VARCHAR2 value containing the data type code, length in bytes, and internal representation of `expr`. The returned result is always in the database character set. The argument `return_fmt` specifies the format of the return value and can have any of the following values:

- 8 returns result in octal notation.
- 10 returns result in decimal notation.
- 16 returns result in a hexadecimal notation.
- 17 returns result as single characters.

By default, the return value contains no character set information. To retrieve the character set name of `expr`, add 1000 to any of the preceding format values. For example, a `return_fmt` of 1008 returns the result in octal and provides the character set name of `expr`.

The arguments `start_position` and `length` combine to determine which portion of the internal representation to return. The default is to return the entire internal representation in decimal notation. If `expr` is null, then this function returns null.

Note: This function does not support CLOB data directly. However, CLOBs can be passed in as arguments through implicit data conversion.

Example

The following examples show how to extract dump information from a string expression and a column:

```
SELECT DUMP('abc', 1016)
   FROM DUAL;

DUMP('ABC',1016)
------------------------------------------
Typ=96 Len=3 CharacterSet=WE8DEC: 61,62,63


SELECT DUMP(last_name, 8, 3, 2) "OCTAL"
   FROM employees
   WHERE last_name = 'Hunold';

OCTAL
-------------------------------------------------------------------
Typ=1 Len=6: 156,157

SELECT DUMP(last_name, 10, 3, 2) "ASCII"
   FROM employees
   WHERE last_name = 'Hunold';

ASCII
--------------------------------------------------------------------
Typ=1 Len=6: 110,111
```

EMPTY_BLOB, EMPTY_CLOB

Syntax

```
empty_blob::=EMPTY_BLOB()
empty_clob::=EMPTY_CLOB()
```

Purpose

`EMPTY_BLOB` and `EMPTY_CLOB` return an empty LOB locator that can be used to initialize a LOB variable or, in an INSERT or UPDATE statement, to initialize a LOB column or attribute to EMPTY. EMPTY means that the LOB is initialized, but not populated with data. You must initialize a LOB attribute that is part of an object type before you can access and populate it.

Note: You cannot use the locator returned from this function as a parameter to the DBMS_LOB package or the OCI.

Example

The following example initializes the ad_photo column of the sample pm.print_media table to EMPTY:

```
UPDATE print_media SET ad_photo = EMPTY_BLOB();
```

NLS_CHARSET_DECL_LEN

Syntax

```
nls_charset_decl_len::=NLS_CHARSET_DECL_LEN(byte_count,charset_id)
```

Purpose

`NLS_CHARSET_DECL_LEN` returns the declaration width (in number of characters) of an NCHAR column. The `byte_count` argument is the width of the column. The `charset_id` argument is the character set ID of the column.

Example

The following example returns the number of characters that are in a 200-byte column when you are using a multibyte character set:

```
SELECT NLS_CHARSET_DECL_LEN(200, nls_charset_id('ja16eucfixed')) FROM DUAL;

NLS_CHARSET_DECL_LEN(200,NLS_CHARSET_ID('JA16EUCFIXED'))
----------------------------------------------------------
                                                       100
```

NLS_CHARSET_ID

Syntax

```
nls_charset_id::= NLS_CHARSET_ID(text)
```

Purpose

`NLS_CHARSET_ID` returns the character set ID number corresponding to character set name `text`. The `text` argument is a run-time VARCHAR2 value. The `text` value

'CHAR_CS' returns the database character set ID number of the server. The text value 'NCHAR_CS' returns the national character set ID number of the server.

Invalid character set names return null.

Example

The following example returns the character set ID number of a character set:

```
SELECT NLS_CHARSET_ID('ja16euc') FROM DUAL;

NLS_CHARSET_ID('JA16EUC')
--------------------------
                       830
```

NLS_CHARSET_NAME

Syntax

```
nls_charset_name::= NLS_CHARSET_NAME(number)
```

Purpose

NLS_CHARSET_NAME returns the name of the character set corresponding to ID number. The character set name is returned as a VARCHAR2 value in the database character set.

If number is not recognized as a valid character set ID, then this function returns null.

Example

The following example returns the character set corresponding to character set ID number 2:

```
SELECT NLS_CHARSET_NAME(2) FROM DUAL;

NLS_CH
--------
WE8DEC
```

NULLIF

Syntax

```
nullif::=NULLIF(expr1,expr2)
```

Purpose

NULLIF compares expr1 and expr2. If they are equal, then the function returns null. If they are not equal, then the function returns expr1. You cannot specify the literal NULL for expr1.

If both arguments are numeric data types, then Oracle Database determines the argument with the higher numeric precedence, implicitly converts the other argument to that data type, and returns that data type. If the arguments are not numeric, then they must be of the same data type, or Oracle returns an error.

