Oracle® Database

Application Developer's Guide - Expression Filter

10*g* Release 1 (10.1)

Part No. B10821-01

December 2003

Oracle Database Application Developer’s Guide - Expression Filter, 10*g* Release 1 (10.1)

Part No. B10821-01

Primary Author: Aravind Yalamanchi

Contributors: William Beauregard, Dieter Gawlick, Helen Grembowicz, Deborah Owens, and Jagannathan Srinivasan

Contents

List of Examples

List of Figures

List of Tables

Send Us Your Comments

Oracle Database Application Developer's Guide - Expression Filter, 10*g* Release 1 (10.1)

Part No. B10821-01

Oracle Corporation welcomes your comments and suggestions on the quality and usefulness of this publication. Your input is an important part of the information used for revision.

- Did you find any errors?
- Is the information clearly presented?
- Do you need more information? If so, where?
- Are the examples correct? Do you need more examples?
- What features did you like most about this manual?

If you find any errors or have any other suggestions for improvement, please indicate the title and part number of the documentation and the chapter, section, and page number (if available). You can send comments to us in the following ways:

- Electronic mail: nedc-doc_us@oracle.com
- FAX: 603.897.3825   Attn: Expression Filter Documentation
- Postal service:
  Oracle Corporation
  Expression Filter Documentation
  One Oracle Drive
  Nashua, NH 03062-2804
  USA

If you would like a reply, please provide your name and contact information.

If you have problems with the software, please contact your local Oracle Support Services.

Preface

Oracle Database Application Developer's Guide - Expression Filter provides usage and reference information about Expression Filter, a feature of Oracle Database that stores, indexes, and evaluates conditional expressions in relational tables.

Audience

Application developers and DBAs can save time and labor by using Oracle Expression Filter to store and evaluate large sets of conditional expressions in the database. Conditional expressions can describe business rules and interests in expected data for applications involving personalized information distribution, demand analysis, and task assignment.

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. Standards will continue to evolve over time, and Oracle Corporation 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 additional information, visit the Oracle Accessibility Program Web site at

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

Accessibility of Code Examples in Documentation JAWS, a Windows screen reader, 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, JAWS 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 Corporation does not own or control. Oracle Corporation neither evaluates nor makes any representations regarding the accessibility of these Web sites.

Related Documentation

Refer to the following documentation for information about related products:

- *Oracle Database SQL Reference*
- *Oracle Database Utilities*
- *Oracle Database Error Messages*
- *Oracle Database Performance Tuning Guide*
- *Oracle XML DB Developer's Guide*
- *Oracle Database Application Developer's Guide - Object-Relational Features*

Printed documentation is available for sale in the Oracle Store at

```
http://oraclestore.oracle.com/
```

To download free release notes, installation documentation, white papers, or other collateral, please visit the Oracle Technology Network (OTN). You must register online before using OTN; registration is free and can be done at

```
http://otn.oracle.com/membership/
```

If you already have a username and password for OTN, then you can go directly to the documentation section of the OTN Web site at

```
http://otn.oracle.com/documentation/
```

Conventions

This section describes the conventions used in the text and code examples of this document. The following table describes those conventions and provides examples of their use.

Convention	Meaning	Example
[ ]	Brackets enclose one or more optional items. Do not enter the brackets.	`DECIMAL (`*`digits`*` [ , `*`precision`*` ])`
{ }	Braces enclose two or more items, one of which is required. Do not enter the braces.	`{ENABLE | DISABLE}`

|	A vertical bar represents a choice of two or more options within brackets or braces. Enter one of the options. Do not enter the vertical bar.	`{ENABLE | DISABLE}`
		`[COMPRESS | NOCOMPRESS]`

. . .	Horizontal ellipsis points indicate either:	
	■ That we have omitted parts of the code that are not directly related to the example	`CREATE TABLE ... AS `*`subquery`*`;`
	■ That you can repeat a portion of the code	`SELECT `*`col1`*`, `*`col2`*`, ... , `*`coln`*` FROM employees;`
. . .	Vertical ellipsis points indicate that we have omitted several lines of code not directly related to the example.	
Bold	Bold typeface indicates terms that are defined in the text or terms that appear in a glossary, or both.	When you specify this clause, you create an **index-organized table**.
`UPPERCASE monospace (fixed-width font)`	Uppercase monospace typeface indicates elements supplied by the system.	You can back up the database by using the `BACKUP` command. Query the `TABLE_NAME` column in the `USER_TABLES` data dictionary view. Use the `DBMS_STATS.GENERATE_STATS` procedure.
`lowercase monospace (fixed-width font)`	Lowercase monospace typeface indicates executables, filenames, directory names, and sample user-supplied elements.	Enter `sqlplus` to open SQL*Plus. Back up the datafiles and control files in the `/disk1/oracle/dbs` directory. The `department_id`, `department_name`, and `location_id` columns are in the `hr.departments` table.
`lowercase monospace (fixed-width font) italic`	Lowercase monospace italic font represents placeholders or variables.	You can specify the *`parallel_clause`*. Run `U`*`old_release`*`.SQL` where *`old_release`* refers to the release you installed prior to upgrading.

1 Oracle Expression Filter Concepts

Oracle Expression Filter is a feature of Oracle Database that allows application developers to store, index, and evaluate conditional expressions (expressions) in one or more columns of a relational table. Expressions are a useful way to describe interests in expected data.

Expression Filter matches incoming data with expressions stored in a column to identify rows of interest. It can also derive complex relationships by matching data in one table with expressions in a second table. Expression Filter simplifies SQL queries; allows expressions to be inserted, updated, and deleted without changing the application; and enables reuse of conditional expressions in business rules by separating them from the application and storing them in the database. Applications involving information distribution, demand analysis, and task assignment can benefit from Expression Filter.

1.1 What Is Expression Filter?

Expression Filter provides a datatype, operator, and indextype to store, evaluate, and index expressions that describe an interest in a data item or piece of information. Expressions are stored in a column of a user table. Expression Filter matches expressions in a column with a data item passed by a SQL statement or with data stored in one or more tables, and evaluates each expression to be true or false. Optionally, expressions can be indexed when using the Enterprise Edition of Oracle Database. Expression Filter includes the following elements:

- Expression datatype: A virtual datatype created through a constraint placed on a VARCHAR2 column in a user table that stores expressions.
- `EVALUATE` operator: An operator that evaluates expressions for each data item.
- Administrative utilities: A set of utilities that validate expressions and suggest optimal index structure.

- Expression indexing: Enhances performance of the EVALUATE operator for large expression sets. Expression indexing is available in Oracle Database Enterprise Edition.

1.1.1 Expression Filter Usage Scenarios

The following sections are examples of how you can use Expression Filter.

Match Incoming Data with Conditional Expressions

Expression Filter can match incoming data with conditional expressions stored in the database to identify rows of interest. For example, consider an application that matches buyers and sellers of cars. A table called Consumer includes a column called BUYER_PREFERENCES with an Expression datatype. The BUYER_PREFERENCES column stores an expression for each consumer that describes the kind of car the consumer wants to purchase, including make, model, year, mileage, color, options, and price. Data about cars for sale is included with the EVALUATE operator in the SQL WHERE clause. The SQL EVALUATE operator matches the incoming car data with the expressions to find prospective buyers.

The SQL EVALUATE operator also enables batch processing of incoming data. Data can be stored in a table called CARS and matched with expressions stored in the CONSUMER table using a join between the two tables.

The SQL EVALUATE operator saves time by matching a set of expressions with incoming data and enabling large expression sets to be indexed for performance. This saves labor by allowing expressions to be inserted, updated, and deleted without changing the application and providing a results set that can be manipulated in the same SQL statement, for instance to order or group results. In contrast, a procedural approach stores results in a temporary table that must be queried for further processing, and those expressions cannot be indexed.

Maintain Complex Table Relationships

Expression Filter can convey *N*-to-*M* (many-to-many) relationships between tables. Using the previous example:

- A car may be of interest to one or more buyers.
- A buyer may be interested in one or more cars.
- A seller may be interested in one or more buyers.

To answer questions about these relationships, the incoming data about cars is stored in a table called CARS with an Expression column (column of Expression datatype) called SELLER_PREFERENCES. The CONSUMERS table includes a column called BUYER_PREFERENCES. The SQL EVALUATE operator can answer questions such as:

- What cars are of interest to each consumer?
- What buyers are of interest to each seller?
- What demand exists for each car? This can help to determine optimal pricing.
- What unsatisfied demand is there? This can help to determine inventory requirements.

This declarative approach saves labor. No action is needed if changes are made to the data or the expressions. Compare this to the traditional approach where a mapping table is created to store the relationship between the two tables. A trigger must be defined to recompute the relationships and to update the mapping table if the data or expressions change. In this case, new data must be compared to all expressions, and a new expression must be compared to all data.

Application Attributes

Expression Filter is a good fit for applications where the data has the following attributes:

- A large number of data items exists to be evaluated.
- Each data item has structured data attributes, for example VARCHAR, NUMBER, DATE, XMLTYPE.
- Incoming data is evaluated by a significant number of unique and persistent queries containing expressions.
- The expression (in SQL WHERE clause format) describes an interest in incoming data items.
- The expressions compare attributes to values using relational operators (=, !=, <, >, and so on).

1.2 Introduction to Expressions

Expressions describe interests in an item of data. Expressions are stored in a column of a user table and compared, using the SQL EVALUATE operator, to incoming data items specified in a SQL WHERE clause or to a table of data. Expressions are evaluated as true or false or return a null value if an expression does not exist for a row.

An **expression** describes interest in an item of data using one or more variables, known as **elementary attributes**. An expression can also include literals, Oracle supplied functions, user-defined functions, and table aliases. A valid expression consists of one or more simple conditions called predicates. The predicates in the expression are linked by the logical operators `AND` and `OR`. Expressions must adhere to the SQL `WHERE` clause format. (For more information about the SQL `WHERE` clause, see *Oracle Database SQL Reference*.) An expression is not required to use all the defined elementary attributes; however, the incoming data must provide a value for every elementary attribute. Null is an acceptable value.

For example, the following expression includes the `UPPER` Oracle supplied function and captures the interest of a user in a car (the data item) with the model, price, and year as elementary attributes.

```
UPPER(Model) = 'TAURUS' and Price < 20000 and Year > 2000
```

Expressions are stored in a column of a user table with an Expression datatype. The values stored in a column of this type are constrained to be expressions. (See Section 1.2.2.) A user table can have one or more Expression columns. A query to display the contents of an Expression column displays the expressions in string format.

You insert, update, and delete expressions using standard SQL. A group of expressions that are stored in a single column is called an **expression set** and shares a common set of elementary attributes. This set of elementary attributes plus any functions used in the expressions are the metadata for the expression set. This metadata is referred to as the **attribute set**. The attribute set consists of the elementary attribute names and their datatypes and any functions used in the expressions. The attribute set is used by the Expression column to validate changes and additions to the expression set. An expression stored in the Expression column can use only the elementary attribute and functions defined in the corresponding attribute set. Expressions cannot contain subqueries.

Expression Filter provides the DBMS_EXPFIL package which contains procedures to manage the expression data.

There are four basic steps to create and use an Expression column:

1. Define an attribute set. See Section 1.2.1.
2. Define an Expression column in a user table. See Section 1.2.2.
3. Insert expressions in the table. See Section 1.2.3.
4. Apply the SQL `EVALUATE` operator to compare expressions to incoming data items. See Section 1.3.

The remaining sections in this chapter guide you through this procedure.

1.2.1 Defining Attribute Sets

A special form of an Oracle object type is used to create an attribute set. (For more information about object types, see *Oracle Database Application Developer's Guide - Object-Relational Features*.)

The attribute set defines the elementary attributes for an expression set. It implicitly allows all Oracle supplied SQL functions to be valid references in the expression set. If the expression set refers to a user-defined function, it must be explicitly added to the attribute set. An elementary attribute in an attribute set can refer to data stored in another database table using table alias constructs. One or more or all elementary attributes in an attribute set can be table aliases. If an elementary attribute is a table alias, the value assigned to the elementary attribute is a `ROWID` from the corresponding table. For more information about table aliases, see Appendix A.

You can create an attribute set using one of two approaches:

- Use an existing object type to create an attribute set with the same name as the object type. This approach is most appropriate to use when the attribute set does not contain any table alias elementary attributes. You use the CREATE_ ATTRIBUTE_SET procedure of the DBMS_EXPFIL package. See Example 1–1.
- Individually add elementary attributes to an existing attribute set. Expression Filter automatically creates an object type to encapsulate the elementary attributes and gives it the same name as the attribute set. This approach is most appropriate to use when the attribute set contains one or more elementary attributes defined as table aliases. You use the ADD_ELEMENTARY_ ATTRIBUTE procedure of the DBMS_EXPFIL package. See Example 1–2.

If the expressions refer to user-defined functions, you must add the functions to the corresponding attribute set, using the ADD_FINCTIONS procedure of the DBMS_ EXPFIL package. See Example 1–3.

Attribute Set Examples

Example 1–1 shows how to use an existing object type to create an attribute set. It uses the CREATE_ATTRIBUTE_SET procedure.

Example 1–1 Defining an Attribute Set From an Existing Object Type

```
CREATE OR REPLACE TYPE Car4Sale AS OBJECT
                                   (Model   VARCHAR2(20),
                                    Year    NUMBER,
                                    Price   NUMBER,
                                    Mileage NUMBER);
/


BEGIN
  DBMS_EXPFIL.CREATE_ATTRIBUTE_SET(attr_set  => 'Car4Sale',
                                   from_type => 'YES');
END;
/
```

For more information about the CREATE_ATTRIBUTE_SET procedure, see "CREATE_ATTRIBUTE_SET Procedure" in Chapter 8.

Example 1–2 shows how to create an attribute set `Car4Sale` and how to define the variables one at a time. It uses the CREATE_ATTRIBUTE_SET and ADD_ELEMENTARY_ATTRIBUTE procedures.

Example 1–2 Defining an Attribute Set Incrementally

```
BEGIN
  DBMS_EXPFIL.CREATE_ATTRIBUTE_SET(attr_set => 'Car4Sale');
  DBMS_EXPFIL.ADD_ELEMENTARY_ATTRIBUTE(
                             attr_set  => 'Car4Sale',
                             attr_name => 'Model',
                             attr_type => 'VARCHAR2(20)');
  DBMS_EXPFIL.ADD_ELEMENTARY_ATTRIBUTE(
                             attr_set  => 'Car4Sale',
                             attr_name => 'Year',
                             attr_type => 'NUMBER');
  DBMS_EXPFIL.ADD_ELEMENTARY_ATTRIBUTE(
                             attr_set  => 'Car4Sale',
                             attr_name => 'Price',
                             attr_type => 'NUMBER');
  DBMS_EXPFIL.ADD_ELEMENTARY_ATTRIBUTE(
                             attr_set  => 'Car4Sale',
                             attr_name => 'Mileage',
                             attr_type => 'NUMBER');
END;
/
```

For more information about the ADD_ELEMENTARY_ATTRIBUTE procedure, see "ADD_ELEMENTARY_ATTRIBUTE Procedure" in Chapter 8.

If the expressions refer to user-defined functions, you must add the functions to the corresponding attribute set. Example 1–3 shows how to add user-defined functions, using the ADD_FUNCTIONS procedure, to an attribute set.

Example 1–3 Adding User-Defined Functions to an Attribute Set

```
CREATE or REPLACE FUNCTION HorsePower(Model VARCHAR2, Year VARCHAR2)
    return NUMBER is
BEGIN
-- Derive HorsePower from other relational tables uisng Model and Year values.--
  return 200;
END HorsePower;
/

CREATE or REPLACE FUNCTION CrashTestRating(Model VARCHAR2, Year VARCHAR2)
    return NUMBER is
BEGIN
-- Derive CrashTestRating from other relational tables using Model --
-- and Year values. --
  return 5;
END CrashTestRating;
/

BEGIN
  DBMS_EXPFIL.ADD_FUNCTIONS (attr_set   => 'Car4Sale',
                             funcs_name => 'HorsePower');
  DBMS_EXPFIL.ADD_FUNCTIONS (attr_set   => 'Car4Sale',
                             funcs_name => 'CrashTestRating');
END;
/
```

For more information about the ADD_FUNCTIONS procedure, see "ADD_FUNCTIONS Procedure" in Chapter 8.

To drop an attribute set, you use the DROP_ATTRIBUTE_SET procedure. For more information, see "DROP_ATTRIBUTE_SET Procedure" in Chapter 8.

1.2.2 Defining Expression Columns

Expression is a virtual datatype. Assigning an attribute set to a `VARCHAR2` column in a user table creates an Expression column. The attribute set determines which elementary attributes and user-defined functions can be used in the expression set. An attribute set can be used to create multiple columns of Expression datatype in the same table and in other tables in the same schema. Note that an attribute set in one schema cannot be associated with a column in another schema.

To create an Expression column:

1. Add a VARCHAR2 column to a table or create a table with the VARCHAR2 column. An existing VARCHAR2 column in a user table can also be used for this purpose. The following example creates a table with a VARCHAR2 column, Interest, that will be used with an attribute set:

   ```
   CREATE TABLE Consumer (CId          NUMBER,
                          Zipcode      NUMBER,
                          Phone        VARCHAR2(12),
                          Interest     VARCHAR2(200));
   ```

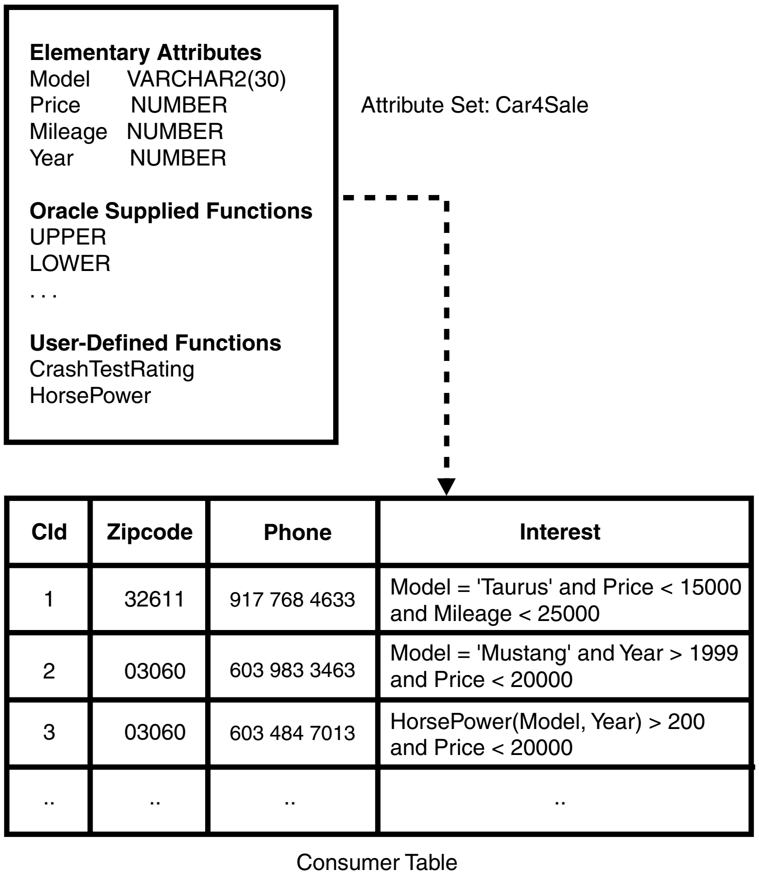
2. Assign an attribute set to the column, using the ASSIGN_ATTRIBUTE_SET procedure. The following example assigns an attribute set to a column named Interest in a table called Consumer:

   ```
   BEGIN
     DBMS_EXPFIL.ASSIGN_ATTRIBUTE_SET (
                                         attr_set => 'Car4Sale',
                                         expr_tab => 'Consumer',
                                         expr_col => 'Interest');
   END;
   /
   ```

   For more information about the ASSIGN_ATTRIBUTE_SET procedure, see "ASSIGN_ATTRIBUTE_SET Procedure" in Chapter 8.