The NULLIF function is logically equivalent to the following CASE expression:

```
CASE WHEN expr1 = expr 2 THEN NULL ELSE expr1 END
```

Example

The following example selects those employees from the sample schema `hr` who have changed jobs since they were hired, as indicated by a `job_id` in the `job_history` table different from the current `job_id` in the `employees` table:

```
SELECT e.last_name, NULLIF(e.job_id, j.job_id) "Old Job ID"
   FROM employees e, job_history j
   WHERE e.employee_id = j.employee_id
   ORDER BY last_name;

LAST_NAME                 Old Job ID
------------------------- ----------
De Haan                   AD_VP
Hartstein                 MK_MAN
Kaufling                  ST_MAN
Kochhar                   AD_VP
Kochhar                   AD_VP
Raphaely                  PU_MAN
Taylor                    SA_REP
Taylor
Whalen                    AD_ASST
Whalen
```

NVL

Syntax

```
nvl::=NVL(attribute1, attribute2)
```

Purpose

If `attribute1` is null, `NVL` returns `attribute2`. If `attribute1` is not null, then `NVL` returns `attribute1`. The arguments `attribute1` and `attribute2` can be any data type. If their data types are different, `attribute2` is converted to the data type of `attribute1` before they are compared. Warehouse Builder provides three variants of `NVL` to support all input values.

The data type of the return value is always the same as the data type of `attribute1`, unless `attribute1` is character data, in which case the return value data type is VARCHAR2, in the character set of `attribute1`.

Example

The following example returns a list of employee names and commissions, substituting "Not Applicable" if the employee receives no commission:

```
SELECT last_name, NVL(TO_CHAR(commission_pct), 'Not Applicable') "COMMISSION"
FROM employees
WHERE last_name LIKE 'B%';

LAST_NAME                 COMMISSION
------------------------- ----------------------------------------
Baer                      Not Applicable
Baida                     Not Applicable
Banda                     .11
Bates                     .16
Bell                      Not Applicable
Bernstein                 .26
Bissot                    Not Applicable
```

```
Bloom                     .21
Bull                      Not Applicable
```

NVL2

Syntax

`nvl2::=NVL2(expr1,expr2,expr3)`

Purpose

`NVL2` lets you determine the value returned by a query based on whether a specified expression is null or not null. If `expr1` is not null, then `NVL2` returns `expr2`. If `expr1` is null, then `NVL2` returns `expr3`. The argument `expr1` can have any data type. The arguments `expr2` and `expr3` can have any data types except LONG.

If the data types of `expr2` and `expr3` are different:

- If `expr2` is character data, then Oracle Database converts `expr3` to the data type of `expr2` before comparing them unless `expr3` is a null constant. In that case, a data type conversion is not necessary. Oracle returns VARCHAR2 in the character set of `expr2`.
- If `expr2` is numeric, then Oracle determines which argument has the highest numeric precedence, implicitly converts the other argument to that data type, and returns that data type.

Example

The following example shows whether the income of some employees is made up of salary plus commission, or just salary, depending on whether the `commission_pct` column of `employees` is null or not.

```
SELECT last_name, salary, NVL2(commission_pct,
   salary + (salary * commission_pct), salary) income
   FROM employees WHERE last_name like 'B%'
   ORDER BY last_name;

LAST_NAME                     SALARY     INCOME
------------------------- ---------- ----------
Baer                           10000      10000
Baida                           2900       2900
Banda                           6200       6882
Bates                           7300       8468
Bell                            4000       4000
Bernstein                       9500      11970
Bissot                          3300       3300
Bloom                          10000      12100
Bull                            4100       4100
```

ORA_HASH

Syntax

`ora_hash::=ORA_HASH(expr,max_bucket,seed_value)`

Purpose

ORA_HASH is a function that computes a hash value for a given expression. This function is useful for operations such as analyzing a subset of data and generating a random sample. The function returns a NUMBER value.

The `expr` argument determines the data for which you want Oracle Database to compute a hash value. There are no restrictions on the type or length of data represented by `expr`, which commonly resolves to a column name. The argument `max_bucket` is optional and it determines the maximum bucket value returned by the hash function. You can specify any value between 0 and 4294967295. The default is 4294967295. The optional `seed_value` argument enables Oracle to produce many different results for the same set of data. Oracle applies the hash function to the combination of `expr` and `seed_value`. You can specify any value between 0 and 4294967295. The default is 0.

Example

The following example creates a hash value for each combination of customer ID and product ID in the ***sh.sales*** table, divides the hash values into a maximum of 100 buckets, and returns the sum of the `amount_sold` values in the first bucket (bucket 0). The third argument (5) provides a seed value for the hash function. You can obtain different hash results for the same query by changing the seed value.

```
SELECT SUM(amount_sold) FROM sales
   WHERE ORA_HASH(CONCAT(cust_id, prod_id), 99, 5) = 0;

SUM(AMOUNT_SOLD)
----------------
            7315
```

The following example retrieves a subset of the data in the `sh.sales` table by specifying 10 buckets (0 to 9) and then returning the data from bucket 1. The expected subset is about 10% of the rows (the sales table has 960 rows):

```
SELECT * FROM sales
   WHERE ORA_HASH(cust_id, 9) = 1;

   PROD_ID    CUST_ID TIME_ID   C   PROMO_ID QUANTITY_SOLD AMOUNT_SOLD
---------- ---------- --------- - ---------- ------------- -----------
      2510       6950 01-FEB-98 S       9999             2          78
      9845       9700 04-FEB-98 C       9999            17         561
      3445      33530 07-FEB-98 T       9999             2         170
. . .
       740      22200 13-NOV-00 S       9999             4         156
      9425       4750 29-NOV-00 I       9999            11         979
      1675      46750 29-NOV-00 S       9999            19        1121
```

PATH

Syntax

```
path::=PATH(correlation_integer)
```

Purpose

`PATH` is an ancillary function used only with the UNDER_PATH and EQUALS_PATH conditions. It returns the relative path that leads to the resource specified in the parent condition. The `correlation_integer` can be any NUMBER integer and is used to correlate this ancillary function with its primary condition. Values less than 1 are treated as 1.

Example

Please refer to the example for DEPTH. This example uses both the ancillary functions of the EQUALS_PATH and UNDER_PATH.

SYS_CONTEXT

Syntax

```
sys_context::=SYS_CONTEXT(namespace,parameter,length)
```

Purpose

`SYS_CONTEXT` returns the value of parameter associated with the context `namespace`. You can use this function in both SQL and PL/SQL statements. For `namespace` and `parameter`, you can specify either a string or an expression that resolves to a string designating a namespace or an attribute. The context `namespace` must already have been created, and the associated `parameter` and its value must also have been set using the `DBMS_SESSION.set_context` procedure. The `namespace` must be a valid SQL identifier. The `parameter` name can be any string. It is not case sensitive, but it cannot exceed 30 bytes in length.

The data type of the return value is VARCHAR2. The default maximum size of the return value is 256 bytes. You can override this default by specifying the optional `length` parameter, which must be a NUMBER or a value that can be implicitly converted to NUMBER. The valid range of values is 1 to 4000 bytes. If you specify an invalid value, then Oracle Database ignores it and uses the default.

Oracle provides a built-in namespace called `USERENV`, which describes the current session. For a description of the predefined parameters of the namespace `USERENV`, refer Table 7-11 of the *Oracle Database SQL Reference*.

Example

The following statement returns the name of the user who logged onto the database:

```
CONNECT OE/OE
SELECT SYS_CONTEXT ('USERENV', 'SESSION_USER')
   FROM DUAL;

SYS_CONTEXT ('USERENV', 'SESSION_USER')
------------------------------------------------------
OE
```

The following hypothetical example returns the group number that was set as the value for the attribute `group_no` in the PL/SQL package that was associated with the context `hr_apps` when `hr_apps` was created:

```
SELECT SYS_CONTEXT ('hr_apps', 'group_no') "User Group"
   FROM DUAL;
```

SYS_GUID

Syntax

```
sys_guid::=SYS_GUID()
```

Purpose

SYS_GUID generates and returns a globally unique identifier (RAW value) made up of 16 bytes. On most platforms, the generated identifier consists of a host identifier, a process or thread identifier of the process or thread invoking the function, and a nonrepeating value (sequence of bytes) for that process or thread.

Example

The following example adds a column to the sample table hr.locations, inserts unique identifiers into each row, and returns the 32-character hexadecimal representation of the 16-byte RAW value of the global unique identifier:

```
ALTER TABLE locations ADD (uid_col RAW(32));

UPDATE locations SET uid_col = SYS_GUID();

SELECT location_id, uid_col FROM locations;

LOCATION_ID UID_COL
----------- ----------------------------------------
       1000 7CD5B7769DF75CEFE034080020825436
       1100 7CD5B7769DF85CEFE034080020825436
       1200 7CD5B7769DF95CEFE034080020825436
       1300 7CD5B7769DFA5CEFE034080020825436
. . .
```

SYS_TYPEID

Syntax

```
sys_typeid::=SYS_TYPEID(object_type_value)
```

Purpose

SYS_TYPEID returns the typeid of the most specific type of the operand. This value is used primarily to identify the type-discriminant column underlying a substitutable column. For example, you can use the value returned by SYS_TYPEID to build an index on the type-discriminant column. You can use the SYS_TYPEID function to create an index on the type-discriminant column of a table.

You can use this function only on object type operands. All final root object types—that is, final types not belonging to a type hierarchy—have a null typeid. Oracle Database assigns to all types belonging to a type hierarchy a unique non-null typeid.

UID

Syntax

```
uid::=UID()
```

Purpose

UID returns an integer that uniquely identifies the session user, such as the user who is logged on when running the session containing the transformation. In a distributed SQL statement, the UID function identifies the user on your local database.