Figure 1–1 is a conceptual image of consumers' interests (in trading cars) being captured in a Consumer table.

Figure 1–1 Expression Datatype



CId	Zipcode	Phone	Interest
1	32611	917 768 4633	Model = 'Taurus' and Price < 15000 and Mileage < 25000
2	03060	603 983 3463	Model = 'Mustang' and Year > 1999 and Price < 20000
3	03060	603 484 7013	HorsePower(Model, Year) > 200 and Price < 20000
..	..	..	..

To remove an attribute set from a column, you use the UNASSIGN_ATTRIBUTE_SET procedure of the DBMS_EXPFIL package. See "UNASSIGN_ATTRIBUTE_SET Procedure" in Chapter 8.

To drop an attribute set not being used for any expression set, you use the DROP_ATTRIBUTE_SET procedure of the DBMS_EXPFIL package. See "DROP_ATTRIBUTE_SET Procedure" in Chapter 8.

To copy an attribute set across schemas, you use the COPY_ATTRIBUTE_SET procedure of the DBMS_EXPFIL package. See "COPY_ATTRIBUTE_SET Procedure" in Chapter 8.

1.2.3 Inserting, Updating, and Deleting Expressions

You use standard SQL to insert, update, and delete expressions. When an expression is inserted or updated, it is checked for correct syntax and constrained to use the elementary attributes and functions specified in the corresponding attribute set. An error message is returned if the expression is not correct. For more information about evaluation semantics, see Section 1.4.

Example 1–4 shows how to insert an expression (the consumer's interest in trading cars, which is depicted in Figure 1–1) into the `Consumer` table using the SQL `INSERT` statement.

Example 1–4 Inserting an Expression into the Consumer Table

```
INSERT INTO Consumer VALUES (1, 32611, '917 768 4633',
            'Model=''Taurus'' and Price < 15000 and Mileage < 25000');
INSERT INTO Consumer VALUES (2, 03060, '603 983 3464',
                'Model=''Mustang'' and Year > 1999 and Price < 20000');
```

If an expression refers to a user-defined function, the function must be added to the corresponding attribute set (as shown in Example 1–3). Example 1–5 shows how to insert an expression with a reference to a user-defined function, `HorsePower`, into the `Consumer` table.

Example 1–5 Inserting an Expression That References a User-Defined Function

```
INSERT INTO Consumer VALUES (3, 03060, '603 484 7013',
                            'HorsePower(Model, Year) > 200 and Price < 20000');
```

Expression data can be loaded into an Expression column using SQL*Loader. For more information about bulk loading, see Section 5.1.

1.3 Applying the SQL EVALUATE Operator

You use the SQL `EVALUATE` operator in the `WHERE` clause of a SQL statement to compare stored expressions to incoming data items. The SQL `EVALUATE` operator returns `1` for an expression that matches the data item and `0` for an expression that does not match. For any null values stored in the Expression column, the SQL `EVALUATE` operator returns `NULL`.

The SQL `EVALUATE` operator has two arguments: the name of the column storing the expressions and the data item to which the expressions are compared. In the data item argument, values must be provided for all elementary attributes in the attribute set associated with the Expression column. Null is an acceptable value.

The data item can be specified either as string-formatted name-value pairs or as an `AnyData` instance.

In the following example, the query returns a row from the `Consumer` table if the expression in the `Interest` column evaluates to true for the data item:

```
SELECT * FROM Consumer WHERE
   EVALUATE (Consumer.Interest, <data item>) = 1;
```

Data Item Formatted as a String

If the values of all the elementary attributes in the attribute set can be represented as readable values, such as those stored in `VARCHAR`, `DATE`, and `NUMBER` datatypes and the constructors formatted as a string, then the data item can be formatted as a string:

Operator Form

```
EVALUATE (VARCHAR2, VARCHAR2)
    returns NUMBER;
```

Example

```
SELECT * FROM Consumer WHERE
    EVALUATE (Consumer.Interest,
              'Model=>''Mustang'',
               Year=>2000,
               Price=>18000,
               Mileage=>22000'
               ) = 1;
```

If a data item does not require a constructor for any of its elementary attribute values, then a list of values provided for the data item can be formatted as a string (name-value pairs) using two `getVarchar` methods (a `STATIC` method and a `MEMBER` method) in the object type associated with the attribute set. The `STATIC` method formats the data item without creating the object instance. The `MEMBER` method can be used if the object instance is already available.

The `STATIC` and `MEMBER` methods are implicitly created for the object type and can be used as shown in the following example:

```
SELECT * FROM Consumer WHERE
    EVALUATE (Consumer.Interest,
              Car4Sale.getVarchar('Mustang',   -- STATIC getVarchar API --
                                   2000,
                                   18000,
                                   22000)
              ) = 1;

SELECT * FROM Consumer WHERE
    EVALUATE (Consumer.Interest,
              Car4Sale('Mustang',
                        2000,
                        18000,
                        22000).getVarchar()     -- MEMBER getVarchar() API --
              ) = 1;
```

Data Item Formatted as an AnyData Instance

Any data item can be formatted using an AnyData instance. AnyData is an Oracle supplied object type that can hold instances of any Oracle datatype, both Oracle supplied and user-defined. For more information, see *Oracle Database Application Developer's Guide - Object-Relational Features*.

Operator Form

```
EVALUATE (VARCHAR2, AnyData)
  returns NUMBER;
```

An instance of the object type capturing the corresponding attribute set is converted into an AnyData instance using the AnyData's `convertObject` method. Using the previous example, the data item can be passed to the SQL `EVALUATE` operator by converting the instance of the `Car4Sale` object type into AnyData, as shown in the following example:

```
SELECT * FROM Consumer WHERE
  EVALUATE (Consumer.Interest,
            AnyData.convertObject(
             Car4Sale('Mustang',
                         2000,
                         18000,
                         22000))
         ) = 1;
```

A data item formatted as an AnyData instance is converted back into the original object before the expressions are evaluated. To avoid the cost of object type conversions, string-formatted data items are recommended whenever possible.

For the syntax of the SQL `EVALUATE` operator, see "EVALUATE" in Chapter 6. For additional examples of the SQL `EVALUATE` operator, see Appendix B.

1.4 Evaluation Semantics

When an expression is inserted or updated, Expression Filter validates the syntax and ensures that the expression refers to valid elementary attributes and functions associated with the attribute set. The SQL EVALUATE operator evaluates expressions using the privileges of the owner of the table that stores the expressions. For instance, if an expression includes a reference to a user-defined function, during its evaluation, the function is executed with the privileges of the owner of the table. References to schema objects with no schema extensions are resolved in the table owner's schema.

An expression that refers to a user-defined function may become invalid if the function is modified or dropped. An invalid expression causes the SQL statement evaluating the expression to fail. To recover from this error, replace the missing or modified function with the original function.

The Expression Validation utility is used to verify an expression set. It identifies expressions that have become invalid since they were inserted, perhaps due to a change made to a user-defined function or table. This utility collects references to the invalid expressions in an exception table. If an exception table is not provided, the utility fails when it encounters the first invalid expression in the expression set.

The following commands collect references to invalid expressions found in the Consumer table. The BUILD_EXCEPTIONS_TABLE procedure creates the exception table, InterestExceptions, in the current schema. The VALIDATE_EXPRESSIONS procedure validates the expressions and stores the invalid expressions in the InterestExceptions table.

```
BEGIN
  DBMS_EXPFIL.BUILD_EXCEPTIONS_TABLE (exception_tab => 'InterestExceptions');

  DBMS_EXPFIL.VALIDATE_EXPRESSIONS (expr_tab => 'Consumer',
                                    expr_col => 'Interest',
                                    exception_tab => 'InterestExceptions');
END;
/
```

For more information, see "BUILD_EXCEPTIONS_TABLE Procedure" and "VALIDATE_EXPRESSIONS Procedure", both in Chapter 8.

1.5 Granting and Revoking Privileges

A user requires SELECT privileges on a table storing expressions to evaluate them. The SQL EVALUATE operator evaluates expressions using the privileges of the owner of the table that stores the expressions. The privileges of the user issuing the query are not considered.

Expressions can be inserted, updated, and deleted by the owner of the table. Others must have INSERT and UPDATE privileges for the table, and they must have INSERT EXPRESSION and UPDATE EXPRESSION privileges for a specific Expression column in the table to be able to make modifications to it.

In the following example, the owner of the Consumer table grants expression privileges, using the GRANT_PRIVILEGE procedure, on the Interest column to a user named Andy:

```
BEGIN
  DBMS_EXPFIL.GRANT_PRIVILEGE (expr_tab => 'Consumer',
                               expr_col => 'Interest',
                               priv_type => 'INSERT EXPRESSION',
                               to_user => 'Andy');
END;
/
```

To revoke privileges, use the REVOKE_PRIVILEGE procedure.

For more information about granting and revoking privileges, see "GRANT_PRIVILEGE Procedure" and "REVOKE_PRIVILEGE Procedure" in Chapter 8.

1.6 Error Messages

The Expression Filter error message numbers are in the range of 38401 to 38600. The error messages are documented in *Oracle Database Error Messages*.

Oracle error message documentation is only available in HTML. If you only have access to the Oracle Documentation CD, you can browse the error messages by range. Once you find the specific range, use your browser's **find in page** feature to locate the specific message. When connected to the Internet, you can search for a specific error message using the error message search feature of the Oracle online documentation.

2 Indexing Expressions

Note: Expression indexing is available only in Oracle Database Enterprise Edition.

An index can be defined on a column storing expressions to quickly find expressions that evaluate to true for a data item. This is most helpful when a large expression set is evaluated for a data item. The SQL `EVALUATE` operator determines whether or not to use the index based on its access cost. The indextype, Expression Filter, is used to create and maintain indexes.

If an Expression column is not indexed, the SQL `EVALUATE` operator builds a dynamic query for each expression stored in the column and executes it using the values passed in as the data item.

This chapter describes the basic approach to indexing including index representation (Section 2.3), index processing (Section 2.4), and user commands for creating and tuning indexes (Section 2.6).

2.1 Concepts of Indexing Expressions

Expressions in a large expression set tend to have certain commonalities in their predicates. An Expression Filter index, defined on an expression set, groups predicates by their commonalities to reduce processing costs. For example, in the case of two predicates with a common left-hand side, such as `Year=1998` and `Year=1999`, in most cases, the falseness or trueness of one predicate can be determined based on the outcome of the other predicate. The left-hand side of a predicate includes arithmetic expressions containing one or more elementary attributes and user-defined functions, for example, `HORSEPOWER(model, year)`.

An operator and a constant on the right-hand side (RHS) completes the predicate, for example, `HORSEPOWER(model, year)>=150`.

An Expression Filter index defined on a set of expressions takes advantage of the logical relationships between multiple predicates by grouping them based on the commonality of their left-hand sides. These left-hand sides are arithmetic expressions that consist of one or more elementary attributes and user-defined functions, for example, `HORSEPOWER(model,year)`. In the expression set, these left-hand sides appear in the predicates along with an operator and a constant on the right-hand side (RHS), for example, `HORSEPOWER(model,year)>=150`.

2.2 Indexable Predicates

The predicates that can be indexed with the Expression Filter indexing mechanism include any predicate with a constant on the right-hand side that uses one of the following predicate operators: `=`, `!=`, `>`, `<`, `>=`, `<=`, `BETWEEN`, `IS NULL`, `IS NOT NULL`, `LIKE`, and `NVL`.

The predicates that cannot be indexed are preserved in their original form and they are evaluated by value substitution in the last stage of expression evaluation. Some of the predicates that cannot be indexed include:

- Predicates with a variable on the right-hand side.
- `IN` list predicates.
- `LIKE` predicates with a leading wild-card character.
- Duplicate predicates in an expression with the same left-hand side. At most, two predicates with a duplicate left-hand side, for example `Year>1995 and Year<2000`, can be indexed if the index is configured for `BETWEEN` operators. See the section about EXF$INDEXOPER in Chapter 7.

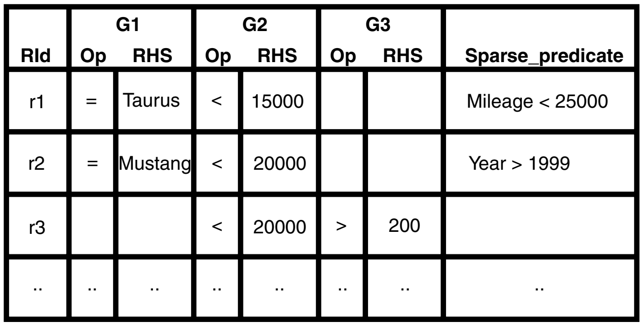
2.3 Index Representation

The Expression Filter index uses persistent database objects internally to maintain the index information for an expression set. The grouping information for all the predicates in an expression set is captured in a relational table called the predicate table. Typically, the predicate table contains one row for each expression in the expression set. An expression containing one or more disjunctions (two simple expressions joined by `OR`) is converted into a disjunctive-normal form (disjunction of conjunctions), and each disjunction in this normal form is treated as a separate expression with the same identifier as the original expression. The predicate table contains one row for each such disjunction.

The Expression Filter index can be tuned for better performance by identifying the most-common left-hand sides of the predicates (or discriminating predicate groups) in the expression set. The owner of the expression set (or the table storing expressions) can identify the predicate's left-hand sides or automate this process by collecting statistics on the expression set. For each common left-hand side, a predicate group is formed with all the corresponding predicates in the expression set. For example, if predicates with `Model`, `Price,` and `HorsePower(Model, Year)` attributes are common in the expression set, three predicate groups are formed for these attributes. The predicate table captures the predicate grouping information as shown in Figure 2–1.

Figure 2–1 Conceptual Predicate Table

Predicate table for the expressions stored in
the Interest column of the Consumer table



	G1		G2		G3		
RId	**Op**	**RHS**	**Op**	**RHS**	**Op**	**RHS**	**Sparse_predicate**
r1	=	Taurus	<	15000			Mileage < 25000
r2	=	Mustang	<	20000			Year > 1999
r3			<	20000	>	200	
..	..	..	..	..	..	..	..

G1 - Predicate Group 1 with predicates on 'Model'
G2 - Predicate Group 2 with predicates on 'Price'
G3 - Predicate Group 3 with predicates on 'HorsePower(Model, Year)'
Op - Predicate Operator
RHS - Constant right side of the predicate
Rid - Identifier of the row storing the corresponding expression
in the CONSUMER table

Empty cells indicate NULL values.

For each predicate group, the predicate table has two columns: one to store the operator of the predicate and the other to store the constant on the right-hand side of the predicate. For a predicate in an expression, its operator and the right-hand side constant are stored under the corresponding columns of the predicate group. The predicates that do not fall into one of the preconfigured groups are preserved in their original form and stored in a `VARCHAR2` column of the predicate table as **sparse predicates.** (For the example in Figure 2–1, the predicates on `Mileage` and `Year` fall in this category.) The predicates with `IN` lists and the predicates with a varying right-hand side (not a constant) are implicitly treated as sparse predicates. Native indexes are created on the predicate table as described in Section 2.4.

2.4 Index Processing

To evaluate a data item for a set of expressions, the left-hand side of each predicate group is computed and its value is compared with the corresponding constants stored in the predicate table using an appropriate operator. For example, using the predicate table, if `HORSEPOWER('TAURUS',2001)` returns 153, then the predicates satisfying this value are those interested in horsepower equal to 153 or those interested in horsepower greater than a value that is below 153, and so on. If the operators and right-hand side constants of the previous group are stored in the `G3_OP` and `G3_RHS` columns of the predicate table (in Figure 2–1), then the following query on the predicate table identifies the rows that satisfy this group of predicates:

```
SELECT  Rid  FROM predicate_table WHERE
     G3_OP = '=' AND G3_RHS = :rhs_val   or
     G3_OP = '>' AND G3_RHS < :rhs_val   or
     ...
-- where :rhs_val is the value from the computation of the left-hand side --
```

Expression Filter uses similar techniques for less than (<), greater than or equal to (>=), less than or equal to (<=), not equal to (`!=`, <>), `LIKE`, `IS NULL`, and `IS NOT NULL` predicates. Predicates with the `BETWEEN` operator are divided into two predicates with greater than or equal to and less than or equal to operators. Duplicate predicate groups can be configured for a left-hand side if it frequently appears more than once in a single expression, for example, `Year >= 1996 and Year <= 2000`.