Use this function when logging audit information into a target table to identify the user running the mappings.

Example

The following returns the local database user id logged into this session:

```
SELECT uid FROM dual;

       UID
----------
        55
```

USER

Syntax

```
user::=USER()
```

Purpose

`USER` returns the name of the session user (the user who logged on) with the data type VARCHAR2.

Oracle compares values of this function with blank-padded comparison semantics. In a distributed SQL statement, the `UID` and `USER` functions identify the user on your local database.

Use this function when logging audit information into a target table to identify the user running the mappings.

Example

The following example returns the local database user logged into this session:

```
SELECT user FROM dual;

USER
------------------------------
OWB9I_RUN
```

USERENV

Syntax

```
userenv::=USERENV(parameter)
```

Purpose

Note: `USERENV` is a legacy function that is retained for backward compatibility. Oracle recommends that you use the `SYS_CONTEXT` function with the built-in `USERENV` namespace for current functionality.

`USERENV` returns information about the current session. This information can be useful for writing an application-specific audit trail table or for determining the language-specific characters currently used by your session. You cannot use `USERENV` in the condition of a CHECK constraint. Table 2–1 describes the values for the `parameter` argument. All calls to `USERENV` return VARCHAR2 data except for calls with the SESSIONID, ENTRYID, and COMMITSCN parameters, which return NUMBER.

Table 2–1 Parameters of the USERENV function

Parameter	Return Value
CLIENT_INFO	CLIENT_INFO returns up to 64 bytes of user session information that can be stored by an application using the DBMS_APPLICATION_INFO package. **Caution:** Some commercial applications may be using this context value. Please refer to the applicable documentation for those applications to determine what restrictions they may impose on use of this context area.
ENTRYID	The current audit entry number. The audit entryid sequence is shared between fine-grained audit records and regular audit records. You cannot use this attribute in distributed SQL statements
ISDBA	ISDBA returns 'TRUE' if the user has been authenticated as having DBA privileges either through the operating system or through a password file.
LANG	LANG returns the ISO abbreviation for the language name, a shorter form than the existing 'LANGUAGE' parameter.
LANGUAGE	LANGUAGE returns the language and territory used by the current session along with the database character set in this form: `language_territory.characterset`
SESSIONID	SESSIONID returns the auditing session identifier. You cannot specify this parameter in distributed SQL statements.
TERMINAL	TERMINAL returns the operating system identifier for the terminal of the current session. In distributed SQL statements, this parameter returns the identifier for your local session. In a distributed environment, this parameter is supported only for remote SELECT statements, not for remote INSERT, UPDATE, or DELETE operations.

Example

The following example returns the `LANGUAGE` parameter of the current session:

```
SELECT USERENV('LANGUAGE') "Language" FROM DUAL;

Language
-----------------------------------
AMERICAN_AMERICA.WE8ISO8859P1
```

VSIZE

Syntax

```
vsize::=VSIZE(expr)
```

Purpose

`VSIZE` returns the number of bytes in the internal representation of expr. If expr is null, then this function returns null. This function does not support CLOB data directly. However, CLOBs can be passed in as arguments through implicit data conversion.

Example

The following example returns the number of bytes in the last_name column of the employees in department 10:

```
SELECT last_name, VSIZE (last_name) "BYTES"
  FROM employees
  WHERE department_id = 10;

LAST_NAME            BYTES
--------------- ----------
Whalen                   6
```

Spatial Transformations

Spatial Transformation is an integrated set of functions and procedures that enables spatial data to be stored, accessed, and analyzed quickly and efficiently in an Oracle database. Spatial transformations included with Warehouse Builder are:

- SDO_AGGR_CENTROID on page 2-99
- SDO_AGGR_CONVEXHULL on page 2-100
- SDO_AGGR_MBR on page 2-100
- SDO_AGGR_UNION on page 2-100

SDO_AGGR_CENTROID

Syntax

```
sdo_aggr_centroid::= SDO_AGGR_CENTROID(AggregateGeometry SDOAGGRTYPE)
```

Purpose

SDO_AGGR_CENTROID returns a geometry object that is the centroid (center of gravity) of the specified geometry objects. The behavior of the function depends on whether the geometry objects are all polygons, all points, or a mixture of polygons and points:

- If the geometry objects are all polygons, the centroid of all the objects is returned.
- If the geometry objects are all points, the centroid of all the objects is returned.
- If the geometry objects are a mixture of polygons and points (specifically, if they include at least one polygon and at least one point), any points are ignored, and the centroid of all the polygons is returned.

The result is weighted by the area of each polygon in the geometry objects. If the geometry objects are a mixture of polygons and points, the points are not used in the calculation of the centroid. If the geometry objects are all points, the points have equal weight.