The `WHERE` clause (shown in the previous query) is repeated for each predicate group in the predicate table, and the predicate groups are all joined by conjunctions. When the complete query (shown in the following example) is issued on the predicate table, it returns the row identifiers for the expressions that evaluate to true with all the predicates in the preconfigured groups. For these resulting expressions, the corresponding sparse predicates that are stored in the predicate table are evaluated using dynamic queries to determine if an expression is true for a particular data item.

```
SELECT Rid, Sparse_predicate FROM predicate_table
 WHERE                 --- predicates in group 1
  (G1_OP IS NULL OR     --- no predicate involving this LHS
   ((:g1_val IS NOT NULL AND
     (G1_OP = '=' AND G1_RHS = :g1_val or
      G1_OP = '>' AND G1_RHS < :g1_val or
      G1_OP = '<' AND G1_RHS > :g1_val or
      ...) or
     (:g1_val IS NULL AND G1_OP = 'IS NULL')))

 AND                  --- predicates in group 2
  (G2_OP IS NULL OR
   ((:g2_val IS NOT NULL AND
     (G2_OP = '=' AND G2_RHS = :g2_val   or
      G2_OP = '>' AND G2_RHS < :g2_val   or
      G2_OP = '<' AND G2_RHS > :g2_val   or
      ...) or
     (:g2_val IS NULL AND G2_OP = 'IS NULL')))
 AND
...
```

For efficient execution of the predicate table query (shown previously), concatenated bitmap indexes are created on the `{Operator, RHS constant}` columns of selected groups. These groups are identified either by user specification or from the statistics about the frequency of the predicates (belonging to a group) in the expression set. With the indexes defined on preconfigured predicate groups, the predicates from an expression set are divided into three classes:

1. Indexed predicates: Predicates that belong to a subset of the preconfigured predicate groups that are identified as most discriminating. Bitmap indexes are created for these predicate groups; thus, these predicates are also called indexed predicates. The previous query performs range scans on the corresponding index to evaluate all the predicates in a group and returns the expressions that evaluate to true with just that predicate. Similar scans are performed on the bitmap indexes of other indexed predicates, and the results from these index scans are combined using `BITMAP AND` operations to determine all the expressions that evaluate to true with all the indexed predicates. This enables multiple predicate groups to be filtered simultaneously using one or more bitmap indexes.
2. Stored predicates: Predicates that belong to groups that are not indexed. These predicates are captured in the corresponding `{Operator, RHS constant}` columns of the predicate table, with no bitmap indexes defined on them. For all the expressions that evaluate to true with the indexed predicates, the previous query compares the values of the left-hand sides of these predicate groups with those stored in the predicate table. Although bitmap indexes are created for a selected number of groups, the optimizer may choose not to use one or more indexes based on their access cost. Those groups are treated as stored predicate

groups. The query issued on the predicate table remains unchanged for a different choice of indexes.

3. Sparse predicates: Predicates that do not belong to any of the preconfigured predicate groups. For expressions that evaluate to true for all the predicates in the indexed and stored groups, sparse predicates (if any) are evaluated last. If the expressions with sparse predicates evaluate to true, they are considered true for the data item.

Optionally, you can specify the common operators that appear with predicates on the left-hand side and reduce the number of range scans performed on the bitmap index. In the previous example, the `Model` attribute commonly appears in equality predicates, and the Expression Filter index can be configured to check only for equality predicates while processing the indexed predicate groups. Sparse predicates along with any other form of predicate on the `Model` attribute are processed and evaluated at the same time.

2.5 Predicate Table Query

Once the predicate groups for an expression set are determined, the structure of the predicate table and the query to be issued on the predicate table are fixed. The choice of indexed or stored predicate groups does not change the query. As part of Expression Filter index creation, the predicate table query is determined and a function is dynamically generated for this query. The same query (with bind variables) is used for any data item passed in for the expression set evaluation. This ensures that the predicate table query is compiled once and reused for evaluating any number of data items.

2.6 Index Creation and Tuning

The cost of evaluating a predicate in an expression set depends on the group to which it belongs. The index for an expression set can be tuned by identifying the appropriate predicate groups as the index parameters.

The steps involved in evaluating the predicates in an indexed predicate group are:

- One-time computation of the left-hand side of the predicate group
- One or more range scans on the bitmap indexes using the computed value

The steps involved in evaluating the predicates in a stored predicate group are:

- One-time computation of the left-hand side of the predicate group

- Comparison of the computed value with the operators and the right-hand side constants of all the predicates remaining in the working set (after filtering, based on indexed predicates)

The steps involved in evaluating the predicates in a sparse predicate group are:

- Parse the subexpression representing the sparse predicates for all the expressions remaining in the working set.
- Evaluate the subexpression through substitution of data values (using a dynamic query).

Creating an Index from Default Parameters

In a schema, an attribute set can be used for one or more expression sets, and you can configure the predicate groups for these expression sets by associating the default index parameters with the attribute set. The (discriminating) predicate groups can be chosen with the knowledge of commonly occurring left-hand sides and their selectivity for the expected data.

The following command uses the DBMS_EXPFIL.DEFAULT_INDEX_ PARAMETERS procedure to configure default index parameters with the `Car4Sale` attribute set:

```
BEGIN
  DBMS_EXPFIL.DEFAULT_INDEX_PARAMETERS('Car4Sale',
    exf$attribute_list (
       exf$attribute (attr_name => 'Model',      --- LHS for predicate group
                      attr_oper => exf$indexoper('='),
                      attr_indexed => 'TRUE'),   --- indexed predicate group
       exf$attribute (attr_name => 'Price',
                      attr_oper => exf$indexoper('all'),
                      attr_indexed => 'TRUE'),
       exf$attribute (attr_name => 'HorsePower(Model, Year)',
                      attr_oper => exf$indexoper('=','<','>','>=','<='),
                      attr_indexed => 'FALSE')    --- stored predicate group
    )
  );
END;
/
```

For an expression set, create the Expression Filter index as follows:

```
CREATE INDEX InterestIndex ON Consumer (Interest)
              INDEXTYPE IS EXFSYS.EXPFILTER;
```

The index derives all its parameters from the defaults (`Model`, `Price`, and `HorsePower(Model, Year)`) associated with the corresponding attribute set. If the defaults are not specified, it implicitly uses all the scalar elementary attributes (`Model`, `Year`, `Price`, and `Mileage`) in the attribute set as its stored and indexed attributes.

You can fine-tune the default parameters derived from the attribute set for each expression set by using the `PARAMETERS` clause when you create the index or by associating index parameters directly with the expression set. The following `CREATE INDEX` statement with the `PARAMETERS` clause configures the index with an additional stored predicate:

```
CREATE INDEX InterestIndex ON Consumer (Interest)
              INDEXTYPE IS exfsys.ExpFilter
  PARAMETERS ('ADD TO DEFAULTS STOREATTRS (CrashTestRating(Model, Year))');
```

For more information about creating indexes from default parameters, see "DEFAULT_INDEX_PARAMETERS Procedure" in Chapter 8 and "CREATE INDEX" in Chapter 6.

Creating an Index from Exact Parameters

If there is a need to fine-tune the index parameters for each expression set associated with the common attribute set, you can assign the exact index parameters directly to the expression set, using the DBMS_EXPFIL.INDEX_PARAMETERS procedure.

The following commands copy the index parameters from the defaults and then fine-tune them for the given expression set. An expression filter index created for the expression set uses these parameters to configure its indexed and stored predicate groups.

```
BEGIN
  -- Derive index parameters from defaults --
  DBMS_EXPFIL.INDEX_PARAMETERS(expr_tab  => 'Consumer',
                               expr_col  => 'Interest',
                               attr_list => null,
                               operation => 'DEFAULT');

  -- Fine-tune the parameters by adding another stored attribute --
  DBMS_EXPFIL.INDEX_PARAMETERS(expr_tab  => 'Consumer',
                               expr_col  => 'Interest',
                               attr_list =>
                                exf$attribute_list (
                                 exf$attribute (
                                   attr_name => 'CrashTestRating(Model, Year)',
                                   attr_oper => exf$indexoper('all'),
                                   attr_indexed => 'FALSE')),
                               operation => 'ADD');
END;
/


CREATE INDEX InterestIndex ON Consumer (Interest)
             INDEXTYPE IS EXFSYS.EXPFILTER;
```

For more information about creating indexes from exact parameters, see "INDEX_ PARAMETERS Procedure" in Chapter 8 and "CREATE INDEX" in Chapter 6.

See Chapter 3 for a discussion on indexing expressions with XPath predicates.

Creating an Index from Statistics

If a representative set of expressions is already stored in the table, the owner of the table can automate the index tuning process by collecting statistics on the expression set, using the DBMS_EXPFIL.GET_EXPRSET_STATS procedure, and creating the index from these statistics, as shown in the following example:

```
BEGIN
  DBMS_EXPFIL.GET_EXPRSET_STATS (expr_tab => 'Consumer',
                                 expr_col => 'Interest');
END;
/

CREATE INDEX InterestIndex ON Consumer (Interest)
             INDEXTYPE IS EXFSYS.EXPFILTER
  PARAMETERS ('STOREATTRS TOP 4 INDEXATTRS TOP 2');
```

For the previous index, four stored attributes are chosen based on the frequency of the corresponding predicate left-hand sides in the expression set, and out of these four attributes, the top two are chosen as indexed attributes. When a `TOP` *n* clause is used, any defaults associated with the corresponding attribute set are ignored. The attributes chosen for an index can be viewed by querying the `USER_EXPFIL_PREDTAB_ATTRIBUTES` view.

For more information about creating indexes from statistics, see "GET_EXPRSET_ STATS Procedure" in Chapter 8 and "CREATE INDEX" in Chapter 6.

2.7 Index Usage

A query using the SQL `EVALUATE` operator on an Expression column can force the use of the index defined on such a column with an optimizer hint. (See the *Oracle Database Performance Tuning Guide*.) In other cases, the optimizer determines the cost of the Expression Filter index-based scan and compares it with the cost of alternate execution plans.

```
SELECT * FROM Consumer WHERE
  EVALUATE (Consumer.Interest,
            Car4Sale.getVarchar('Mustang',2000,18000,22000)) = 1 and
  Consumer.Zipcode BETWEEN 03060 and 03070;
```

For the previous query, if the `Consumer` table has an Expression Filter index defined on the `Interest` column and a native index defined on the `Zipcode` column, the optimizer chooses the appropriate index based on their selectivity and their access cost. In the current release, the selectivity and the cost of Expression Filter indexes are set as domain index defaults, and they are not computed for the expression set.

You can use the `EXPLAIN PLAN` statement to see if the optimizer picked the Expression Filter index for a query.

2.8 Index Storage and Maintenance

The Expression Filter index uses persistent database objects to maintain the index on a column storing expressions. All these secondary objects are created in the schema in which the Expression Filter index is created. There are three types of secondary objects for each Expression Filter index, and they use the following naming conventions:

- Conventional table called the predicate table: `EXF$PTAB_`*n*
- One or more indexes on the predicate table: `EXF$PTAB_`*n*`_IDX_`*m*
- Package called the Access Function package: `EXF$AFUN_`*n*

To ensure the expression evaluation is valid, a table with an Expression column and the Expression Filter index on the Expression column should belong to the same schema. A user with `CREATE INDEX` privileges on a table cannot create an Expression Filter index unless the user is the owner of the table. By default, the predicate table is created in the user's default tablespace. You can specify an alternate storage clause for the predicate table when you create the index by using the `PREDSTORAGE` parameter clause. (See the section about the CREATE INDEX statement in Chapter 6.) The indexes on the predicate table are always created in the same tablespace as the predicate table.

An Expression Filter index created for an Expression column is automatically maintained to reflect any changes made to the expressions (with the SQL INSERT, UPDATE, or DELETE statements or SQL*Loader). The bitmap indexes defined on the predicate table could become fragmented when a large number of expressions are modified, added to the set, or deleted. You can rebuild these indexes online to reduce the fragmentation using the DBMS_EXPFIL.DEFRAG_INDEX procedure, as shown in the following example:

```
BEGIN
  DBMS_EXPFIL.DEFRAG_INDEX (idx_name => 'InterestIndex');
END;
/
```

See "DEFRAG_INDEX Procedure" in Chapter 8 for more information about this procedure.

You can rebuild the complete Expression Filter index offline by using the ALTER INDEX ... REBUILD statement. This is useful when the index is marked UNUSABLE following a table maintenance operation. When the default index parameters associated with an attribute set are modified, they can be incorporated into the existing indexes using the ALTER INDEX ... REBUILD statement with the DEFAULT parameter clause. See the section about ALTER INDEX REBUILD statement in Chapter 6.

3 Expressions with XPath Predicates

The expressions stored in a column of a table may contain XPath predicates defined on XMLType attributes. This section describes an application for XPath predicates using the Car4Sale example introduced in Chapter 1. For this purpose, the information published for each car going on sale includes a Details attribute in addition to the Model, Price, Mileage, and Year attributes. The Details attribute contains additional information about the car in XML format as shown in the following example:

```
<details>
  <color>White</color>
  <accessory>
     <stereo make="Koss">CD</stereo>
     <GPS>
       <resolution>1FT</resolution>
       <memory>64MB</memory>
     </GPS>
  </accessory>
</details>
```

A sample predicate on the Details attribute is `extract(Details, '//stereo[@make="Koss"]') IS NOT NULL`. This predicate can be combined with one or more predicates on other XML or non-XML attributes.

3.1 Using XPath Predicates in Expressions

Using the Oracle supplied XMLType datatype, users can apply XPath predicates on XML documents within a standard SQL WHERE clause of a query. These predicates use operators such as EXTRACT and EXISTSNODE on an instance of the XMLType datatype to process an XPath expression for the XML instance. For more information, see *Oracle Database SQL Reference* and *Oracle XML DB Developer's Guide*.

To allow XPath predicates in an expression set, the corresponding attribute set should be created with an attribute of `sys.XMLType` datatype, as shown in the following example:

```
CREATE OR REPLACE TYPE Car4Sale AS OBJECT
                                   (Model   VARCHAR2(20),
                                    Year    NUMBER,
                                    Price   NUMBER,
                                    Mileage NUMBER,
                                    Details sys.XMLType);
/

BEGIN
  DBMS_EXPFIL.CREATE_ATTRIBUTE_SET(attr_set  => 'Car4Sale',
                                   from_type => 'YES');
END;
/
```

The expression sets using this attribute set can include predicates on the `XMLType` attribute, as shown in the following example:

```
Model='Taurus' and Price < 15000 and Mileage < 25000 AND
             extract(Details, '//stereo[@make="Koss"]') IS NOT NULL

                      -- or --

Model='Taurus' and Price < 15000 and Mileage < 25000 AND
             existsNode(Details, '//stereo[@make="Koss"]') = 1
```

Now, a set of expressions stored in the `Interest` column of the `Consumer` table can be processed for a data item by passing an instance of `XMLType` for the `Details` attribute along with other attribute values to the `EVALUATE` operator:

```
SELECT * FROM Consumer WHERE
    EVALUATE (Consumer.Interest,
              'Model=>''Mustang'',
               Year=>2000,
               Price=>18000,
               Mileage=>22000,
               Details=>sys.XMLType(''<details>
                                       <color>White</color>
                                       <accessory>
                                        <stereo make="Koss">CD</stereo>
                                        <GPS>
                                         <resolution>1FT</resolution>
```

```
                                                              <memory>64MB</memory>
                                                            </GPS>
                                                          </accessory>
                                                        </details>'')'
          ) = 1;
```

The previous query identifies all the rows with expressions that are true based on their XPath and non-XPath predicates.

3.2 Indexing XPath Predicates

To process a large set of XPath predicates in an expression set efficiently, the Expression Filter index defined for the expression set can be configured for the XPath predicates (in addition to some simple predicates). The Expression Filter indexes use the commonalities in the XPath expressions to efficiently compare them to a data item. These commonalities are based on the positions and the values for the XML elements and attributes appearing in the XPath expressions.

The indexable constructs in an XPath expression are the levels (or positions) of XML elements, the values for text nodes in XML elements, the positions of XML attributes, and the values for XML attributes. For this purpose, an XPath predicate is treated as a combination of positional and value filters on XML elements and attributes appearing in an XML document. For example, the following XPath expression can be deciphered as a set of checks on the XML document. The list following the example explains those checks.

```
extract(Details, '//stereo[@make="Koss" and /*/*/GPS/memory[text()="64MB"]]')
                                                                  IS NOT NULL
```

1. Level (position) of `stereo` element is 1 or higher.
2. The `stereo` element appearing at level 1 or higher has a `make` attribute.
3. The value for `stereo` element's `make` attribute is `Koss`.
4. The `GPS` element appears at level 3.
5. The `memory` element appears at level 4.
6. The `memory` element has a text node with a value of `64MB`.

3.2.1 Indexable XPath Predicates

The Expression Filter index does not support some constructs in an XPath predicate. Therefore, the XPath predicate is always included in the sparse predicates and evaluated during last phase of expression filtering. For more information about sparse predicates, see Section 2.4.

A positional filter for an Expression Filter index can be configured from any XML element or attribute. A value filter can only be configured from equality predicates on XML attributes and text nodes in XML elements. XPath predicates that are indexed in an expression set must use either the EXTRACT or the EXISTSNODE operator with a positive test on the return value. For example `extract(Details, '//stereo[@make="Koss"]') IS NOT NULL` can be indexed, but a similar predicate with an `IS NULL` check on the return value cannot be indexed.

Some of the XPath constructs that cannot be indexed by the Expression Filter include:

- Inequality or range predicates in the node test. For example, the predicate on the `stereo` element's `make` attribute cannot be indexed in the following XPath predicate:

  ```
  extract(Details, '//stereo[@make!="Koss"]') IS NOT NULL
  ```

- Disjunctions in the node test. For example, the predicates on the `stereo` element's `make` attribute cannot be indexed in the following XPath predicate:

  ```
  extract(Details, '//stereo[@make="Koss" or @make="Bose"]') IS NOT NULL
  ```

- Node tests using XML functions other than `text()`. For example, the predicate using the XML function, `position`, cannot be indexed.

  ```
  extract(Details, '//accessory/stereo[position()=3]') IS NOT NULL
  ```

  However, the `text()` function in the following example can be a value filter on the `stereo` element:

  ```
  extract(Details, '//accessory/stereo[text()="CD"]') IS NOT NULL
  ```

- Duplicate references to an XML element or an attribute within a single XPath expression. For example, if the `stereo` element appears in an XPath expression at two different locations, only the last occurrence is indexed, and all other references are processed during sparse predicate evaluation.

3.2.2 Index Representation

The Expression Filter index can be configured to process the XPath predicates efficiently by using the most discriminating XML elements and attributes as positional and value filters. Each one forms a predicate group for the expression set.

For the purpose of indexing XPath predicates, the predicate table structure described in Section 2.3 is extended to include two columns for each XML tag. For an XML tag configured as positional filter, these columns capture the relative and absolute positions of the tag in various XPath predicates. For an XML tag configured as value filter, these columns capture the constants appearing with the tag in the node tests and their relational operators. (Only equality operators are indexed in this release.)

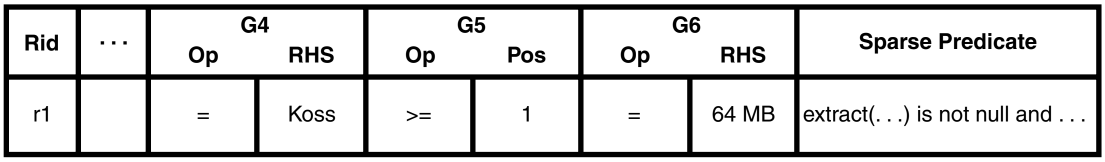
Figure 3–1 shows the predicate table structure for the index configured with the following XML tags:

- XML attribute `stereo@make` as value filter. (Predicate Group 4 - G4)
- XML element `stereo` as positional filter. (Predicate Group 5 - G5)
- Text node of the XML element `memory` as value filter. (Predicate Group 6 - G6)

This image can be viewed as an extension of the predicate table shown in Figure 2–1. The partial row shown in the predicate table captures the following XPath predicate:

```
extract(Details, '//stereo[@make="Koss" and /*/*/GPS/memory[text()="64MB"]]')
                                                                  IS NOT NULL
```

Figure 3–1 Conceptual Predicate Table with XPath Predicates



Rid	. . .	G4		G5		G6		Sparse Predicate
		Op	RHS	Op	Pos	Op	RHS	
r1		=	Koss	>=	1	=	64 MB	extract(. . .) is not null and . . .