Example

The following example returns the centroid of the geometry objects in the COLA_ MARKETS table.

```
SELECT SDO_AGGR_CENTROID(SDOAGGRTYPE(shape, 0.005))
  FROM cola_markets;
```

```
SDO_AGGR_CENTROID(SDOAGGRTYPE(SHAPE,0.005))(SDO_GTYPE, SDO_SRID, SDO_POINT
--------------------------------------------------------------------------------
SDO_GEOMETRY(2001, NULL, SDO_POINT_TYPE(5.21295938, 5.00744233, NULL), NULL, NULL)
```

SDO_AGGR_CONVEXHULL

Syntax

```
sod_aggr_convexhull::= SDO_AGGR_CONVEXHULL(AggregateGeometry SDOAGGRTYPE)
```

Purpose

SDO_AGGR_CONVEXHULL returns a geometry object that is the convex hull of the specified geometry objects.

Example

The following example returns the convex hull of the geometry objects in the `COLA_MARKETS` table.

```
SELECT SDO_AGGR_CONVEXHULL(SDOAGGRTYPE(shape, 0.005))
  FROM cola_markets;

SDO_AGGR_CONVEXHULL(SDOAGGRTYPE(SHAPE,0.005))(SDO_GTYPE, SDO_SRID, SDO_POI
--------------------------------------------------------------------------------
SDO_GEOMETRY(2003, NULL, NULL, SDO_ELEM_INFO_ARRAY(1, 1003, 1), SDO_ORDINATE_
ARRAY(8, 1, 10, 7, 10, 11, 8, 11, 6, 11, 1, 7, 1, 1, 8, 1))
```

SDO_AGGR_MBR

Syntax

```
sod_aggr_mbr::= SDO_AGGR_MBR(geom SDO_GEOMETRY)
```

Purpose

SDO_AGGR_MBR returns the minimum bounding rectangle (MBR) of the specified geometries, that is, a single rectangle that minimally encloses the geometries.

Example

The following example returns the minimum bounding rectangle of the geometry objects in the `COLA_MARKETS` table.

```
SELECT SDO_AGGR_MBR(shape) FROM cola_markets;

SDO_AGGR_MBR(C.SHAPE)(SDO_GTYPE, SDO_SRID, SDO_POINT(X, Y, Z), SDO_ELEM_INFO, SD
--------------------------------------------------------------------------------
SDO_GEOMETRY(2003, NULL, NULL, SDO_ELEM_INFO_ARRAY(1, 1003, 3), SDO_ORDINATE_
ARRAY(1, 1, 10, 11))
```

SDO_AGGR_UNION

Syntax

```
SDO_AGGR_UNION(
        AggregateGeometry SDOAGGRTYPE

        ) RETURN SDO_GEOMETRY;
```

Purpose

SDO_AGGR_UNION returns a geometry object that is the topological union (OR operation) of the specified geometry objects.

Example

The following example returns the union of the first three geometry objects in the COLA_MARKETS table (that is, all except cola_d).

```
SELECT SDO_AGGR_UNION(
  SDOAGGRTYPE(c.shape, 0.005))
  FROM cola_markets c
  WHERE c.name < 'cola_d';
SDO_AGGR_UNION(SDOAGGRTYPE(C.SHAPE,0.005))(SDO_GTYPE, SDO_SRID, SDO_POINT(
--------------------------------------------------------------------------------
SDO_GEOMETRY(2007, NULL, NULL, SDO_ELEM_INFO_ARRAY(1, 1003, 2, 11, 1003, 1), SDO_
ORDINATE_ARRAY(8, 11, 6, 9, 8, 7, 10, 9, 8, 11, 1, 7, 1, 1, 5, 1, 8, 1, 8, 6, 5,
7, 1, 7))
```

Streams Transformations

The Streams transformations category contains one transformation called REPLICATE. The following section describes this transformation.

REPLICATE

Syntax

```
REPLICATE(lcr, conflict_resolution)
```

where `lcr` stands for Logical Change Record and encapsulates the DML change. Its data type is SYS.LCR$_ROW_RECORD. `conflict_resolution` is a Boolean variable. If its value is TRUE, any conflict resolution defined for the table will be used to resolve conflicts resulting from the execution of the LCR. For more information about conflict resolution, refer to *Oracle Streams Replication Administrators' Guide*.

Purpose

REPLICATE is used to replicate a DML change (INSERT, UPDATE, or DELETE) that has occurred on a table in the source system on an identical table in the target system. The table in the target system should be identical to the table in the source system in the following respects:.

- The name of the schema that contains the target table should be the same as the name of the schema that contains the source table.
- The name of the target table should the same as the name of the source table.
- The structure of the target table should be the same as that of the source table. The structure includes the number, name, and data type of the columns in the table.

Example

Consider a table T1(c1 varchar2(10), c2 number primary key) in schema S on the source system and an identical table in the target system. Consider the following insert operation on the table T1 on the source system

```
insert into T1 values ('abcde', 10)
```

An LCR representing the change following the above insert of a row on the table T1 in the source system will have the following details

```
LCR.GET_OBJECT_OWNER will be 'S'
LCR.GET_OBJECT_NAME will be 'T1'
LCR.GET_COMMAND_TYPE will be 'INSERT'
LCR.GET_VALUE('c1', 'new') will have the value for the column 'c1' - i.e. 'abcde'
LCR.GET_VALUE('c2', 'new') will have the value for the column 'c2' - i.e. 10
```

Such an LCR will be created and enqueued by a Streams Capture Process on the source system that captures changes on table S.T1

REPLICATE(lcr, true) - will result in a row ('abcde', 10) being inserted into the table T1 on the target system.

Note: Using this approach will not provide lineage information. If lineage is important, then do not use this function. Use the more direct approach of using an LCRCast operator bound to the source table and a table operator bound to the target table and connecting the attributes of these two operators with the same name ('Match by name'). Further information on LCR (Logical Change Record) is available in Oracle Database 10g Documentation (Information Integration)

XML Transformations

XML transformations provide Warehouse Builder users with functionality to perform transformations on XML objects. These transformations enable Warehouse Builder users to load and transform XML documents and Oracle AQs.

To enable loading of XML sources, Warehouse Builder provides access to the database XML functionality through custom functions, as detailed in this chapter.

Following are the XML transformations:

- EXISTSNODE on page 2-103
- EXTRACT on page 2-103
- EXTRACTVALUE on page 2-104
- SYS_XMLAGG on page 2-104
- SYS_XMLGEN on page 2-105
- WB_XML_LOAD on page 2-106
- WB_XML_LOAD_F on page 2-106
- XMLCONCAT on page 2-107
- XMLSEQUENCE on page 2-108
- XMLTRANSFORM on page 2-109

EXISTSNODE

Syntax

```
existsnode::=EXISTSNODE(XMLType_instance,XPath_string,namespace_string)
```

Purpose

`EXISTSNODE` determines whether traversal of an XML document using a specified path results in any nodes. It takes as arguments the `XMLType` instance containing an XML document and a VARCHAR2 XPath string designating a path. The optional `namespace_string` must resolve to a VARCHAR2 value that specifies a default mapping or namespace mapping for prefixes, which Oracle Database uses when evaluating the XPath expression(s).

This function returns a NUMBER value. It returns 0 if no nodes remain after applying the XPath traversal on the document. It returns 1 if any nodes remain.

Example

The following example tests for the existence of the `/Warehouse/Dock` node in the XML path of the `warehouse_spec` column of the sample table `oe.warehouses`:

```
SELECT warehouse_id, warehouse_name
   FROM warehouses
   WHERE EXISTSNODE(warehouse_spec, '/Warehouse/Docks') = 1;

WAREHOUSE_ID WAREHOUSE_NAME
------------ -----------------------------------
           1 Southlake, Texas
           2 San Francisco
           4 Seattle, Washington
```

EXTRACT

Syntax

```
extract::=EXTRACT(XMLType_instance,XPath_string,namespace_string)
```

Purpose

`EXTRACT` is similar to the `EXISTSNODE` function. It applies a VARCHAR2 XPath string and returns an `XMLType` instance containing an XML fragment. The optional `namespace_string` must resolve to a VARCHAR2 value that specifies a default mapping or namespace mapping for prefixes, which Oracle Database uses when evaluating the XPath expression(s).

Example

The following example extracts the value of the `/Warehouse/Dock` node of the XML path of the `warehouse_spec` column in the sample table `oe.warehouses`:

```
SELECT warehouse_name, EXTRACT(warehouse_spec, '/Warehouse/Docks')
   "Number of Docks"
   FROM warehouses
   WHERE warehouse_spec IS NOT NULL;

WAREHOUSE_NAME       Number of Docks
-------------------- --------------------
Southlake, Texas          <Docks>2</Docks>
```

```
San Francisco            <Docks>1</Docks>
New Jersey               <Docks/>
Seattle, Washington      <Docks>3</Docks>
```

EXTRACTVALUE

Syntax

```
extractvalue::=EXTRACTVALUE(XMLType_instance,XPath_string,namespace_string)
```

Purpose

The `EXTRACTVALUE` function takes as arguments an `XMLType` instance and an XPath expression and returns a scalar value of the resultant node. The result must be a single node and be either a text node, attribute, or element. If the result is an element, then the element must have a single text node as its child, and it is this value that the function returns. If the specified XPath points to a node with more than one child, or if the node pointed to has a non-text node child, then Oracle returns an error. The optional `namespace_string` must resolve to a VARCHAR2 value that specifies a default mapping or namespace mapping for prefixes, which Oracle uses when evaluating the XPath expression(s).

For documents based on XML schemas, if Oracle can infer the type of the return value, then a scalar value of the appropriate type is returned. Otherwise, the result is of type VARCHAR2. For documents that are not based on XML schemas, the return type is always VARCHAR2.