3.2.3 Index Processing

The XPath predicates captured in the predicate table are compared to an XML document that is included in the data item passed to the `EVALUATE` operator. The positions and values of the XML tags used in the index are computed for the XML document, and these are compared with the values stored in the corresponding columns of the predicate table. Assuming that the relational operators and the right-hand-side constants for the value filter on `stereo@make` attribute are stored in `G4_OP` and `G4_RHS` columns of the predicate table (Figure 3.1), the following query on the predicate table identifies the rows that satisfy this check for an XML document:

```
SELECT Rid FROM predicate_table
```

```
  WHERE G4_OP = '=' AND
        G4_RHS in (SELECT column_value FROM TABLE (:G4ValuesArray))
```

For the previous query, the values for all the occurrences of the `stereo@make` attribute in the given XML document are represented as a VARRAY and bound to the `:G4ValuesArray` variable.

Similarly, assuming that the position constraints and the absolute levels (positions) of the `stereo` element are stored in the `G5_OP` and `G5_POS` columns of the predicate table, the following query identifies all the rows that satisfy these positional checks for an XML document:

```
SELECT Rid FROM predicate_table
    WHERE (G5_OP = '=' AND                     --- absolute position check --
           G5_POS in (SELECT column_value FROM table (:G5PosArray))) OR
           (G5_OP = '>=' AND                   --- relative position check --
            G5_POS <= SELECT max(column_value) FROM table (:G5PosArray)))
```

For the previous query, the `:G5PosArray` contains the levels for all the occurrences of the `stereo` element in the XML document. These checks on each predicate group can be combined with the checks on other (XPath and non-XPath) predicate groups to form a complete predicate table query. A subset of the XML tags can be identified as the most selective predicate groups, and they can be configured as the indexed predicate groups (See Section 2.4). Bitmap indexes are created for the selective predicate groups, and these indexes are used along with indexes defined for other indexed predicate groups to efficiently process the predicate table query.

3.2.4 Index Tuning for XPath Predicates

The most discriminating XML tags in a set of XPath predicates are classified as positional filters and value filters. A value filter is considered discriminating if node tests using the XML tag are selective enough to match only a subset of XML documents. Similarly, a positional filter is considered discriminating if the tag appears at different levels or does not appear in all XML documents, and thus match only a subset of them.

The XPath positional and value filters can be further mapped to indexed predicate groups or stored predicate groups. PL/SQL procedures are provided to configure an Expression Filter index with these parameters. For an attribute set consisting of two or more `XMLType` attributes, the XML tags can be associated with each of these attributes

The XPath index parameters for a set of expressions are considered part of the index parameter, and they can be assigned to an attribute set or an expression set (the column storing the expressions). The index parameters assigned to the attribute set act as defaults and are shared across all the expression sets associated with the attribute set.

A few XPath index parameters can be assigned to an `XMLType` attribute of an attribute set using the DMBS_EXPFIL.DEFAULT_XPINDEX_PARAMETERS procedure, as shown in the following example:

```
BEGIN
  DBMS_EXPFIL.DEFAULT_XPINDEX_PARAMETERS(
       attr_set   => 'Car4Sale',
       xmlt_attr  => 'Details',                         --- XMLType attribute
       xptag_list =>                                    --- Tag list
         exf$xpath_tags(
           exf$xpath_tag(tag_name    => 'stereo@make',  --- XML attribute
                         tag_indexed => 'TRUE',
                         tag_type    => 'VARCHAR(15)'), --- value filter
           exf$xpath_tag(tag_name    => 'stereo',       --- XML element
                         tag_indexed => 'FALSE',
                         tag_type    => null),   --- null => positional filter
           exf$xpath_tag(tag_name    => 'memory',       --- XML element
                         tag_indexed => 'TRUE',
                         tag_type    => 'VARCHAR(10)')  --- value filter
          )
        );
END;
/
```

Note that a missing or null value for the `tag_type` argument configures the XML tag as a positional filter.

For more information about assigning XPath index parameters, see "DEFAULT_XPINDEX_PARAMETERS Procedure" in Chapter 8.

By default, the previous XPath index parameters are used for any index created on an expression set that is associated with the `Car4Sale` attribute set.

```
CREATE INDEX InterestIndex ON Consumer (Interest)
        INDEXTYPE IS EXFSYS.EXPFILTER;
```

Unlike simple index parameters, the XPath index parameters cannot be fine-tuned for an expression set when the index is created. However, you can achieve this by associating index parameters directly with the expression set using the DBMS_EXPFIL.INDEX_PARAMETERS and DBMS_EXPFIL.XPINDEX_PARAMETERS procedures and then creating the index, as shown in the following example:

```
BEGIN
  -- Derive the index parameters including XPath index params from defaults --
  DBMS_EXPFIL.INDEX_PARAMETERS(expr_tab  => 'Consumer',
                               expr_col  => 'Interest',
                               attr_list => null,
                               operation => 'DEFAULT');

  -- fine-tune the XPath index parameters by adding another Tag --
  DBMS_EXPFIL.XPINDEX_PARAMETERS(expr_tab  => 'Consumer',
                                 expr_col  => 'Interest',
                                 xmlt_attr => 'Details',
                                 xptag_list =>
                                   exf$xpath_tags(
                                     exf$xpath_tag(tag_name    => 'GPS',
                                                   tag_indexed => 'TRUE',
                                                   tag_type    => null)),
                                 operation => 'ADD');
END;
/


CREATE INDEX InterestIndex ON Consumer (Interest)
            INDEXTYPE IS EXFSYS.EXPFILTER;
```

For more information, see "INDEX_PARAMETERS Procedure" and "XPINDEX_PARAMETERS Procedure" in Chapter 8.

Once the index is created on a column storing the expressions, a query with the `EVALUATE` operator can process a large set of XPath and non-XPath predicates for a data item efficiently:

```
SELECT * FROM Consumer WHERE
    EVALUATE (Consumer.Interest,
              'Model=>''Mustang'',
               Year=>2000,
               Price=>18000,
               Mileage=>22000,
               Details=>sys.XMLType(''<details>
                                        <color>White</color>
                                        <accessory>
                                         <stereo make="Koss">CD</stereo>
                                         <GPS>
                                          <resolution>1FT</resolution>
                                          <memory>64MB</memory>
                                         </GPS>
```

```
                                                </accessory>
                                               </details>'')'
              ) = 1;
```

Expression Filter index tuning based on XPath statistics is not supported in the current release.

4 Expression Filter Internal Objects

The Expression Filter feature uses schema objects to maintain an Expression column in a user table. Most of these objects are created in the schema of the table with the Expression column. These objects are created with the `EXF$` prefix and are maintained using the Expression Filter APIs. The user should not modify these objects.

4.1 Attribute Set Object Type

The Expression Filter maintains the concept of an attribute set through an object type with a matching name. The object type used for an attribute set may not contain any user methods, and it should not be an evolved type (with the use of `ALTER TYPE` command). If the attribute set is not created from an existing object type, Expression Filter creates the object type with the matching name and maintains it throughout the life of the attribute set. It also generates functions for the object type for data item management, dynamic expression evaluation, and expression type checking.

In addition to the object type, Expression Filter creates a nested table type of the object type in the same schema. This nested table type uses a namespace `EXF$NTT_`*`n`*, and it is used internally for the expression validation.

The object type created for the attribute set can be used to create a table storing the corresponding data items. Such tables could include a column of the object type or the table itself could be created from the object type. These tables can be joined with the table storing expressions. This is shown in the following example using the application example in Chapter 1:

```
-- a table of type --
CREATE TABLE CarInventory OF Car4Sale;

INSERT INTO CarInventory VALUES ('Mustang',2000, 18000, 22000);
INSERT INTO CarInventory VALUES ('Mustang',2000, 18000, 22000);
INSERT INTO CarInventory VALUES ('Taurus',1997, 14000, 24500);

SELECT * FROM Consumer, CarInventory Car WHERE
   EVALUATE (Consumer.Interest, Car.getVarchar()) = 1;

-- table with the object type column --
CREATE TABLE CarStock (CarId NUMBER, Details Car4Sale);

INSERT INTO CarStock VALUES (1, Car4Sale('Mustang',2000, 18000, 22000));
INSERT INTO CarStock VALUES (2, Car4Sale('Mustang',2000, 18000, 22000));
INSERT INTO CarStock VALUES (3, Car4Sale('Taurus',1997, 14000, 24500));

SELECT * FROM Consumer, CarStock Car WHERE
  EVALUATE (Consumer.Interest, Car.Details.getVarchar()) = 1;
```

You should not modify the object type used to maintain an attribute set with the `ALTER TYPE` or `CREATE OR REPLACE TYPE` commands. System triggers are used to restrict you from modifying these objects.

4.2 Expression Validation Trigger

When an Expression column is created by assigning an attribute set to a `VARCHAR2` column in a user table, a `BEFORE ROW` trigger is created on the table. This trigger is used to invoke the expression validation routines when a new expression is added or an existing expression is modified. This trigger is always created in the `EXFSYS` schema, and it uses the `EXF$VALIDATE_`*`n namespace`*.

4.3 Expression Filter Index Objects

The Expression Filter index defined for a column is maintained using database objects created in the schema in which the index is created. These are described in Section 2.8.

4.4 Expression Filter System Triggers

Expression Filter uses system triggers to manage the integrity of the system. These include system triggers to restrict the user from dropping an object type created by an attribute set, to drop the attribute set and associated metadata when the user is dropped with a `CASCADE` option, and to maintain the Expression Filter dictionary through `DROP` and `ALTER` operations on the table with one or more Expression columns. These triggers are created in the `EXFSYS` schema.

5

Using Expression Filter with Utilities

This chapter describes the use of SQL*Loader and Data Pump Export/Import utilities in the presence of one or more Expression columns.

5.1 Bulk Loading of Expression Data

Bulk loading can import large amounts of ASCII data into an Oracle database. You use the SQL*Loader utility to bulk load data.

For SQL*Loader operations, the expression data is treated as strings loaded into a `VARCHAR2` column of a database table. The data file can hold the expression data in any format allowed for `VARCHAR2` data, and the control file can refer to the column storing expressions as a column of a `VARCHAR2` datatype.

A sample control file used to load a few rows into the `Consumer` table is shown in the following example:

```
LOAD DATA
INFILE *
INTO TABLE Consumer
FIELDS TERMINATED BY ',' OPTIONALLY ENCLOSED BY '"'
(CId, Zipcode, Phone, Interest)
BEGINDATA
1,32611,"917 768 4633","Model='Taurus' and Price < 15000 and Mileage < 25000"
2,03060,"603 983 3464","Model='Mustang' and Year > 1999 and Price < 20000"
3,03060,"603 484 7013","HorsePower(Model, Year) > 200 and Price < 20000"
```

The data loaded into an Expression column is automatically validated using the attribute set associated with the column. This validation is done by the `BEFORE ROW` trigger defined on the column storing expressions. Therefore, a direct load cannot be used when the table has one or more Expression columns.

If an Expression Filter index is defined on the column storing expressions, it is automatically maintained during the SQL*Loader operations.

To achieve faster bulk loads, the expression validation can be bypassed by following these steps:

1. Drop any Expression Filter indexes defined on Expression columns in the table:

   ```
   DROP INDEX InterestIndex;
   ```

2. Convert the Expression columns back into `VARCHAR2` columns by unassigning the attribute sets, using the UNASSIGN_ATTRIBUTE_SET procedure:

   ```
   BEGIN
     DBMS_EXPFIL.UNASSIGN_ATTRIBUTE_SET (expr_tab => 'Consumer',
                                         expr_col => 'Interest');
   END;
   /
   ```

3. Perform the bulk load operation. Because the Expression columns are converted to `VARCHAR2` columns in the previous step, a direct load is possible in this step.

4. Convert the `VARCHAR2` columns with expression data into Expression columns by assigning a value of `TRUE` for the `force` argument of the ASSIGN_ATTRIBUTE_SET procedure:

   ```
   BEGIN
     DBMS_EXPFIL.ASSIGN_ATTRIBUTE_SET (
                                       attr_set => 'Car4Sale',
                                       expr_tab => 'Consumer',
                                       expr_col => 'Interest',
                                       force    => 'TRUE');
   END;
   /
   ```

5. To avoid runtime validation errors, the expressions in the table can be validated using the DBMS_EXPFIL.VALIDATE_EXPRESSIONS procedure:

   ```
   BEGIN
     DBMS_EXPFIL.VALIDATE_EXPRESSIONS (expr_tab => 'Consumer',
                                        expr_col => 'Interest');
   END;
   /
   ```

6. Re-create the indexes on the Expression columns.

5.2 Exporting and Importing Tables, Users, and Databases

A table with one or more Expression columns can be exported and imported back to the same database or a different Oracle database. If a table with Expression columns is being imported into an Oracle database, ensure Expression Filter is installed.

5.2.1 Exporting and Importing Tables Containing Expression Columns

When a table with one or more Expression columns is exported, the corresponding attribute set definitions, along with their object type definitions, are placed in the export dump file. An attribute set definition placed in the dump file includes its default index parameters and the list of approved user-defined functions. However, definitions for the user-defined functions are not placed in the export dump file.

While importing a table with one or more Expression columns from the export dump file, the attribute set creation may fail if a matching attribute set exists in the destination schema. If the attribute set is defined with one or more (embedded) object typed attributes, these types should exist in the database importing the attribute set. While importing the default index parameters and user-defined function list, the import driver continues the import process if it encounters missing dependent objects. For example, if the function `HorsePower` does not exist in the schema importing the `Consumer` table, the import of the table and the attribute set proceeds without errors. However, the corresponding entries in the Expression Filter dictionary display null values for object type or output datatype fields, an indication the import process was incomplete.

When the Expression Filter index defined on an Expression column is exported, all its metadata is placed in the export dump file. This metadata includes a complete list of stored and indexed attributes configured for the index. During import, this list is used. The attributes are not derived from the default index parameters. If one or more stored attributes use object references (functions) that are not valid in the schema importing the index, the index creation fails with an error. However, the index metadata is preserved in the Expression Filter dictionary.

A table imported incompletely due to broken references to dependent schema objects (in the function list, default index parameters list, and exact index parameters list) may cause runtime errors during subsequent expression evaluation or expression modifications (through DML). Import of such tables can be completed from a SQL*Plus session by resolving all the broken references. Running the Expression Validation utility (`DBMS_EXPFIL.VALIDATE_EXPRESSIONS` procedure) can identify errors in the expression metadata and the expressions. You can create any missing objects identified by this utility and repeat the process until all the errors in the expression set are resolved. Then, you can recover the Expression Filter index with the `ALTER INDEX ... REBUILD` statement.

5.2.2 Exporting a User Owning Attribute Sets

In addition to exporting tables and indexes defined in the schema, export of a user places the definitions for attribute sets that are not associated with any Expression column into the export dump file. All the restrictions that apply to the export of tables also apply to the export of a user.

5.2.3 Exporting a Database Containing Attribute Sets

During a database export, attribute set definitions are placed in the export file along with all other objects. The contents of `EXFSYS` schema are excluded from the database export.

6

SQL Operators and Statements

This chapter provides reference information about the SQL `EVALUATE` operator and SQL statements used to index expression data. Table 6–1 lists the statements and their descriptions. For complete information about SQL statements, see *Oracle Database SQL Reference*.

Table 6–1 Expression Filter Index Creation and Usage Statements

Statement	Description
`EVALUATE`	Matches an expression set with a given data item or table of data items
`ALTER INDEX REBUILD`	Rebuilds an Expression Filter index
`ALTER INDEX RENAME TO`	Changes the name of an Expression Filter index
`CREATE INDEX`	Creates an Expression Filter index on a column storing expressions
`DROP INDEX`	Drops an Expression Filter index

EVALUATE

The EVALUATE operator is used in the WHERE clause of a SQL statement to compare stored expressions to incoming data items.

The expressions to be evaluated are stored in an Expression column, which is created by assigning an attribute set to a VARCHAR2 column in a user table.

Format

```
EVALUATE (expression_column, <dataitem>)

<dataitem> := <varchar_dataitem> | <anydata_dataitem>
<varchar_dataitem> := attribute_name => attribute_value
                      {, attribute_name => attribute_value}
<anydata_dataitem> := AnyData.convertObject(attribute_set_instance)
```

Keywords and Parameters

expression_column
Name of the column storing the expressions.

attribute_name
Name of an attribute from the corresponding attribute set.

attribute_value
Value for the attribute.

attribute_set_instance
Instance of the object type associated with the corresponding attribute set.

Returns

The EVALUATE operator returns a 1 for an expression that matches the data item, and returns a 0 for an expression that does not match the data item. For any null values stored in the Expression column, the EVALUATE operator returns NULL.

Usage Notes

The EVALUATE operator can be used in the WHERE clause of a SQL statement. When an Expression Filter index is defined on a column storing expressions, the EVALUATE operator on such column may use the index for the expression set evaluation based on its usage cost. The EVALUATE operator can be used as a join predicate between a table storing expressions and a table storing the corresponding data items.

If the values of all elementary attributes in the attribute set can be represented as readable values, such as those stored in VARCHAR, DATE, and NUMBER datatypes and the constructors formatted as a string, then the data item can be formatted as a string of attribute name-value pairs. If a data item does not require a constructor for any of its elementary attribute values, then a list of values provided for the data item can be formatted as a string of name-value pairs using two getVarchar methods (a STATIC method and a MEMBER method) in the object type associated with the attribute set.

Any data item can be formatted using an AnyData instance. An attribute set with one or more binary typed attributes must use the AnyData form of the data item.

See "Applying the SQL EVALUATE Operator" on page 1-10 for more information about the EVALUATE operator.