Example

The following example takes as input the same arguments as the example for EXTRACT. Instead of returning an XML fragment, as does the EXTRACT function, it returns the scalar value of the XML fragment:

```
SELECT warehouse_name,
   EXTRACTVALUE(e.warehouse_spec, '/Warehouse/Docks')
   "Docks"
   FROM warehouses e
   WHERE warehouse_spec IS NOT NULL;

WAREHOUSE_NAME       Docks
-------------------- ------------
Southlake, Texas     2
San Francisco        1
New Jersey
Seattle, Washington  3
```

SYS_XMLAGG

Syntax

```
sys_xmlagg::=SYS_XMLAGG(expr,fmt)
```

Purpose

`SYS_XMLAGG` aggregates all of the XML documents or fragments represented by `expr` and produces a single XML document. It adds a new enclosing element with a default name ROWSET. If you want to format the XML document differently, then specify `fmt`, which is an instance of the `XMLFormat` object.

Example

The following example uses the SYS_XMLGEN function to generate an XML document for each row of the sample table `employees` where the employee's last name begins with the letter R, and then aggregates all of the rows into a single XML document in the default enclosing element ROWSET:

```
SELECT SYS_XMLAGG(SYS_XMLGEN(last_name))
   FROM employees
   WHERE last_name LIKE 'R%';

SYS_XMLAGG(SYS_XMLGEN(LAST_NAME))
--------------------------------------------------------------------
<ROWSET>
  <LAST_NAME>Raphaely</LAST_NAME>
  <LAST_NAME>Rogers</LAST_NAME>
  <LAST_NAME>Rajs</LAST_NAME>
  <LAST_NAME>Russell</LAST_NAME>
</ROWSET>
```

SYS_XMLGEN

Syntax

```
sys_xmlgen::=SYS_XMLGEN(expr,fmt)
```

Purpose

`SYS_XMLGEN` takes an expression that evaluates to a particular row and column of the database, and returns an instance of type `XMLType` containing an XML document. The `expr` can be a scalar value, a user-defined type, or an `XMLType` instance.

If `expr` is a scalar value, then the function returns an XML element containing the scalar value. If `expr` is a type, then the function maps the user-defined type attributes to XML elements. If `expr` is an XMLType instance, then the function encloses the document in an XML element whose default tag name is ROW.

By default the elements of the XML document match the elements of `expr`. For example, if `expr` resolves to a column name, then the enclosing XML element will be the same column name. If you want to format the XML document differently, then specify `fmt`, which is an instance of the `XMLFormat` object.

Example

The following example retrieves the employee email ID from the sample table `oe.employees` where the `employee_id` value is 205, and generates an instance of an `XMLType` containing an XML document with an `EMAIL` element.

```
SELECT SYS_XMLGEN(email)
   FROM employees
   WHERE employee_id = 205;

SYS_XMLGEN(EMAIL)
-------------------------------------------------------------------
<EMAIL>SHIGGINS</EMAIL>
```

WB_XML_LOAD

Syntax:

```
WB_XML_LOAD(control_file)
```

Purpose

This program unit extracts and loads data from XML documents into database targets. The `control_file`, an XML document, specifies the source of the XML documents, the targets, and any runtime controls. After the transformation has been defined, a mapping in Warehouse Builder calls the transformation as a pre-map or post-map trigger.

Example

The following example illustrates a script that can be used to implement a Warehouse Builder transformation that extracts data from an XML document stored in the file products.xml and loads it into the target table called books:

```
begin
wb_xml_load('<OWBXMLRuntime>'
||
'<XMLSource>'
||
' <file>\ora817\GCCAPPS\products.xml</file>'
||
'</XMLSource>'
||
'<targets>'
||
' <target XSLFile="\ora817\XMLstyle\GCC.xsl">books</target>'
||
'</targets>'
||
'</OWBXMLRuntime>'
);
end;
```

For more information on control files, see the *Oracle Warehouse Builder User's Guide.*

WB_XML_LOAD_F

Syntax

```
WB_XML_LOAD_F(control_file)
```

Purpose

`WB_XML_LOAD_F` extracts and loads data from XML documents into database targets. The function returns the number of XML documents read during the load. The `control_file`, itself an XML document, specifies the source of the XML documents, the targets, and any runtime controls. After the transformation has been defined, a mapping in Warehouse Builder calls the transformation as a pre-map or post-map trigger.

Example

The following example illustrates a script that can be used to implement a Warehouse Builder transformation that extracts data from an XML document stored in the file `products.xml` and loads it into the target table books:

```
begin
wb_xml_load_f('<OWBXMLRuntime>'
||
'<XMLSource>'
||
' <file>\ora817\GCCAPPS\products.xml</file>'
||
'</XMLSource>'
||
'<targets>'
||
' <target XSLFile="\ora817\XMLstyle\GCC.xsl">books</target>'
||
'</targets>'
||
'</OWBXMLRuntime>'
);
end;
```

For more information on the types handled and detailed information on `control_files`, see the *Oracle Warehouse Builder User's Guide*.