Related views: USER_EXPFIL_ATTRIBUTE_SETS, USER_EXPFIL_ATTRIBUTES, and USER_EXPFIL_EXPRESSION_SETS

Examples

The following query uses the VARCHAR form of the data item generated by the getVarchar() function:

```
SELECT * FROM Consumer WHERE
    EVALUATE (Consumer.Interest,
               Car4Sale('Mustang',
                         2000,
                         18000,
                         22000).getVarchar()
               ) = 1;
```

For the previous query, the data item can be passed in the AnyData form with the following syntax:

```
SELECT * FROM Consumer WHERE
    EVALUATE (Consumer.Interest,
              AnyData.convertObject (
                      Car4Sale ('Mustang',
                                 2000,
                                 18000,
                                 22000)
              )) = 1;
```

When a large set of data items are stored in a table, the table storing expressions can be joined with the table storing data items with the following syntax:

```
SELECT i.CarId, c.CId, c.Phone
FROM Consumer c, Inventory i
WHERE
     EVALUATE (c.Interest,
               Car4Sale(i.Model, i.Year, i.Price, i.Mileage).getVarchar()) = 1
ORDER BY i.CarId;
```

ALTER INDEX REBUILD

The ALTER INDEX REBUILD statement rebuilds an Expression Filter index created on a column storing expressions. The Expression Filter index DOMIDX_OPSTATUS status in the USER_INDEXES view must be VALID for the rebuild operation to succeed.

Format

```
ALTER INDEX [schema_name.]index_name REBUILD
 [PARAMETERS ('DEFAULT')]
```

Keywords and Parameters

DEFAULT
The list of stored and indexed attributes for the Expression Filter index being rebuilt are derived from the default index parameters associated with the corresponding attribute set.

Usage Notes

When the ALTER INDEX ... REBUILD statement is issued without a PARAMETERS clause, the Expression Filter index is rebuilt using the current list of stored and indexed attributes. This statement can also be used for indexes that failed during IMPORT operation due to missing dependent objects.

The default index parameters associated with an attribute set can be modified without affecting the existing Expression Filter indexes. These indexes can be rebuilt to use the new set of defaults by using the DEFAULT parameter with the ALTER INDEX ... REBUILD statement. Index parameters assigned to the expression set are cleared when an index is rebuilt using the defaults.

The bitmap indexes defined for the indexed attributes of an Expression Filter index get fragmented as the expressions stored in the corresponding column are frequently modified (using INSERT, UPDATE, or DELETE operations). Rebuilding those indexes could improve the performance of the query using the EVALUATE operator. The bitmap indexes can be rebuilt online using the DBMS_ EXPFIL.DEFRAG_INDEX procedure.

See "Index Storage and Maintenance" on page 2-10 for more information about rebuilding indexes.

Related views: USER_EXPFIL_INDEXES and USER_EXPFIL_PREDTAB_
ATTRIBUTES

Examples

The following statement rebuilds the index using its current parameters:

```
ALTER INDEX InterestIndex REBUILD;
```

The following statement rebuilds the index using the default index parameters associated with the corresponding attribute set:

```
ALTER INDEX InterestIndex REBUILD PARAMETERS('DEFAULT');
```

ALTER INDEX RENAME TO

The ALTER INDEX RENAME TO statement renames an Expression Filter index.

Format

```
ALTER INDEX [schema_name.]index_name RENAME TO new_index_name;
```

Keywords and Parameters

None.

Usage Notes

None.

Examples

The following statement renames the index:

```
ALTER INDEX InterestIndex RENAME TO ExprIndex;
```

CREATE INDEX

The CREATE INDEX statement creates an Expression Filter index for a set of expressions stored in a column. The column being indexed should be configured to store expressions (with an attribute set assigned to it), and the index should be created in the same schema as the table (storing expressions).

Format

```
CREATE INDEX [schema_name.]index_name ON
[schema_name.].table_name (column_name) INDEXTYPE IS EXFSYS.EXPFILTER
[ PARAMETERS (' <parameters_clause> ' ) ...;
<parameters_clause>:= [ADD TO DEFAULTS | REPLACE DEFAULTS]
             [<storeattrs_clause>] [<indexattrs_clause>][<predstorage_clause>]
<storeattrs_clause>  :=  STOREATTRS [ ( attr1, attr2, ..., attrx   ) | TOP n ]
<indexattrs_clause>  :=  INDEXATTRS [ ( attr1, attr2, ..., attry   ) | TOP m ]
<predstorage_clause> := PREDSTORAGE (<storage_clause>)
```

Keywords and Parameters

EXFSYS.EXPFILTER
The name of the index type that implements the Expression Filter index.

ADD TO DEFAULTS
When this parameter is specified, the attributes listed in the STOREATTRS and INDEXATTRS clauses are added to the defaults associated with the corresponding attribute set. This is the default behavior.

REPLACE DEFAULTS
When this parameter is specified, the index is created using only the list of stored and indexed attributes specified after this clause. In this case, the default index parameters associated with the corresponding attribute set are ignored.

STOREATTRS
Parameter to list the stored attributes for the Expression Filter index.

INDEXATTRS
Parameter to list the indexed attributes for the Expression Filter index.

TOP

This parameter can be used for both STOREATTRS and INDEXATTRS clauses only when expression set statistics are collected. (See the section about GET_EXPRSET_STATS Procedure in Chapter 8.) The number after the TOP parameter indicates the number of (the most-frequent) attributes to be stored or indexed for the Expression Filter index.

PREDSTORAGE

Storage clause for the predicate table. See *Oracle Database SQL Reference* for the <storage_clause> definition.

Usage Notes

When the index parameters are directly assigned to an expression set (column storing expressions), the PARAMETERS clause in the CREATE INDEX statement cannot contain STOREATTRS or INDEXATTRS clauses. In this case, the Expression Filter index is always created using the parameters associated with the expression set. (See the "INDEX_PARAMETERS Procedure" and "XPINDEX_PARAMETERS Procedure" sections in Chapter 8 and the "USER_EXPFIL_INDEX_PARAMS View" in Chapter 9.)

When the PARAMETERS clause is not used with the CREATE INDEX statement and the index parameters are not assigned to the expression set, the default index parameters associated with the corresponding attribute set are used for the Expression Filter index. If the default index parameters list is empty, all the scalar attributes defined in the attribute set are stored and indexed in the predicate table.

For an Expression Filter index, all the indexed attributes are also stored. So, the list of stored attributes is derived from those listed in the STOREATTRS clause and those listed in the INDEXATTRS clause. If REPLACE DEFAULTS clause is not specified, this list is merged with the default index parameters associated with the corresponding attribute set.

If the REPLACE DEFAULTS clause is not specified, the list of indexed attributes for an Expression Filter index is derived from the INDEXATTRS clause and the default index parameters associated with the corresponding attribute set. If this list is empty, the system picks at most 10 stored attributes and indexes them.

If an attribute is listed in the PARAMETERS clause as well as the default index parameters, its stored versus indexed property is decided by the PARAMETERS clause specification.

Predicate statistics for the expression set should be available to use the TOP clause in the parameters of the CREATE INDEX statement. (See the "GET_EXPRSET_

STATS Procedure" in Chapter 8 for more information.) When the `TOP` clause is used for the `STOREATTRS` parameter, the `INDEXATTRS` parameter (if specified) should also use the `TOP` clause. Also, the number specified for the `TOP` clause of the `INDEXATTRS` parameter should be less than or equal to the one specified for the `STOREATTRS` parameter. When a `TOP` clause is used, `REPLACE DEFAULTS` usage is implied. That is, the stored and indexed attributes are picked solely based on the predicate statistics available in the dictionary.

The successful creation of the Expression Filter index creates a predicate table, one or more bitmap indexes on the predicate table, and a package with access functions in the same schema as the base table. By default the predicate table and its indexes are created in the user default tablespace. Alternate tablespace and other storage parameters for the predicate table can be specified using the `PREDSTORAGE` clause. The indexes on the predicate table are always created in the same tablespace as the predicate table.

See Chapter 2 for information about indexing expressions.

Related views: USER_EXPFIL_INDEXES, USER_EXPFIL_INDEX_PARAMETERS, USER_EXPFIL_DEF_INDEX_PARAMS, USER_EXPFIL_EXPRSET_STATS, and USER_EXPFIL_PREDTAB_ATTRIBUTES

Examples

When index parameters are not directly assigned to the expression set, you can create an Expression Filter index using the default index parameters specified for the corresponding attribute set as follows:

```
CREATE INDEX InterestIndex ON Consumer (Interest) INDEXTYPE IS EXFSYS.EXPFILTER;
```

You can create an index with one additional stored attribute using the following statement:

```
CREATE INDEX InterestIndex ON Consumer (Interest) INDEXTYPE IS EXFSYS.EXPFILTER
  PARAMETERS ('STOREATTRS (CrashTestRating(Model, Year))
               PREDSTORAGE (tablespace tbs_1) ');
```

You can specify the complete list of stored and indexed attributes for an index with the following statement:

```
CREATE INDEX InterestIndex ON Consumer (Interest) INDEXTYPE IS EXFSYS.EXPFILTER
  PARAMETERS ('REPLACE DEFAULTS
               STOREATTRS (Model, CrashTestRating(Model, Year))
               INDEXATTRS (Model, Year, Price)
               PREDSTORAGE (tablespace tbs_1) ');
```

The TOP clause can be used in the parameters clause when statistics are computed for the expression set. These statistics are accessible from the USER_EXPFIL_EXPRSET_STATS view.

```
BEGIN
  DBMS_EXPFIL.GET_EXPRSET_STATS (expr_tab => 'Consumer',
                                 expr_col => 'Interest');
END;
/

CREATE INDEX InterestIndex ON Consumer (Interest) INDEXTYPE IS EXFSYS.EXPFILTER
  PARAMETERS ('STOREATTRS TOP 4 INDEXATTRS TOP 3');
```

DROP INDEX

The DROP INDEX statement drops an Expression Filter index.

Format

```
DROP INDEX [schema_name.]index_name;
```

Keyword and Parameters

None.

Usage Notes

Dropping an Expression Filter index automatically drops all the secondary objects maintained for the index. These objects include a predicate table, one or more indexes on the predicate table, and an access function package.

Examples

```
DROP INDEX InterestIndex;
```

7 Object Types

The Expression Filter feature is supplied with a set of predefined types and public synonyms for these types. Most of these types are used for configuring index parameters with the Expression Filter procedural APIs. The `EXF$TABLE_ALIAS` type is used to support expressions defined on one or more database tables.

All the values and names passed to the types defined in this chapter are not case sensitive. To preserve the case, you use double quotation marks around the values.

EXF$ATTRIBUTE

The EXF$ATTRIBUTE type is used to handle stored and indexed attributes for the Expression Filter indexes.

Attributes

Name	Datatype	Description
attr_name	VARCHAR2(350)	The arithmetic expression that constitutes the stored or indexed attribute.
attr_oper	EXF$INDEXOPER	The list of common operators in the predicates with the attribute. Default value: EXF$INDEXOPER('all')
attr_ indexed	VARCHAR2(5)	TRUE if the attribute is indexed, else FALSE. Default value: FALSE.

Usage Notes

The EXF$ATTRIBUTE type is used to specify the stored and indexed attributes for an Expression Filter index using the DBMS_EXPFIL.DEFAULT_INDEX_PARAMETERS procedure. When values for attr_oper and attr_indexed fields are omitted during EXF$ATTRIBUTE instantiation, it is considered a stored attribute with a default value for common operators (EXF$INDEXOPER('all')).

Examples

A stored attribute with no preference on the list of common operators is represented as follows:

```
exf$attribute (attr_name => 'HorsePower(Model, Year)')
```

An indexed attribute is represented as follows:

```
exf$attribute (attr_name => 'HorsePower(Model, Year)',
               attr_indexed => 'TRUE')
```

An indexed attribute with a list of common operators is represented as follows:

```
exf$attribute (attr_name => 'HorsePower(Model, Year)',
               attr_oper => exf$indexoper('=','<','>','>=','<='),
               attr_indexed => 'TRUE')
```

EXF$ATTRIBUTE_LIST

The `EXF$ATTRIBUTE_LIST` type is defined as follows:

```
CREATE or REPLACE TYPE exf$attribute_list as VARRAY(490) of exf$attribute;
```

Attributes

None.

Usage Notes

The `EXF$ATTRIBUTE_LIST` type is used to specify a list of stored and indexed attributes while configuring the index parameters. (Also see the "DEFAULT_INDEX_PARAMETERS Procedure" in Chapter 8 for more information.)

Examples

A list of stored and indexed attributes can be represented as follows:

```
exf$attribute_list (
       exf$attribute (attr_name => 'Model',
                      attr_oper => exf$indexoper('='),
                      attr_indexed => 'TRUE'),
       exf$attribute (attr_name => 'Price',
                      attr_oper => exf$indexoper('all'),
                      attr_indexed => 'TRUE'),
       exf$attribute (attr_name => 'HorsePower(Model, Year)',
                      attr_oper => exf$indexoper('=','<','>','>=','<='),
                      attr_indexed => 'FALSE')
    )
```

EXF$INDEXOPER

The EXF$INDEXOPER type is used to specify the list of common operators in predicates with a stored or an indexed attribute.

The EXF$INDEXOPER type is defined as follows:

```
CREATE or REPLACE TYPE exfsys.exf$indexoper as VARRAY(20) of VARCHAR2(15);
```

The values for the EXF$INDEXOPER array are expected to be from the list in the following table:

Value	Predicate Description
=	Equality predicates
>	Greater than predicates
<	Less than predicates
>=	Greater than or equal to predicates
<=	Less than or equal to predicates
!= or <> or ^=	Not equal to predicates
IS NULL	IS NULL predicates
IS NOT NULL	IS NOT NULL predicates
ALL	All the operators listed in this table starting with the equality predicate through the IS NOT NULL predicate
NVL	Predicates with NVL (equality) operator
LIKE	Predicates with LIKE operator
BETWEEN	BETWEEN predicates

Attributes

None.

Usage Notes

A value of ALL for one of the EXF$INDEXOPER items implies that all the simple operators (=,>,<,>=,<=,!=, IS NULL, IS NOT NULL) are common in the predicates with an attribute. This value can be used along with one or more complex operators (NVL, LIKE and BETWEEN).

A predicate with a BETWEEN operator is treated as two predicates with binary operators, one with '>=' operator and another with '<=' operator. By default, only one of these operators is indexed, and the other operator is evaluated by value substitution. However, if predicates with the BETWEEN operator are common for an attribute (stored or indexed), both the binary operators resulting from the BETWEEN operator can be indexed by specifying BETWEEN in the EXF$INDEXOPER VARRAY. However, because this uses additional space in the predicate table, this operator should be used only when majority of predicates with an attribute use the BETWEEN operator.

When the LIKE operator is chosen as one of the common operators for an attribute, LIKE predicates on that attributes are indexed. Indexing a LIKE operator is beneficial only if the VARCHAR2 constant on the right-hand side of the predicate does not lead with a wild-card character. For example, indexing a LIKE operator will filter the following predicates efficiently:

```
company LIKE 'General%'
company LIKE 'Proctor%'
```

But, the following predicates are evaluated as sparse predicates in the last stage:

```
company LIKE '%Electric'
company LIKE "%Gamble'
```

Examples

An attribute with a list of common operators is represented as follows:

```
exf$attribute (attr_name => 'HorsePower(Model, Year)',
               attr_oper => exf$indexoper('=','<','>','>=','<=', 'between'),
               attr_indexed => 'TRUE')
```

EXF$TABLE_ALIAS

A table alias is a special form of elementary attribute that can be included in the attribute set. These attributes are used to manage expressions defined on one or more database tables.

Attributes

Name	Datatype	Description
`table_name`	`VARCHAR2(70)`	Name of the table with a possible schema extension.

Usage Notes

The concept of a table alias attribute is captured in the Expression Filter dictionary and the corresponding attribute in the attribute set's object type is created with a `VARCHAR2` datatype. (Also see Appendix A and "ADD_ELEMENTARY_ ATTRIBUTE Procedure" in Chapter 8.)

Examples

For a set of expressions defined on database tables, the corresponding table alias attributes are configured as follows:

```
BEGIN
  DBMS_EXPFIL.ADD_ELEMENTARY_ATTRIBUTE (
                             attr_set  => 'HRAttrSet',
                             attr_name => 'EMP',
                             tab_alias => exf$table_alias('SCOTT.EMP'));
  DBMS_EXPFIL.ADD_ELEMENTARY_ATTRIBUTE (
                             attr_set  => 'HRAttrSet',
                             attr_name => 'DEPT',
                             tab_alias => exf$table_alias('DEPT'));
END;
/
```

The Expression column using the previous attribute set can store expressions of form `EMP.JOB = 'Clerk' and EMP.NAME = 'Joe'`, where `JOB` and `NAME` are the names of the columns in the `SCOTT.EMP` table.

EXF$XPATH_TAG

The EXF$XPATH_TAG type is used to configure an XML element or an XML attribute for indexing a set of XPath predicates.

Attributes

Name	Datatype	Description
tag_name	VARCHAR2(70)	Name of the XML element or attribute. The name for an XML attribute is formatted as: <ElementName>@<AttributeName>.
tag_indexed	VARCHAR2(5)	TRUE if XML tag is indexed; otherwise FALSE. Default: TRUE if the tag is a positional filter. FALSE if the tag is a value filter.
tag_type	VARCHAR2(30)	Datatype for the value in the case of value filter. NULL for positional filters.

Usage Notes

EXF$XPATH_TAG type is used to configure an XML element or an attribute as a positional or a value filter for an Expression Filter index (see Chapter 3). An instance of the EXF$XPATH_TAG type with NULL value for tag_type configures the XML tag as a positional filter. In the current release, the only other possible values for the tag_type attribute are strings (CHAR or VARCHAR) and such tags are configured as value filters. By default, all positional filters are indexed and the value filters are not indexed. This behavior can be overridden by setting a TRUE or FALSE value for the tag_indexed attribute accordingly.

Examples

An XML element can be configured as a positional filter and be indexed using the following instance of the EXF$XPATH_TAG type.

```
exf$xpath_tag(tag_name    => 'stereo',        --- XML element
              tag_indexed => 'TRUE',          --- indexed predicate group
              tag_type    => null)            --- positional filter
```

An XML attribute can be configured as a value filter and be indexed using the following type instance.

```
exf$xpath_tag(tag_name    => 'stereo@make',  --- XML attribute
              tag_indexed => 'TRUE',         --- indexed predicate group
              tag_type    => 'VARCHAR(15)')  --- value filter
```

EXF$XPATH_TAGS

A type used to specify a list of XML tags while configuring the Expression Filter index parameters. This type is defined as follows:

```
CREATE or REPLACE TYPE exf$xpath_tags as VARRAY(490) of exf$xpath_tag;
```

Attributes

None.

Usage Notes

`EXF$XPATH_TAGS` type is used to specify a list of XML tags while configuring the Expression Filter index parameters. (See "DEFAULT_XPINDEX_PARAMETERS Procedure" in Chapter 8.)

Examples

A list of XML tags configured as positional and value filters can be represented as follows:

```
exf$xpath_tags(
           exf$xpath_tag(tag_name    => 'stereo@make',  --- XML attribute
                         tag_indexed => 'TRUE',
                         tag_type    => 'VARCHAR(15)'), --- value filter
           exf$xpath_tag(tag_name    => 'stereo',       --- XML element
                         tag_indexed => 'FALSE',
                         tag_type    => null),          --- positional filter
           exf$xpath_tag(tag_name    => 'memory',       --- XML element
                         tag_indexed => 'TRUE',
                         tag_type    => 'VARCHAR(10)')  --- value filter
          )
```

8 Management Procedures Using the DBMS_ EXPFIL Package

The Expression Filter DBMS_EXPFIL package contains all the procedures used to manage attribute sets, expression sets, expression indexes, optimizer statistics, and privileges. Table 8–1 describes the procedures in the DBMS_EXPFIL package. These procedures are further described in this chapter.

All the values and names passed to the procedures defined in the DBMS_EXPFIL package are not case sensitive, unless otherwise mentioned. To preserve the case, you use double quotation marks around the values.

Table 8–1 DBMS_EXPFIL Procedures

Procedure	Description
ADD_ELEMENTARY_ ATTRIBUTE	Adds the specified attribute to the attribute set.
ADD_FUNCTIONS	Adds a function, type, or package to the approved list of functions with an attribute set.
ASSIGN_ATTRIBUTE_ SET	Assigns an attribute set to a column storing expressions.
BUILD_EXCEPTIONS_ TABLE	Creates an exception table to hold references to invalid expressions.
CLEAR_EXPRSET_STATS	Clears the predicate statistics for an expression set.
COPY_ATTRIBUTE_SET	Makes a copy of the attribute set.
CREATE_ATTRIBUTE_ SET	Creates an attribute set.
DEFAULT_INDEX_ PARAMETERS	Assigns default index parameters to an attribute set.

Table 8–1 DBMS_EXPFIL Procedures

Procedure	Description
DEFAULT_XPINDEX_ PARAMETERS	Assigns default XPath index parameters to an attribute set.
DEFRAG_INDEX	Rebuilds the bitmap indexes online to reduce fragmentation.
DROP_ATTRIBUTE_SET	Drops an unused attribute set.
GET_EXPRSET_STATS	Collects predicate statistics for an expression set.
GRANT_PRIVILEGE	Grants an expression DML privilege to a user.
INDEX_PARAMETERS	Assigns index parameters to an expression set.
REVOKE_PRIVILEGE	Revokes an expression DML privilege from a user.
UNASSIGN_ATTRIBUTE_ SET	Breaks the association between a column storing expressions and the attribute set.
VALIDATE_ EXPRESSIONS	Validates expression metadata and the expressions stored in a column.
XPINDEX_PARAMETERS	Assigns XPath index parameters to an expression set.

ADD_ELEMENTARY_ATTRIBUTE Procedure

This procedure adds the specified attribute to the attribute set.

Format

```
procedure ADD_ELEMENTARY_ATTRIBUTE (
              attr_set   IN   VARCHAR2,         --- attr set name
              attr_name  IN   VARCHAR2,         --- attr name
              attr_type  IN   VARCHAR2);        --- attr type

--- or

procedure ADD_ELEMENTARY_ATTRIBUTE (
              attr_set   IN   VARCHAR2,         --- attr set name
              attr_name  IN   VARCHAR2,         --- table alias (name)
              tab_alias  IN   exf$table_alias); --- table alias for
```

Arguments

attr_set
Name of the attribute set to which this attribute is added.

attr_name
Name of the elementary attribute to be added. No two attributes in a set can have the same name.

attr_type
Datatype of the attribute. This argument accepts any standard SQL datatype or the name of an object type that is accessible to the current user.

tab_alias
The type that identifies the database table to which the attribute is aliased.

Usage Notes

This procedure adds an elementary attribute to an attribute set. If the attribute set was originally created from an existing object type, then additional attributes cannot be added.

One or more, or all elementary attributes in an attribute set can be table aliases. If an elementary attribute is a table alias, then the value assigned to the elementary attribute is a ROWID from the corresponding table. An attribute set with one or more table alias attributes cannot be created from an existing object type. For more information about table aliases, see Appendix A.

Elementary attributes cannot be added to an attribute set that is already assigned to a column storing expressions.

See "Defining Attribute Sets" on page 1-5 for more information about adding elementary attributes.

Related views: USER_EXPFIL_ATTRIBUTE_SETS and USER_EXPFIL_ATTRIBUTES.

Examples

The following commands add two elementary attributes to an attribute set:

```
BEGIN
  DBMS_EXPFIL.ADD_ELEMENTARY_ATTRIBUTE (
                             attr_set  => 'HRAttrSet',
                             attr_name => 'HRREP',
                             attr_type => 'VARCHAR2(30)');
  DBMS_EXPFIL.ADD_ELEMENTARY_ATTRIBUTE (
                             attr_set  => 'HRAttrSet',
                             attr_name => 'DEPT',
                             tab_alias => exf$table_alias('DEPT'));
END;
/
```

ADD_FUNCTIONS Procedure

This procedure adds a user-defined function, package, or type representing a set of functions to the attribute set.

Format

```
PROCEDURE ADD_FUNCTIONS (
              attr_set   IN   VARCHAR2,    --- attr set name
              funcs_name IN   VARCHAR2);   --- function/package/type name
```

Arguments

attr_set
Name of the attribute set to which the functions are added.

funcs_name
Name of a function, package, or type (representing a function set) or its synonyms.

Usage Notes

By default, an attribute set implicitly allows references to all Oracle supplied SQL functions for use by the expression set. If the expression set refers to a user-defined function, the expression set must be explicitly added to the attribute set.

The ADD_FUNCTIONS procedure adds a user-defined function or a package (or type) representing a set of functions to the attribute set. Any new or modified expressions are validated using this list.

The function or the package name can be specified with a schema extension. If a function name is specified without a schema extension, only such references in the expression set are considered valid. The expressions in a set can be restricted to use a synonym to a function or a package by adding the corresponding synonym to the attribute set. This preserves the portability of the expression set to other schemas.

See "Defining Attribute Sets" on page 1-5 for more information about adding functions to an attribute set.

Related views: USER_EXPFIL_ATTRIBUTE_SETS and USER_EXPFIL_ASET_
FUNCTIONS

Examples

The following commands add two functions to the attribute set:

```
BEGIN
  DBMS_EXPFIL.ADD_FUNCTIONS (attr_set   => 'Car4Sale',
                             funcs_name => 'HorsePower');
  DBMS_EXPFIL.ADD_FUNCTIONS (attr_set   => 'Car4Sale',
                             funcs_name => 'Scott.CrashTestRating');
END;
/
```

ASSIGN_ATTRIBUTE_SET Procedure

This procedure assigns an attribute set to a `VARCHAR2` column in a user table to create an Expression column.

Format

```
PROCEDURE ASSIGN_ATTRIBUTE_SET (
              attr_set   IN   VARCHAR2,    --- attr set name
              expr_tab   IN   VARCHAR2,    --- name of the table
              expr_col   IN   VARCHAR2,    --- exp column in the table
              force      IN   VARCHAR2     --- to use existing expressions
                         default 'FALSE');
```

Arguments

attr_set
The name of the attribute set.

expr_tab
The table storing the expression set.

expr_col
The column in the table that stores the expressions.

force
Argument used to trust the existing expressions in a table (and skip validation).

Usage Notes

The ASSIGN_ATTRIBUTE_SET procedure assigns an attribute set to a `VARCHAR2` column in a user table to create an Expression column. The attribute set contains the elementary attribute names and their datatypes and any functions used in the expressions. The attribute set is used by the Expression column to validate changes and additions to the expression set.

An attribute set can be assigned only to a table column in the same schema as the attribute set. An attribute set can be assigned to one or more table columns. Assigning an attribute set to a column storing expressions implicitly creates methods for the associated object type. For this operation to succeed, the object type cannot have any dependent objects before the attribute set is assigned.

By default, the column should not have any expressions at the time of association. However, if the values in the column are known to be valid expressions, you can use a value of `'TRUE'` for the `force` argument to assign the attribute set to a column containing expressions.

See "Defining Expression Columns" on page 1-7 for more information about adding elementary attributes.

Related views: USER_EXPFIL_ATTRIBUTE_SETS and USER_EXPFIL_ EXPRESSION_SETS

Examples

The following command assigns the attribute set to a column storing expressions. The expression set should be empty at the time of association.

```
BEGIN
  DBMS_EXPFIL.ASSIGN_ATTRIBUTE_SET (attr_set => 'Car4Sale',
                                    expr_tab => 'Consumer',
                                    expr_col => 'Interest');
END;
/
```

BUILD_EXCEPTIONS_TABLE Procedure

This procedure creates the exception table, used in validation, in the current schema.

Format

```
PROCEDURE BUILD_EXCEPTIONS_TABLE (
              exception_tab IN VARCHAR2);  -- exception table to be created --
```

Arguments

exception_tab
Name of the exception table.

Usage Notes

The expressions stored in a table column can be validated using the VALIDATE_EXPRESSIONS procedure. During expression validation, you can optionally provide the name of the exception table in which the references to the invalid expressions are stored. The BUILD_EXCEPTIONS_TABLE procedure creates the exception table in the current schema.

See "Evaluation Semantics" on page 1-13 and "VALIDATE_EXPRESSIONS Procedure" on page 8-30 for more information.

Related view: USER_TABLES

Examples

The following command creates the exception table, `InterestExceptions`, in the current schema:

```
BEGIN
  DBMS_EXPFIL.BUILD_EXCEPTIONS_TABLE (
                              exception_tab => 'InterestExceptions');
END;
/
```

CLEAR_EXPRSET_STATS Procedure

This procedure clears the predicate statistics for the expression set stored in a table column.

Format

```
PROCEDURE CLEAR_EXPRSET_STATS (
              expr_tab   IN   VARCHAR2,    --- table storing expression set
              expr_col   IN   VARCHAR2);   --- column in the table with set
```

Arguments

exp_tab
The table storing the expression set.

expr_col
The column in the table that stores the expressions.

Usage Notes

This procedure clears the predicate statistics for the expression set stored in a table column. See also "GET_EXPRSET_STATS Procedure" on page 8-21 for information about gathering the statistics.

Related views: USER_EXPFIL_EXPRESSION_SETS and USER_EXPFIL_EXPRSET_STATS

Examples

The following command clears the predicate statistics for the expression set stored in `Interest` column of the `Consumer` table:

```
BEGIN
  DBMS_EXPFIL.CLEAR_EXPRSET_STATS (expr_tab => 'Consumer',
                                   expr_col => 'Interest');
END;
/
```

COPY_ATTRIBUTE_SET Procedure

This procedure copies an attribute set along with its user-defined function list and default index parameters to another set.

Format

```
PROCEDURE COPY_ATTRIBUTE_SET (
             from_set   IN   VARCHAR2,    --- name of an existing att set
             to_set     IN   VARCHAR2);   --- new set name
```

Arguments

from_set
Name of an existing attribute set to be copied.

to_set
Name of the new attribute set.

Usage Notes

A schema-extended name can be used for the `from_set` argument to copy an attribute set across schemas. The user issuing the command must have `EXECUTE` privileges for the object type associated with the original attribute set. The user must ensure that any references to schema objects (user-defined functions, tables, and embedded objects) are valid in the new schema.

The default index parameters and the user-defined function list of the new set can be changed independent of the original set.

Related views: ALL_EXPFIL_ATTRIBUTE_SETS and ALL_EXPFIL_ATTRIBUTES.

Examples

The following command makes a copy of the Car4Sale attribute set:

```
BEGIN
  DBMS_EXPFIL.COPY_ATTRIBUTE_SET (from_set => 'Car4Sale',
                                  to_set   => 'Vehicle');
END;
/
```

CREATE_ATTRIBUTE_SET Procedure

This procedure creates an empty attribute set or an attribute set with a complete set of elementary attributes derived from an object type with a matching name.

Format

```
PROCEDURE CREATE_ATTRIBUTE_SEt (
              attr_set   IN   VARCHAR2,    --- attr set name
              from_type  IN   VARCHAR2     --- object type for attributes
                         default 'NO');
```

Arguments

attr_set
The name of the attribute set to be created.

from_type
YES, if the attributes for the attribute set should be derived from an existing object type.

Usage Notes

The object type used for an attribute set cannot contain any user methods, and it should not be an evolved type (with the use of ALTER TYPE command). This object type should not have any dependent objects at the time of the attribute set creation. If the attribute set is not derived from an existing object type, this procedure creates an object type with a matching name.

An attribute set with one or more table alias attributes cannot be derived from an object type. For this purpose, create an empty attribute set and add one elementary attribute at a time using the DBMS_EXPFIL.ADD_ELEMENTARY_ATTRIBUTE procedure. (See Appendix A for more information.)

See "Defining Attribute Sets" on page 1-5 and "ADD_ELEMENTARY_ATTRIBUTE Procedure" on page 8-3 for more information.

Related views: USER_EXPFIL_ATTRIBUTE_SET and USER_EXPFIL_ATTRIBUTES.

Examples

The following commands create an attribute set with all the required elementary attributes derived from the `Car4Sale` type:

```
CREATE OR REPLACE TYPE Car4Sale AS OBJECT
                                   (Model   VARCHAR2(20),
                                    Year    NUMBER,
                                    Price   NUMBER,
                                    Mileage NUMBER);
/


BEGIN
  DBMS_EXPFIL.CREATE_ATTRIBUTE_SET(attr_set  => 'Car4Sale',
                                   from_type => 'YES');
END;
/
```

Assuming that the `Car4Sale` type does not exist, the attribute set can be created from scratch as shown in the following example:

```
BEGIN
  DBMS_EXPFIL.CREATE_ATTRIBUTE_SET(attr_set => 'Car4Sale');
  DBMS_EXPFIL.ADD_ELEMENTARY_ATTRIBUTE(
                               attr_set  => 'Car4Sale',
                               attr_name => 'Model',
                               attr_type => 'VARCHAR2(20)');
  DBMS_EXPFIL.ADD_ELEMENTARY_ATTRIBUTE(
                               attr_set  => 'Car4Sale',
                               attr_name => 'Year',
                               attr_type => 'NUMBER');
  DBMS_EXPFIL.ADD_ELEMENTARY_ATTRIBUTE(
                               attr_set  => 'Car4Sale',
                               attr_name => 'Price',
                               attr_type => 'NUMBER');
  DBMS_EXPFIL.ADD_ELEMENTARY_ATTRIBUTE(
                               attr_set  => 'Car4Sale',
                               attr_name => 'Mileage',
                               attr_type => 'NUMBER');
END;
/
```

DEFAULT_INDEX_PARAMETERS Procedure

This procedure assigns default index parameters to an attribute set. It also adds or drops a partial list of stored and indexed attributes to or from the default list associated with the attribute list.

Format

```
PROCEDURE DEFAULT_INDEX_PARAMETERS (
              attr_set   IN   VARCHAR2,    --- attribute set name
              attr_list  IN   EXF$ATTRIBUTE_LIST,
                                           --- stored and indexed attributes
              operation  IN   VARCHAR2     --- to ADD or DROP
                               default 'ADD');
```

Arguments

attr_set
The name of the attribute set.

attr_list
An instance of `EXF$ATTRIBUTE_LIST` with a partial list of (default) stored and indexed attributes for an Expression Filter index.

operation
The operation to be performed on the list of index parameters. Default value: `ADD`. Valid values: `ADD` and `DROP`.

Usage Notes

Existing Expression Filter indexes are not modified when the default parameters for the corresponding attribute set are changed. The new index defaults are used when a new Expression Filter index is created and when an existing index is rebuilt. (See ALTER INDEX REBUILD in Chapter 6 for more information about rebuilding indexes.)

See "Creating an Index from Default Parameters" on page 2-7 for more information about assigning default index parameters to an attribute set.

Related views: USER_EXPFIL_ATTRIBUTE_SETS and USER_EXPFIL_DEF_INDEX_PARAMS

Examples

The following command adds the specified stored and indexed attributes to the attribute set's default index parameters list:

```
BEGIN
 DBMS_EXPFIL.DEFAULT_INDEX_PARAMETERs(
    attr_set  => 'Car4Sale',
    attr_list => exf$attribute_list (
       exf$attribute (attr_name => 'Model',
                      attr_oper => exf$indexoper('='),
                      attr_indexed => 'TRUE'),
       exf$attribute (attr_name => 'Price',
                      attr_oper => exf$indexoper('all'),
                      attr_indexed => 'TRUE'),
       exf$attribute (attr_name => 'HorsePower(Model, Year)',
                      attr_oper => exf$indexoper('=','<','>','>=','<='),
                      attr_indexed => 'FALSE'),
       exf$attribute (attr_name => 'CrashTestRating(Model, Year)',
                      attr_oper => exf$indexoper('=','<','>','>=','<='),
                      attr_indexed => 'FALSE')),
    operation => 'ADD');
END;
/
```

The following command drops the `CrashTestRating(Model, Year)` attribute (stored or indexed) from the previous list.

```
BEGIN
  DBMS_EXPFIL.DEFAULT_INDEX_PARAMETERS(
      attr_set  => 'Car4Sale',
      attr_list => exf$attribute_list (
         exf$attribute (attr_name => 'CrashTestRating(Model, Year)')),
      operation => 'DROP');
END;
/
```

DEFAULT_XPINDEX_PARAMETERS Procedure

This procedure adds (or drops) a partial list of XPath parameters to the default index parameters associated with the attribute set.

Format

```
PROCEDURE DEFAULT_XPINDEX_PARAMETERS (
              attr_set   IN   VARCHAR2,    --- attribute set name
              xmlt_attr  IN   VARCHAR2,    --- XMLType attrubue name
              xptag_list IN   EXF$XPATH_TAGS,
                                           --- XPath tags for index
              operation  IN   VARCHAR2     --- to ADD or DROP
                               default 'ADD');
```

Arguments

attr_set
The name of the attribute set.

xmlt_attr
The name of the attribute with the XMLType datatype.

xptag_list
An instance of EXF$XPATH_TAGS type with a partial list of XML elements and attributes to be configured for the Expression Filter index.

operation
The operation to be performed on the list of index parameters. Default value: ADD. Valid values: ADD and DROP.

Usage Notes

The attribute set used for an expression set may have one or more XML type attributes (defined with XMLType datatype) and the corresponding expressions may contain XPath predicates on these attributes. The Expression Filter index created for the expression set can be tuned to process these XPath predicates efficiently by using some XPath-specific index parameters (in addition to some non-XPath index parameters).

The DEFAULT_XPINDEX_PARAMETERS procedure adds (or drops) a partial list of XPath parameters to the default index parameters associated with the attribute set.

The XPath parameters are assigned to a specific XMLType attribute in the attribute set and this information can be viewed using the USER_EXPFIL_DEF_INDEX_PARAMS view. The DEFAULT_INDEX_PARAMETERS procedure and the DEFAULT_XPINDEX_PARAMETERS procedure can be used independent of each other. They maintain a common list of default index parameters for the attribute set.