XMLCONCAT

Syntax

```
xmlconcat::=XMLCONCAT(XMLType_instance)
```

Purpose

`XMLCONCAT` takes as input a series of `XMLType` instances, concatenates the series of elements for each row, and returns the concatenated series. `XMLCONCAT` is the inverse of XMLSEQUENCE. Null expressions are dropped from the result. If all the value expressions are null, then the function returns null.

Example

The following example creates XML elements for the first and last names of a subset of employees, and then concatenates and returns those elements:

```
SELECT XMLCONCAT(XMLELEMENT("First", e.first_name),
   XMLELEMENT("Last", e.last_name)) AS "Result"
   FROM employees e
   WHERE e.employee_id > 202;

Result
----------------------------------------------------------------
<First>Susan</First>
<Last>Mavris</Last>

<First>Hermann</First>
<Last>Baer</Last>

<First>Shelley</First>
<Last>Higgins</Last>
```

```
<First>William</First>
<Last>Gietz</Last>
```

XMLSEQUENCE

Syntax

```
xmlsequence:=xmlsequence(XMLType_instance XMLType)
```

Purpose

`XMLSEQUENCE` takes an `XMLType` instance as input and returns a varray of the top-level nodes in the `XMLType`. You can use this function in a `TABLE` clause to unnest the collection values into multiple rows, which can in turn be further processed in the SQL query.

Example

The following example shows how `XMLSequence` divides up an XML document with multiple elements into VARRAY single-element documents. The TABLE keyword instructs the Oracle Database to consider the collection a table value that can be used in the FROM clause of the subquery.

```
SELECT EXTRACT(warehouse_spec, '/Warehouse') as "Warehouse"
   FROM warehouses
   WHERE warehouse_name = 'San Francisco';

Warehouse
------------------------------------------------------------
<Warehouse?
  <Building>Rented</Building>
  <Area>50000</Area>
  <Docks>1</Docks>
  <DockType>Side load</DockType>
  <WaterAccess>Y</WaterAccess>
  <RailAccess>N</RailAccess>
  <Parking>Lot</Parking>
  <VClearance>12 ft</VClearance>
</Warehouse>


SELECT VALUE(p)
   FROM warehouses w,
   TABLE(XMLSEQUENCE(EXTRACT(warehouse_spec,'/Warehouse/*'))) p
   WHERE w.warehouse_name = 'San Francisco';

VALUE(p)
------------------------------------------------------------

<Building>Rented</Building>
<Area>50000</Area>
<Docks>1</Docks>
<DockType>Side load</DockType>
<WaterAccess>Y</WaterAccess>
<RailAccess>N</RailAccess>
<Parking>Lot</Parking>
<VClearance>12 ft</VClearance>
```

XMLTRANSFORM

Syntax

```
xmltransform::=XMLTRANSFORM(XMLType_instance,XMLType_instance)
```

Purpose

XMLTRANSFORM takes as arguments an XMLType instance and an XSL style sheet, which is itself a form of XMLType instance. It applies the style sheet to the instance and returns an XMLType. This function is useful for organizing data according to a style sheet as you are retrieving it from the database.

Example

The XMLTRANSFORM function requires the existence of an XSL style sheet. Here is an example of a very simple style sheet that alphabetizes elements within a node:

```
CREATE TABLE xsl_tab (col1 XMLTYPE);

INSERT INTO xsl_tab VALUES (
   XMLTYPE.createxml(
   '<?xml version="1.0"?>
    <xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
>
      <xsl:output encoding="utf-8"/>
      <!-- alphabetizes an xml tree -->
      <xsl:template match="*">
        <xsl:copy>
          <xsl:apply-templates select="*|text()">
            <xsl:sort select="name(.)" data-type="text" order="ascending"/>
          </xsl:apply-templates>
        </xsl:copy>
      </xsl:template>
      <xsl:template match="text()">
        <xsl:value-of select="normalize-space(.)"/>
      </xsl:template>
    </xsl:stylesheet>  '));
```

The next example uses the xsl_tab XSL style sheet to alphabetize the elements in one warehouse_spec of the sample table oe.warehouses:

```
SELECT XMLTRANSFORM(w.warehouse_spec, x.col1).GetClobVal()
   FROM warehouses w, xsl_tab x
   WHERE w.warehouse_name = 'San Francisco';

XMLTRANSFORM(W.WAREHOUSE_SPEC,X.COL1).GETCLOBVAL()
--------------------------------------------------------------------------------
<Warehouse>
  <Area>50000</Area>
  <Building>Rented</Building>
  <DockType>Side load</DockType>
  <Docks>1</Docks>
  <Parking>Lot</Parking>
  <RailAccess>N</RailAccess>
  <VClearance>12 ft</VClearance>
  <WaterAccess>Y</WaterAccess>
</Warehouse>
```

Index