See "Index Tuning for XPath Predicates" on page 3-6 for more information about XPath parameters to the default index parameters of an attribute set. See also "DEFAULT_INDEX_PARAMETERS Procedure" on page 8-14 for more information about default index parameters.

Related views: USER_EXPFIL_ATTRIBUTES and USER_EXPFIL_DEF_INDEX_PARAMS.

Note: The values assigned to the `tag_name` argument of `exf$xpath_tag` type are case sensitive.

Examples

The following command adds the specified XML tags to the default index parameters list along with their preferences such as positional or value filter and indexed or stored predicate group:

```
BEGIN
  DBMS_EXPFIL.DEFAULT_XPINDEX_PARAMETERS(
       attr_set   => 'Car4Sale',
       xmlt_attr  => 'Details',
       xptag_list =>                                    --- XPath tag list
         exf$xpath_tags(
           exf$xpath_tag(tag_name    => 'stereo@make',  --- XML attribute
                         tag_indexed => 'TRUE',
                         tag_type    => 'VARCHAR(15)'), --- value filter
           exf$xpath_tag(tag_name    => 'stereo',       --- XML element
                         tag_indexed => 'FALSE',
                         tag_type    => null),          --- positional filter
           exf$xpath_tag(tag_name    => 'memory',       --- XML element
                         tag_indexed => 'TRUE',
                         tag_type    => 'VARCHAR(10)'), --- value filter
           exf$xpath_tag(tag_name    => 'GPS',
                         tag_indexed => 'TRUE',
                         tag_type    => null)
          )
        );
END;
```

```
/
```

The following command drops the `stereo@make` tag from the default index parameters:

```
BEGIN
  DBMS_EXPFIL.DEFAULT_XPINDEX_PARAMETERS(
       attr_set   => 'Car4Sale',
       xmlt_attr  => 'Details',
       xptag_list =>                                    --- XPath tag list
         exf$xpath_tags(
           exf$xpath_tag(tag_name    => 'stereo@make')
         ),
       operation => 'DROP'
       );
END;
/
```

DEFRAG_INDEX Procedure

This procedure rebuilds the bitmap indexes online and thus reduces the fragmentation.

Format

```
PROCEDURE DEFRAG_INDEX (
              idx_name   IN  VARCHAR2);    --- expfil index to defrag
```

Arguments

idx_name
The name of the Expression Filter index.

Usage Notes

The bitmap indexes defined for the indexed attributes of an Expression Filter index become fragmented as additions and updates are made to the expression set. The DEFRAG_INDEX procedure rebuilds the bitmap indexes online and thus reduces the fragmentation.

Indexes can be defragmented when the expression set is being modified. However, you should schedule defragmentation when the workload is relatively light.

See "Index Storage and Maintenance" on page 2-10 for more information about rebuilding indexes.

Related views: USER_EXPFIL_INDEXES and USER_INDEXES.

Examples

The following command is issued to defragment the bitmap indexes associated with the Expression Filter index:

```
BEGIN
  DBMS_EXPFIL.DEFRAG_INDEX (idx_name => 'InterestIndex');
END;
/
```

DROP_ATTRIBUTE_SET Procedure

This procedure drops an attribute set not being used for any expression set.

Format

```
PROCEDURE DROP_ATTRIBUTE_SET (
              attr_set   IN   VARCHAR2);   --- attr set name
```

Arguments

attr_set
The name of the attribute set to be dropped.

Usage Notes

The DROP_ATTRIBUTE_SET procedure drops an attribute set not being used for any expression set. If the attribute set was initially created from an existing object type, the object type remains after dropping the attribute set. Otherwise, the object type is dropped with the attribute set.

Related views: USER_EXPFIL_ATTRIBUTE_SETS and USER_EXPFIL_
EXPRESSION_SETS.

Examples

Assuming that the attribute set is not used by an Expression column, the following command drops the attribute set:

```
BEGIN
  DBMS_EXPFIL.DROP_ATTRIBUTE_SET(attr_set => 'Car4Sale');
END;
/
```

GET_EXPRSET_STATS Procedure

This procedure computes the predicate statistics for an expression set and stores them in the expression filter dictionary.

Format

```
PROCEDURE GET_EXPRSET_STATS (
              expr_tab   IN   VARCHAR2,    --- table storing expression set
              expr_col   IN   VARCHAR2);   --- column in the table with set
```

Arguments

expr_tab
The table storing the expression set.

expr_col
The column in the table that stores the expressions.

Usage Notes

When a representative set of expressions are stored in a table column, you can use predicate statistics for those expressions to configure the corresponding Expression Filter index (using the `TOP` parameters clause). The GET_EXPRSET_STATS procedure computes the predicate statistics for an expression set and stores them in the expression filter dictionary.

See "Creating an Index from Statistics" on page 2-9 for more information about using predicate statistics.

Related views: USER_EXPFIL_EXPRESSION_SETS and USER_EXPFIL_EXPRSET_STATS.

Examples

The following command computes the predicate statistics for the expressions stored in the `Interest` column of the `Consumer` table:

```
BEGIN
  DBMS_EXPFIL.GET_EXPRSET_STATS (expr_tab => 'Consumer',
                                 expr_col => 'Interest');
END;
/
```

GRANT_PRIVILEGE Procedure

This procedure grants privileges on one or more Expression columns to other users.

Format

```
PROCEDURE GRANT_PRIVILEGE (
              expr_tab   IN  VARCHAR2,     --- table w/ the expr column
              expr_col   IN  VARCHAR2,     --- column storing the expressions
              priv_type  IN  VARCHAR2,     --- type of priv to be granted
              to_user    IN  VARCHAR2);    --- user to which the priv is
                                           ---   granted
```

Arguments

expr_tab
The table storing the expression set.

expr_col
The column in the table that stores the expressions.

priv_type
Types of the privilege to be granted. Valid values: INSERT EXPRESSION, UPDATE EXPRESSION, ALL

to_user
User to which the privilege is granted.

Usage Notes

The SQL `EVALUATE` operator evaluates expressions with the privileges of the owner of the table that stores the expressions. The privileges of the user issuing the query are not considered. The owner of the table can insert, update, and delete expressions. Other users must have `INSERT` and `UPDATE` privileges for the table and `INSERT EXPRESSION` and `UPDATE EXPRESSION` privilege for a specific Expression column in the table.

Using the GRANT_PRIVILEGE procedure, the owner of the table can grant `INSERT EXPRESSION` or `UPDATE EXPRESSION` privileges on one or more Expression columns to other users. Both the privileges can be granted to a user by specifying `ALL` for the privilege type.

See "REVOKE_PRIVILEGE Procedure" on page 8-27 and "Granting and Revoking Privileges" on page 1-13 for more information about granting and revoking privileges.

Related views: USER_EXPFIL_EXPRESSION_SETS and USER_EXPFIL_ PRIVILEGES.

Examples

The owner of `Consumer` table can grant `INSERT EXPRESSION` privileges to user `SCOTT` with the following command. User `SCOTT` should also have `INSERT` privileges on the table so that he can add new expressions to the set.

```
BEGIN
  DBMS_EXPFIL.GRANT_PRIVILEGE (expr_tab  => 'Consumer',
                               expr_col  => 'Interest',
                               priv_type => 'INSERT EXPRESSION',
                               to_user   => 'SCOTT');
END;
/
```

INDEX_PARAMETERS Procedure

This procedure fine-tunes the index parameters for each expression set before index creation.

Format

```
PROCEDURE INDEX_PARAMETERS (
              expr_tab   IN   VARCHAR2,    --- table with expr column
              expr_col   IN   VARCHAR2,    --- column storing expressions
              attr_list  IN   EXF$ATTRIBUTE_LIST,
              operation  IN   VARCHAR2     --- type of operation
                               default 'ADD');
```

Arguments

expr_tab
The table storing the expression set.

expr_col
The column in the table that stores the expressions.

attr_list
An instance of `EXF$ATTRIBUTE_LIST` with a partial list of stored and indexed attributes.

operation
The operation to be performed on the list of index parameters. Default value: `ADD`. Valid values: `ADD`, `DROP`, `DEFAULT`, and `CLEAR`.

Usage Notes

An attribute set can be used by multiple expression sets stored in different columns of user tables. By default, the index parameters associated with the attribute set are used to define an Expression Filter index on an expression set. If you need to fine-tune the index for each expression set, you can specify a small list of the index parameters in the `PARAMETERS` clause of the `CREATE INDEX` statement. However, when an Expression Filter index uses a large number of index parameters or if the index is configured for XPath predicates, fine-tuning the parameters with the `CREATE INDEX` statement is not possible.

The INDEX_PARAMETERS procedure fine-tunes the index parameters for each expression set before index creation. This procedure can be used to copy the defaults from the corresponding attribute set and selectively add (or drop) additional index parameters for the expression set. (You use the XPINDEX_PARAMETERS procedure to add and drop XPath index parameters.) The Expression Filter index defined for an expression set with a non-empty list of index parameters always uses these parameters. The INDEX_PARAMETERS procedure cannot be used when the Expression Filter index is already defined for the column storing expressions.

The operations allowed with this procedure include:

- Deriving the current list of default index parameters (including any XPath-specific parameters) from the corresponding attribute set and assigning them to the specified expression set (a value of `DEFAULT` for the operation argument).
- Adding (or dropping) one or more attributes to (or from) the current list of parameters assigned to the expression set (values of `ADD` or `DROP` for the operation argument).
- Clearing the index parameters assigned to the expression set. This enables the user to start using default parameters or tune the parameters from scratch (a value of `CLEAR` for the operation argument).

Note: This procedure is useful only when an attribute set is shared across multiple expression sets. In all other cases, the defaults assigned to the attribute set can be tuned for the expression set using it.

See "Creating an Index from Exact Parameters" on page 2-8 and "XPINDEX_PARAMETERS Procedure" on page 8-32 for more information.

Related views: USER_EXPFIL_EXPRESSION_SETS, USER_EXPFIL_DEF_INDEX_PARAMETERS and USER_EXPFIL_INDEX_PARAMETERS.

Examples

The following command synchronizes the expression set's index parameters with the defaults associated with the corresponding attribute set:

```
BEGIN
  DBMS_EXPFIL.INDEX_PARAMETERS(expr_tab  => 'Consumer',
                               expr_col  => 'Interest',
                               attr_list => null,
                               operation => 'DEFAULT');
END;
/
```

The following command adds a stored attribute to the expression set's index parameters.

```
BEGIN
  DBMS_EXPFIL.INDEX_PARAMETERS(expr_tab  => 'Consumer',
                               expr_col  => 'Interest',
                               attr_list =>
                                exf$attribute_list (
                                 exf$attribute (
                                   attr_name => 'CrashTestRating(Model, Year)',
                                   attr_oper => exf$indexoper('all'),
                                   attr_indexed => 'FALSE')),
                               operation => 'ADD');
END;
/
```

The following command clears the index parameters associated with the expression set:

```
BEGIN
  DBMS_EXPFIL.INDEX_PARAMETERS(expr_tab  => 'Consumer',
                               expr_col  => 'Interest',
                               attr_list => null,
                               operation => 'CLEAR');
END;
/
```

A subsequent index creation will use the default index parameters assigned to the corresponding attribute set.

REVOKE_PRIVILEGE Procedure

This procedure revokes an expression privilege previously granted by the owner.

Format

```
PROCEDURE REVOKE_PRIVILEGE (
              expr_tab   IN  VARCHAR2,     --- table with the expr column
              expr_col   IN  VARCHAR2,     --- column storing the expression
              priv_type  IN  VARCHAR2,     --- type of privilege to be revoked
              from_user  IN  VARCHAR2);    --- user from which the priv is
                                           ---   revoked
```

Arguments

expr_tab
The table storing the expression set.

expr_col
The column in the table that stores the expressions.

priv_type
Type of privilege to be revoked.

from_user
User from which the privilege is revoked.

Usage Notes

The REVOKE_PRIVILEGE procedure revokes an expression privilege previously granted by the owner.

See "GRANT_PRIVILEGE Procedure" on page 8-22 and "Granting and Revoking Privileges" on page 1-13 for more information about granting and revoking privileges.

Related views: USER_EXPFIL_EXPRESSION_SETS and USER_EXPFIL_PRIVILEGES.

Examples

The following command revokes the `INSERT EXPRESSION` privilege on the `Interest` column of the `Consumer` table from user `SCOTT`:

```
BEGIN
  DBMS_EXPFIL.REVOKE_PRIVILEGE (expr_tab  => 'Consumer',
                                expr_col  => 'Interest',
                                priv_type => 'INSERT EXPRESSION',
                                 from_user => 'SCOTT');
END;
/
```

UNASSIGN_ATTRIBUTE_SET Procedure

This procedure unassigns an attribute set from a column storing expressions.

Format

```
PROCEDURE UNASSIGN_ATTRIBUTE_SET (
              expr_tab   IN   VARCHAR2,    --- table with expr. column
              expr_col   IN   VARCHAR2);   --- column storing expr. set
```

Arguments

expr_tab
The table storing the expression set.

expr_col
The column in the table that stores the expressions.

Usage Notes

A column of an expression datatype can be converted back to a VARCHAR2 type by unassigning the attribute set. You can unassign an attribute set from a column storing expressions if an Expression Filter index is not defined on the column.

See "ASSIGN_ATTRIBUTE_SET Procedure" on page 8-7 for information about assigning attribute sets.

Related views: USER_EXPFIL_EXPRESSION_SETS and USER_EXPFIL_INDEXES.

Examples

The following command unassigns the attribute set previously assigned to the Interest column of the Consumer table. (See Section 5.1.)

```
BEGIN
  DBMS_EXPFIL.UNASSIGN_ATTRIBUTE_SET (expr_tab => 'Consumer',
                                      expr_col => 'Interest');
END;
/
```

VALIDATE_EXPRESSIONS Procedure

This procedure validates all the expressions in a set.

Format

```
PROCEDURE VALIDATE_EXPRESSIONS (
              expr_tab      IN  VARCHAR2,  --- expressions table
              expr_col      IN  VARCHAR2,  --- column storing expressions
              exception_tab IN  VARCHAR2   --- exception table
                    default null);
```

Arguments

expr_tab
The table storing the expression set.

expr_col
The column in the table that stores the expressions.

exception_tab
Name of the exception table. This table is created using the BUILD_EXCEPTIONS_TABLE procedure.

Usage Notes

The expressions stored in a table may have references to schema objects like user-defined functions and tables. When these schema objects are dropped or modified, the expressions could become invalid and the subsequent evaluation (query with `EVALUATE` operator) could fail.

The VALIDATE_EXPRESSIONS procedure validates all the expressions in a set. By default, the expression validation utility fails on the first expression that is invalid. Optionally, the caller can pass an exception table to store references to all the invalid expressions. In addition to validating expressions in the set, this procedure validates the parameters (stored and indexed attributes) of the associated index and the approved list of user-defined functions. Any errors in the index parameters or the user-defined function list are immediately reported to the caller.

See "Evaluation Semantics" on page 1-13 and "BUILD_EXCEPTIONS_TABLE Procedure" on page 8-9 for more information.

Related views: USER_EXPFIL_EXPRESSION_SETS, USER_EXPFIL_ASET_FUNCTIONS, and USER_EXPFIL_PREDTAB_ATTRIBUTES.

Examples

The following command validates the expressions stored in the `Interest` column of the `Consumer` table.

```
BEGIN
  DBMS_EXPFIL.VALIDATE_EXPRESSIONS (expr_tab => 'Consumer',
                                    expr_col => 'Interest');
END;
/
```

XPINDEX_PARAMETERS Procedure

This procedure is used in conjunction with the INDEX_PARAMETERS procedure to fine-tune the XPath-specific index parameters for each expression set.

Format

```
PROCEDURE XPINDEX_PARAMETERS (
              expr_tab   IN   VARCHAR2,    --- table with expr column
              expr_col   IN   VARCHAR2,    --- column storing expressions
              xmlt_attr  IN   VARCHAR2,    --- XMLType attrubue name
              xptag_list IN   EXF$XPATH_TAGS,
              operation  IN   VARCHAR2     --- to ADD or DROP
                               default 'ADD');
```

Arguments

expr_tab
The table storing the expression set.

expr_col
The column in the table that stores the expressions.

xmlt_attr
The name of the attribute with the `XMLType` datatype.

xptag_list
An instance of `EXF$XPATH_TAGS` type with a partial list of XML elements and attributes.

operation
The operation to be performed on the list of index parameters. Default value: `ADD`. Valid values: `ADD` or `DROP`.

Usage Notes

When an attribute set is shared by multiple expression sets, the INDEX_ PARAMETERS procedure can be used to tune the simple (non-XPath) index parameters for each expression set. The XPINDEX_PARAMETERS procedure is used in conjunction with the INDEX_PARAMETERS procedure to fine-tune the XPath-specific index parameters for each expression set.

See also "INDEX_PARAMETERS Procedure" on page 8-24 and "Index Tuning for XPath Predicates" on page 3-6 for more information.

Related views: USER_EXPFIL_ATTRIBUTES, USER_EXPFIL_DEF_INDEX_PARAMS, and USER_EXPFIL_INDEX_PARAMS.

Note: The values assigned to the `tag_name` argument of `exf$xpath_tag` type are case-sensitive.

Examples

The following command synchronizes the expression set's index parameters (XPath and non-XPath) with the defaults associated with the corresponding attribute set:

```
BEGIN
  DBMS_EXPFIL.INDEX_PARAMETERS(expr_tab  => 'Consumer',
                               expr_col  => 'Interest',
                               attr_list => null,
                               operation => 'DEFAULT');
END;
/
```

The following command adds an XPath-specific index parameter to the expression set:

```
BEGIN
  DBMS_EXPFIL.XPINDEX_PARAMETERS(expr_tab   => 'Consumer',
                                 expr_col   => 'Interest',
                                 xmlt_attr  => 'Details',
                                 xptag_list =>
                                  exf$xpath_tags(
                                   exf$xpath_tag(tag_name    => 'GPS',
                                                 tag_indexed => 'TRUE',
                                                 tag_type    => null)),
                                 operation  => 'ADD');
END;
/
```

9 Expression Filter Views

The Expression Filter metadata can be viewed using the Expression Filter views defined with a xxx_EXPFIL prefix, where xxx can be USER or ALL. These views are read-only to the users and are created and maintained by the Expression Filter procedures.

Table 9–1 lists the names of the views and their descriptions.

Table 9–1 Expression Filter Views

View Name	Description
USER_EXPFIL_ASET_FUNCTIONS	List of functions and packages approved for the attribute set.
USER_EXPFIL_ATTRIBUTES	List of elementary attributes of the attribute set.
USER_EXPFIL_ATTRIBUTE_SETS	List of attribute set.
USER_EXPFIL_DEF_INDEX_PARAMS	List of default index parameters.
USER_EXPFIL_EXPRESSION_SETS	List of expression sets.
USER_EXPFIL_EXPRSET_STATS	List of predicate statistics for the expression sets.
USER_EXPFIL_INDEX_PARAMS	List of index parameters assigned to the expression set.
USER_EXPFIL_INDEXES	List of expression filter indexes.
USER_EXPFIL_PREDTAB_ATTRIBUTES	List of stored and indexed attributes for the indexes.
USER_EXPFIL_PRIVILEGES	List of all the expression privileges of the current user.

9.1 USER_EXPFIL_ASET_FUNCTIONS View

This view lists all the functions and packages that are allowed in the expressions using a particular attribute set. This view is defined with the columns described in the following table:

Column Name	Datatype	Description
`ATTRIBUTE_SET_NAME`	`VARCHAR2`	Name of the attribute set.
`UDF_NAME`	`VARCHAR2`	Name of the user-defined function or package (or type) as specified by the user (with or without schema extension).
`OBJECT_OWNER`	`VARCHAR2`	Owner of the function or package (or type).
`OBJECT_NAME`	`VARCHAR2`	Name of the function or package (or type).
`OBJECT_TYPE`	`VARCHAR2`	Type of the object at the time the object was added to the attribute set: ■ Function: If the object is a function ■ Package: If the object is a package ■ Type: If the object is a type ■ Embedded type: If the object is a type that is implicitly added to the function list as the type is used by one of the elementary attributes in the set. ■ Synonym: Synonym to a function or package or type.

9.2 USER_EXPFIL_ATTRIBUTES View

This view lists all the elementary attributes of the attribute sets defined in the user's schema. This view is defined with the columns described in the following table:

Column Name	Datatype	Description
`ATTRIBUTE_SET_NAME`	`VARCHAR2`	Name of the attribute set.
`ATTRIBUTE`	`VARCHAR2`	Name of the elementary attribute.
`DATA_TYPE`	`VARCHAR2`	Datatype of the attribute.
`ASSOCIATED_TABLE`	`VARCHAR2`	Name of the corresponding table for the table alias attribute. Null for all other types of attributes.

9.3 USER_EXPFIL_ATTRIBUTE_SETS View

This view lists the attribute sets defined in the user's schema. This view is defined with the column described in the following table:

Column Name	Datatype	Description
ATTRIBUTE_ SET_NAME	VARCHAR2	Name of the attribute set.

9.4 USER_EXPFIL_DEF_INDEX_PARAMS View

This view lists the default index parameters (stored and indexed attributes) associated with the attribute sets defined in the user's schema. This view is defined with the columns described in the following table:

Column Name	Datatype	Description
ATTRIBUTE_SET_ NAME	VARCHAR2	Name of the attribute set.
ATTRIBUTE	VARCHAR2	Name of the stored attribute.
DATA_TYPE	VARCHAR2	Datatype of the attribute.
ELEMENTARY	VARCHAR2	YES, if the attribute is also the elementary attribute of the attribute set; otherwise, NO.
INDEXED	VARCHAR2	YES, if the stored attribute is also the indexed attribute; otherwise, NO.
OPERATOR_LIST	VARCHAR2	String representation of the common operators configured for the attribute.
XMLTYPE_ATTR	VARCHAR2	Name of the corresponding XMLType elementary attribute when the stored or indexed attribute is an XML tag.

9.5 USER_EXPFIL_EXPRESSION_SETS View

This view lists the expression sets defined in the user's schema. This view is defined with the columns described in the following table:

Column Name	Datatype	Description
EXPR_TABLE	VARCHAR2	Name of the table storing expressions.

Column Name	Datatype	Description
EXPR_COLUMN	VARCHAR2	Name of the column (in the table) storing expressions.
ATTRIBUTE_SET	VARCHAR2	Name of the corresponding attribute set.
LAST_ANALYZED	DATE	Date on which the predicate statistics for this expression set were recently computed. Null if statistics were not collected.
NUM_ EXPRESSIONS	NUMBER	Number of expressions in the set when the set was last analyzed.
PREDS_PER_EXPR	NUMBER	Average number of predicates for each expression (when last analyzed).
NUM_SPARSE_ PREDS	NUMBER	Number of sparse predicates in the expression set (when last analyzed).

9.6 USER_EXPFIL_EXPRSET_STATS View

This view lists the predicate statistics for the expression sets in the user's schema. This view is defined with the columns described in the following table:

Column Name	Datatype	Description
EXPR_TABLE	VARCHAR2	Name of the table storing expressions.
EXPR_COLUMN	VARCHAR2	Name of the column (in the table) storing expressions.
ATTRIBUTE_EXP	VARCHAR2	The arithmetic expression that represents a common LHS in the predicates of the expression set.
PCT_OCCURRENCE	NUMBER	Percentage occurrence of the attribute in the expression set.
PCT_EQ_OPER	NUMBER	Percentage of predicates (of the attribute) with equality (=) operator.
PCT_LT_OPER	NUMBER	Percentage of predicates (of the attribute) with the less than (<) operator.
PCT_GT_OPER	NUMBER	Percentage of predicates (of the attribute) with the greater than (>) operator.
PCT_LTEQ_OPER	NUMBER	Percentage of predicates (of the attribute) with the less than or equal to (<=) operator.
PCT_GTEQ_OPER	NUMBER	Percentage of predicates (of the attribute) with the greater than or equal to (>=) operator.

Column Name	Datatype	Description
PCT_NEQ_OPER	NUMBER	Percentage of predicates (of the attribute) with the not equal to (!=) operator.
PCT_NUL_OPER	NUMBER	Percentage of predicates (of the attribute) with the IS NULL operator.
PCT_NNUL_OPER	NUMBER	Percentage of predicates (of the attribute) with the IS NOT NULL operator.
PCT_BETW_OPER	NUMBER	Percentage of predicates (of the attribute) with the BETWEEN operator.
PCT_NVL_OPER	NUMBER	Percentage of predicates (of the attribute) with the NVL operator.
PCT_LIKE_OPER	NUMBER	Percentage of predicates (of the attribute) with the LIKE operator.

9.7 USER_EXPFIL_INDEX_PARAMS View

This view lists the index parameters associated with the expression sets defined in the user's schema. This view is defined with the columns described in the following table:

Column Name	Datatype	Description
EXPSET_TABLE	VARCHAR2	Name of the table storing the expressions.
EXPSET_COLUMN	VARCHAR2	Name of the column storing the expressions.
ATTRIBUTE	VARCHAR2	Name of the stored attribute.
DATA_TYPE	VARCHAR2	Datatype of the attribute.
ELEMENTARY	VARCHAR2	YES if the attribute is also the elementary attribute of the attribute set; otherwise, NO.
INDEXED	VARCHAR2	YES if the stored attribute is also the indexed attribute; otherwise, NO.
OPERATOR_LIST	VARCHAR2	String representation of the common operators configured for the attribute.
XMLTYPE_ATTR	VARCHAR2	Name of the corresponding XMLType elementary attribute when the stored or indexed attribute is an XML tag.

9.8 USER_EXPFIL_INDEXES View

This view lists the Expression Filter indexes defined in the user's schema. This view is defined with the columns described in the following table:

Column Name	Datatype	Description
INDEX_NAME	VARCHAR2	Name of the index.
PREDICATE_TABLE	VARCHAR2	Name of the predicate table used for the index.
ACCESS_FUNC_PACKAGE	VARCHAR2	Name of the package that defines the functions with queries on the predicate table.
ATTRIBUTE_SET	VARCHAR2	Name of the corresponding attribute set.
EXPRESSION_TABLE	VARCHAR2	Name of the table on which the index is defined.
EXPRESSION_COLUMN	VARCHAR2	Name of the column on which the index is defined.
STATUS	VARCHAR2	Index status: ■ VALID: Index was created successfully. ■ FAILED: Index build failed, and it should be dropped and re-created. ■ FAILED RBLD: Index build or rebuild failed, and it can be rebuilt using the ALTER INDEX REBUILD statement.

9.9 USER_EXPFIL_PREDTAB_ATTRIBUTES View

This view shows the exact list of stored and indexed attributes used for expression filter indexes in the user's schema. This view is defined with the columns described in the following table:

Column Name	Datatype	Description
INDEX_NAME	VARCHAR2	Name of the index.
ATTRIBUTE_ID	NUMBER	Attribute identifier (unique for an index).
ATTRIBUTE_ALIAS	VARCHAR2	Alias given to the stored attribute.

Column Name	Datatype	Description
SUBEXPRESSION	VARCHAR2	The arithmetic expression that represents the stored attribute (also the LHS of predicates in the set).
DATA_TYPE	VARCHAR2	Derived datatype for the stored attribute.
INDEXED	VARCHAR2	YES, if the stored attribute is also the indexed attribute; otherwise, NO.
OPERATOR_LIST	VARCHAR2	String representation of the common operators configured for the attribute.
XMLTYPE_ATTR	VARCHAR2	Name of the corresponding XMLType elementary attribute when the stored or indexed attribute is an XML tag.
XPTAG_TYPE	VARCHAR2	Type of the XML tag: XML ELEMENT or XML ATTRIBUTE
XPFILTER_TYPE	VARCHAR2	Type of filter configured for the XML tag: POSITIONAL or [CHAR|INT|DATE] VALUE

9.10 USER_EXPFIL_PRIVILEGES View

This view lists the privileges of the current user on expression sets belonging to other schemas and the privileges of other users on the expression sets owned by the current user. This view is defined with the columns described in the following table:

Column Name	Datatype	Description
EXPSET_OWNER	VARCHAR2	Owner of the expression set.
EXPSET_TABLE	VARCHAR2	Name of the table storing expressions.
EXPSET_COLUMN	VARCHAR2	Name of the column storing the expressions.
GRANTEE	VARCHAR2	Grantee of the privilege.
INSERT_PRIV	VARCHAR2	Y if the grantee has the INSERT EXPRESSION privilege on the expression set; otherwise, N.
UPDATE_PRIV	VARCHAR2	Y if the grantee has the UPDATE EXPRESSION privilege on the expression set; otherwise, N.

A Managing Expressions Defined on One or More Database Tables

An Expression column can store expressions defined on one or more database tables. These expressions use special elementary attributes called table aliases. The elementary attributes are created using the EXF$TABLE_ALIAS type, and the name of the attribute is treated as the alias to the table specified through the EXF$TABLE_ALIAS type.

For example, there is a set of expressions defined on a transient variable HRMGR and two database tables, SCOTT.EMP and SCOTT.DEPT.

```
hrmgr='Greg' and emp.job='SALESMAN' and emp.deptno = dept.deptno and
    dept.loc = 'CHICAGO'
```

The attribute set for this type of expression is created as shown in the following example:

```
BEGIN
  -- Create the empty Attribute Set --
  DBMS_EXPFIL.CREATE_ATTRIBUTE_SET('hrdb');

  -- Add elementary attributes to the Attribute Set --
  DBMS_EXPFIL.ADD_ELEMENTARY_ATTRIBUTE('hrdb','hrmgr','VARCHAR2(20)');

  -- Define elementary attributes of EXF$TABLE_ALIAS type --
  DBMS_EXPFIL.ADD_ELEMENTARY_ATTRIBUTE('hrdb','emp',
                                      EXF$TABLE_ALIAS('scott.emp'));
  DBMS_EXPFIL.ADD_ELEMENTARY_ATTRIBUTE('hrdb','dept',
                                      EXF$TABLE_ALIAS('scott.dept'));
END;
/
```

The table HRInterest stores the expressions defined for this application. The Expression column in this table is configured as shown in the following example:

```
CREATE TABLE HRInterest (SubId number, Interest VARCHAR2(100));

BEGIN
  DBMS_EXPFIL.ASSIGN_ATTRIBUTE_SET('hrdb','HRInterest','Interest');
END;
/
-- insert the rows with expressions into the HRInterest table --
```

The expressions that use one or more table alias attributes can be indexed similar to those not using the table alias attributes. For example, the following CREATE INDEX statement configures stored and indexed attributes for the index defined on the Expression column:

```
CREATE INDEX HRIndex ON HRInterest (Interest) INDEXTYPE IS EXFSYS.EXPFILTER
  PARAMETERS ('STOREATTRS (emp.job, dept.loc, hrmgr)
               INDEXATTRS (emp.job, hrmgr)');
```

When the expression is evaluated, the values for the attributes defined as table aliases are passed by assigning the ROWIDs from the corresponding tables. The expressions stored in the HRInterest table can be evaluated for the data (rows) stored in EMP and DEPT tables (and a value of HRMGR) with the following query:

```
SELECT empno, job, sal, loc, SubId, Interest
   FROM emp, dept, HRInterest
   WHERE emp.deptno = dept.deptno AND
    EVALUATE(Interest, hrdb.getVarchar('Greg',emp.rowid,dept.rowid)) = 1;
```

Additional predicates can be added to the previous query if the expressions are evaluated only for a subset of rows in the EMP and DEPT tables:

```
SELECT empno, job, sal, loc, SubId, Interest
   FROM emp, dept, HRInterest
   WHERE emp.deptno = dept.deptno AND
         emp.sal > 1400 AND
       EVALUATE(Interest, hrdb.getVarchar('Greg',emp.rowid,dept.rowid)) = 1;
```

B Application Examples

This appendix describes examples of applications using the Expression Filter.

Active Application

In an active database system, the server performs some actions when certain criteria are met. For example, an application could monitor changes to data in a database table and react to these changes accordingly.

Consider the `Car4Sale` application described in Chapter 1. In this application, the `Consumer` table stores the information about consumers interested in buying used cars. In addition to the `Consumer` table described in Chapter 1, assume that there is an `Inventory` table that stores information about all the used cars available for sale, as defined in the following example:

```
CREATE TABLE Inventory (Model   VARCHAR2(20),
                        Year    NUMBER,
                        Price   NUMBER,
                        Mileage NUMBER);
```

Now, you can design the application such that the system reacts to any changes made to the data in the `Inventory` table, by defining a row trigger on the table:

```
CREATE TRIGGER activechk AFTER insert OR update ON Inventory
  FOR EACH ROW
  DECLARE
    cursor c1 (ditem VARCHAR2) is
      SELECT CId, Phone FROM Consumer WHERE EVALUATE (Interest, ditem) = 1;
    ditem VARCHAR2(200);
 BEGIN
  ditem := Car4Sale.getVarchar(:new.Model, :new.Year, :new.Price, :new.Mileage);

  for cur in c1(ditem) loop
    DBMS_OUTPUT.PUT_LINE('  For Model '||:new.Model||' Call '||cur.CId||
                         ' @ '||cur.Phone);
  end loop;
END;
/
```

This trigger evaluates the expressions for every row inserted (or updated) into the `Inventory` table and prints a message if a consumer is interested in the car. An Expression Filter index on the `Interest` column can speed up the query on the `Consumer` table.

Batch Evaluation of Expressions

To evaluate a set of expressions for a batch of data items, you can perform a simple join of the table storing data items and the table storing expressions. You can join the `Consumer` table with the `Inventory` table to determine the interest in each car, as shown in the following example:

```
SELECT DISTINCT Inventory.Model, count(*) as Demand
   FROM  Consumer, Inventory
   WHERE EVALUATE (Consumer.Interest,
                Car4Sale.getVarchar(Inventory.Model,
                                    Inventory.Year,
                                    Inventory.Price,
                                    Inventory.Mileage)) = 1
   GROUP BY Inventory.Model
   ORDER BY Demand DESC;
```

The `EVALUATE` operator's join semantics can also be used to maintain complex *N*-to-*M* (many-to-many) relationships between data stored in multiple tables.

Resource Management

Consider an application that manages IT support resources based on the responsibilities (or duties) and the workload of each representative. In this application, the responsibilities of the representatives are captured as expressions defined using variables such as the priority of the problem, organization, and the environment.

Create a table named `ITResource` to store information about all the available representatives, as shown in the following example:

```
-- Create the object type and the attribute set for ticket description --
CREATE OR REPLACE TYPE ITTicket AS OBJECT (
                       Priority       NUMBER,
                       Environment    VARCHAR2(10),
                       Organization   VARCHAR2(10));
```

```
/
BEGIN
  DBMS_EXPFIL.CREATE_ATTRIBUTE_SET(attr_set => 'ITTicket',
                                   from_type => 'Yes');
END;
/

-- Table storing expressions --
CREATE TABLE ITResource (RId        NUMBER,
                         Duties     VARCHAR2(100));

BEGIN
  DBMS_EXPFIL.ASSIGN_ATTRIBUTE_SET(attr_set => 'ITTicket',
                                   expr_tab => 'ITResource',
                                   expr_col => 'Duties');
END;
/

INSERT INTO ITResource (RId, Duties) VALUES
   (1, 'Priority <= 2 and Environment = ''NT'' and Organization =
                                                 ''Research''');

INSERT INTO ITResource (RId, Duties) VALUES
   (2, 'Priority = 1 and (Environment = ''UNIX'' or Environment = ''LINUX'')
        and Organization = ''APPS''');
```

Create a table named ITProblem to store the problems filed, as shown in the following example:

```
CREATE TABLE ITProblem (PId             NUMBER,
                        Description     ITTicket,
                        AssignedTo      NUMBER);
```

The AssignedTo column in the ITProblem table stores the identifier of the representative handling the problem.

Now, use the following UPDATE statement to assign all the previously unassigned problems to capable IT representatives:

```
UPDATE ITProblem p SET AssignedTo =
             (SELECT RId FROM ITResource r
              WHERE EVALUATE(r.Duties, p.Description.getVarchar()) = 1
                    and rownum < 2)
    WHERE AssignedTo IS NULL;
```

The previous UPDATE operation can benefit from an Expression Filter index defined on the `Duties` column of the `Resource` table.

C Installing Oracle Expression Filter

Expression Filter provides a SQL schema and PL/SQL and Java packages that are used to store, retrieve, update, and query collections of expressions in an Oracle database.

Expression Filter is installed automatically with Oracle Database 10*g* Standard Edition and Oracle Database 10*g* Enterprise Edition. It is supplied as a set of PL/SQL packages, a Java package, a set of dictionary tables, and catalog views. All these objects are created in a dedicated schema named `EXFSYS`.

The script to install the Expression Filter is named `catexf.sql` and is found in the $ORACLE_HOME/rdbms/admin/directory. This script should be executed from a SQL*Plus session while connected as `SYSDBA`. Expression Filter can be uninstalled using the `catnoexf.sql` script in the same directory.

The Expression Filter functionality is the same in the Standard and Enterprise Editions. Support for indexing expressions is available only in the Enterprise Edition because it requires bitmap index support.

During installation of the Oracle database, a demonstration script is installed for the Expression Filter feature. The script `exfdemo.sql` is located in the $ORACLE_HOME/rdbms/demo/ directory.

Index